UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T Z.200

(11/1999)

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

SERIE Z: LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

Lenguajes de programación – CHILL: El lenguaje de programación del UIT-T

CHILL – El lenguaje de programación del UIT-T

Recomendación UIT-T Z.200

(Anteriormente Recomendación del CCITT)

RECOMENDACIONES UIT-T DE LA SERIE Z

LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

TÉCNICAS DE DESCRIPCIÓN FORMAL	
Lenguaje de especificación y descripción	Z.100–Z.109
Aplicación de técnicas de descripción formal	Z.110–Z.119
Gráficos de secuencias de mensajes	Z.120–Z.129
LENGUAJES DE PROGRAMACIÓN	
CHILL: El lenguaje de programación del UIT-T	**Z.200–Z.209**
LENGUAJE HOMBRE-MÁQUINA	
Principios generales	Z.300–Z.309
Sintaxis básica y procedimientos de diálogo	Z.310–Z.319
LHM ampliado para terminales con pantalla de visualización	Z.320–Z.329
Especificación de la interfaz hombre-máquina	Z.330–Z.399
CALIDAD DE SOPORTES LÓGICOS DE TELECOMUNICACIONES	Z.400–Z.499
MÉTODOS PARA VALIDACIÓN Y PRUEBAS	Z.500–Z.599

Para más información, véase la Lista de Recomendaciones del UIT-T.

NORMA INTERNACIONAL 9496

RECOMENDACIÓN UIT-T Z.200

CHILL – EL LENGUAJE DE PROGRAMACIÓN DEL UIT-T

Resumen

Esta Recomendación | Norma Internacional define el lenguaje de programación CHILL del UIT-T. El CHILL es un lenguaje fuertemente tipificado, estructurado en bloques y diseñado primordialmente para la implementación de grandes y complejos sistemas modulares.

El CHILL se diseñó para proporcionar fiabilidad y eficacia en tiempo de ejecución, al mismo tiempo que suficiente flexibilidad y potencia para abarcar la gama de aplicaciones requeridas. El CHILL también ofrece medios que estimulan el desarrollo por piezas y modulado de grandes sistemas.

Orígenes

La Recomendación UIT-T Z.200 ha sido preparada por la Comisión de Estudio 10 (1997-2000) del UIT-T y aprobada el 19 de noviembre de 1999. Un texto idéntico también se publica como ISO/CEI 9496.

PREFACIO

La UIT (Unión Internacional de Telecomunicaciones) es el organismo especializado de las Naciones Unidas en el campo de las telecomunicaciones. El UIT-T (Sector de Normalización de las Telecomunicaciones de la UIT) es un órgano permanente de la UIT. Este órgano estudia los aspectos técnicos, de explotación y tarifarios y publica Recomendaciones sobre los mismos, con miras a la normalización de las telecomunicaciones en el plano mundial.

La Conferencia Mundial de Normalización de las Telecomunicaciones (CMNT), que se celebra cada cuatro años, establece los temas que han de estudiar las Comisiones de Estudio del UIT-T, que a su vez producen Recomendaciones sobre dichos temas.

La aprobación de Recomendaciones por los Miembros del UIT-T es el objeto del procedimiento establecido en la Resolución N.° 1 de la CMNT.

En ciertos sectores de la tecnología de la información que corresponden a la esfera de competencia del UIT-T, se preparan las normas necesarias en colaboración con la ISO y la CEI.

NOTA

En esta Recomendación, la expresión "Administración" se utiliza para designar, en forma abreviada, tanto una administración de telecomunicaciones como una empresa de explotación reconocida de telecomunicaciones.

PROPIEDAD INTELECTUAL

La UIT señala a la atención la posibilidad de que la utilización o aplicación de la presente Recomendación suponga el empleo de un derecho de propiedad intelectual reivindicado. La UIT no adopta ninguna posición en cuanto a la demostración, validez o aplicabilidad de los derechos de propiedad intelectual reivindicados, ya sea por los miembros de la UIT o por terceros ajenos al proceso de elaboración de Recomendaciones.

En la fecha de aprobación de la presente Recomendación, la UIT no ha recibido notificación de propiedad intelectual, protegida por patente, que puede ser necesaria para aplicar esta Recomendación. Sin embargo, debe señalarse a los usuarios que puede que esta información no se encuentre totalmente actualizada al respecto, por lo que se les insta encarecidamente a consultar la base de datos sobre patentes de la TSB.

ÍNDICE

Página

1 Introducción 1
1.1 Generalidades 1
1.2 Examen general del lenguaje 1
1.3 Modos y clases 2
1.4 Localizaciones y sus accesos 3
1.5 Valores y operaciones sobre los mismos 3
1.6 Acciones 4
1.7 Entrada y salida 4
1.8 Manejo de excepciones 4
1.9 Supervisión de tiempo 5
1.10 Estructura del programa 5
1.11 Ejecución concurrente 5
1.12 Propiedades semánticas generales 6
1.13 Opciones de implementación 6

2 Preliminares 7
2.1 Metalenguaje 7
2.2 Vocabulario 8
2.3 Utilización de espacios 9
2.4 Comentarios 9
2.5 Caracteres de formato 9
2.6 Directivas de compilación 10
2.7 Nombres y sus ocurrencias de definición 10

3 Modos y clases 12
3.1 Generalidades 12
3.2 Definiciones de modos 13
3.3 Clasificación de modos 16
3.4 Modos discretos 17
3.5 Modos reales 20
3.6 Modos conjuntistas 22
3.7 Modos referencia 22
3.8 Modos procedimiento 23
3.9 Modos ejemplar 24
3.10 Modos sincronización 25
3.11 Modos entrada-salida 26
3.12 Modos temporización 28
3.13 Modos compuestos 29
3.14 Modos dinámicos 37
3.15 Modos moreta 38

4 Localizaciones y sus accesos 45
4.1 Declaraciones 45
4.2 Localizaciones 47

5 Valores y operaciones sobre los mismos 54
5.1 Definiciones de sinónimos 54
5.2 Valor primitivo 55
5.3 Valores y expresiones 70

Página

6 Acciones ... 79
6.1 Generalidades ... 79
6.2 Acción de asignación ... 79
6.3 Acción condicional ... 81
6.4 Acción de caso ... 81
6.5 Acción hacer ... 83
6.6 Acción salir ... 86
6.7 Acción llamar ... 87
6.8 Acción resultar y acción retornar ... 90
6.9 Acción ir a ... 90
6.10 Acción afirmar ... 91
6.11 Acción vacía ... 91
6.12 Acción causar ... 91
6.13 Acción arrancar ... 91
6.14 Acción parar ... 91
6.15 Acción continuar ... 92
6.16 Acción demorar ... 92
6.17 Acción demorar y elegir ... 92
6.18 Acción enviar ... 93
6.19 Acción recibir y elegir ... 94
6.20 Llamadas a rutina incorporada CHILL ... 97

7 Entrada y salida ... 102
7.1 Modelo de referencia E/S ... 102
7.2 Valores de asociación ... 104
7.3 Valores de acceso ... 104
7.4 Rutinas incorporadas para entrada/salida ... 105
7.5 Entrada/salida de texto ... 112

8 Manejo de excepciones ... 120
8.1 Generalidades ... 120
8.2 Manejadores ... 121
8.3 Identificación del manejador ... 121

9 Supervisión de tiempo ... 122
9.1 Generalidades ... 122
9.2 Procesos temporizables ... 122
9.3 Acciones de temporización ... 122
9.4 Rutinas incorporadas relacionadas con el tiempo ... 124

10 Estructura del programa ... 125
10.1 Generalidades ... 125
10.2 Dominios y anidamiento ... 127
10.3 Bloques principio-fin ... 129
10.4 Definiciones de procedimiento ... 129
10.5 Especificaciones y definiciones de proceso ... 134
10.6 Módulos ... 134
10.7 Regiones ... 135
10.8 Programa ... 135
10.9 Asignación de espacio de almacenamiento y tiempo de vida ... 136
10.10 Construcciones para programación por piezas (o programación separada) ... 136
10.11 Genericidad ... 141

Página

11 Ejecución concurrente ... 144
11.1 Procesos, tareas, hilos y sus definiciones ... 144
11.2 Exclusión mutua y regiones ... 145
11.3 Demora de un hilo ... 148
11.4 Reactivación de un hilo ... 148
11.5 Sentencias de definición de señal ... 148
11.6 Compleción de localizaciones de región y de tarea ... 149
12 Propiedades semánticas generales ... 149
12.1 Reglas de modos ... 149
12.2 Visibilidad y vinculación de nombres ... 160
12.3 Selección de caso ... 167
12.4 Definición y resumen de las categorías semánticas ... 169
13 Opciones de implementación ... 173
13.1 Rutinas incorporadas definidas por la implementación ... 173
13.2 Modos enteros definidos por la implementación ... 173
13.3 Modos coma flotante definidos por la implementación ... 173
13.4 Nombres de proceso definidos por la implementación ... 173
13.5 Manejadores definidos por la implementación ... 173
13.6 Nombres de excepción definidos por la implementación ... 173
13.7 Otras características definidas por la implementación ... 173
Apéndice I – Juego de caracteres CHILL ... 175
Apéndice II – Símbolos especiales ... 176
Apéndice III – Cadenas de nombre simple especiales ... 177
III.1 Cadenas de nombre simple reservadas ... 177
III.2 Cadenas de nombre simple predefinidas ... 178
III.3 Nombres de excepción ... 178
Apéndice IV – Ejemplos de programas ... 179
IV.1 Operaciones sobre enteros ... 179
IV.2 Las mismas operaciones sobre fracciones ... 179
IV.3 Las mismas operaciones sobre números complejos ... 180
IV.4 Aritmética de orden general ... 180
IV.5 Adición bit por bit y verificación del resultado ... 180
IV.6 Operaciones con fechas ... 181
IV.7 Numerales romanos ... 182
IV.8 Cuenta de letras en una cadena de caracteres de longitud arbitraria ... 183
IV.9 Números primos ... 184
IV.10 Implementación de pilas de dos formas distintas, transparentes para el usuario ... 184
IV.11 Fragmento para jugar al ajedrez ... 185
IV.12 Construcción y manejo de una lista circularmente enlazada ... 188
IV.13 Una región para manejar accesos competitivos a un recurso ... 189
IV.14 Cola de espera para las llamadas que llegan a una central ... 190
IV.15 Asignar y desasignar un conjunto de recursos ... 190
IV.16 Asignar y desasignar un conjunto de recursos empleando tampones ... 192
IV.17 Explorador de cadena 1 ... 194
IV.18 Explorador de cadena 2 ... 195
IV.19 Supresión de un ítem en una lista doblemente enlazada ... 196
IV.20 Actualizar un registro de un fichero ... 196
IV.21 Fusión de dos ficheros clasificados ... 197
IV.22 Lectura de un fichero con registros de longitud variable ... 198
IV.23 Utilización de módulos de espec ... 199
IV.24 Ejemplo de un contexto ... 199
IV.25 Utilización de prefijación y módulos remotos ... 199

Página

IV.26 Utilización de entrada/salida de texto ... 200
IV.27 Una pila genérica ... 201
IV.28 Un tipo de datos abstracto ... 202
IV.29 Ejemplo de un módulo de espec ... 202
IV.30 Orientación a objetos: modos para pilas simples, secuenciales ... 202
IV.31 Orientación a objetos: extensión de modo: pila simple, secuencial con operación "top" ... 204
IV.32 Orientación a objetos: modos para pilas con sincronización de acceso ... 204

Apéndice V – Características suprimidas ... 206
V.1 Directiva de liberación ... 206
V.2 Sintaxis de los modos enteros ... 206
V.3 Modos conjunto con huecos ... 206
V.4 Sintaxis de los modos procedimiento ... 206
V.5 Sintaxis de modo cadena ... 207
V.6 Sintaxis de los modos matriz ... 207
V.7 Notación de la estructura de niveles ... 207
V.8 Nombres de referencia de correspondencia ... 207
V.9 Declaraciones basadas ... 207
V.10 Literales de cadena de caracteres ... 207
V.11 Expresiones de recibir ... 207
V.12 Notación addr ... 207
V.13 Sintaxis de asignación ... 207
V.14 Sintaxis de la acción de caso ... 207
V.15 Sintaxis de la acción hacer para ... 207
V.16 Contadores de bucle explícitos ... 208
V.17 Sintaxis de la acción llamar ... 208
V.18 Excepción RECURSEFAIL ... 208
V.19 Sintaxis de la acción arrancar ... 208
V.20 Nombres explícitos de valor a recibir ... 208
V.21 Bloques ... 208
V.22 Sentencia de entrada ... 208
V.23 Nombres de registro ... 208
V.24 Atributo recursivo ... 208
V.25 Cuasi sentencias de causa y cuasi manejadores ... 209
V.26 Sintaxis de cuasi sentencias ... 209
V.27 Nombres débilmente visibles y sentencias de visibilidad ... 209
V.28 Nombres débilmente visibles y sentencias de visibilidad ... 209
V.29 Infiltrabilidad ... 209
V.30 Toma mediante nombre modulión ... 209
V.31 Cadenas de nombre simple predefinidas ... 209

Apéndice VI – Índice de las reglas de producción ... 210

NORMA INTERNACIONAL

RECOMENDACIÓN UIT-T

CHILL – EL LENGUAJE DE PROGRAMACIÓN DEL UIT-T

1 Introducción

Esta Recomendación | Norma Internacional define el lenguaje de programación del UIT-T CHILL. Cuando CHILL fue definido por primera vez en1980, "CHILL" eran las siglas inglesas de la expresión Lenguaje de alto nivel del CCITT (CCITT High Level Language).

En las subcláusulas siguientes de esta cláusula se exponen algunos de los fundamentos del diseño de este lenguaje y se examinan las características generales del mismo.

En los manuales "Introduction to CHILL" y "CHILL user's manual" puede verse información sobre el diverso material de introducción y de formación sobre este tema.

El manual titulado "Formal definition of CHILL" presenta una definición alternativa de CHILL, en una forma matemática estricta (basada en la notación VDM).

1.1 Generalidades

CHILL es un lenguaje estructurado en bloques, sumamente tipificado, concebido sobre todo para la implementación de amplios y complejos sistemas insertados.

CHILL se diseñó con los siguientes objetivos:

- mejorar la fiabilidad y la eficacia en la ejecución mediante una amplia utilización de comprobaciones durante la compilación;
- ser lo suficientemente flexible y potente para atender la necesaria gama de aplicaciones y explotar diversas modalidades de soporte físico;
- proporcionar facilidades que estimulen el desarrollo por piezas y modular de sistemas extensos;
- responder a implementaciones en tiempo real proporcionando funciones primitivas incorporadas de concurrencia y supervisión de tiempo;
- permitir la generación de un código objeto de gran eficacia;
- ser fácil de aprender y usar.

El poder expresivo propio del diseño del lenguaje permite que los ingenieros seleccionen las construcciones apropiadas tomándolas de un rico conjunto de facilidades, de manera que la implementación resultante se ajuste de una manera más precisa a la especificación original.

Dado que CHILL distingue cuidadosamente entre objetos estáticos y dinámicos, casi todas las comprobaciones semánticas pueden realizarse durante la compilación, lo que evidentemente repercute en beneficio de la ejecución. La violación de las reglas dinámicas CHILL produce excepciones en la ejecución, que pueden ser interceptadas por un manejador de excepciones apropiado (no obstante, la generación de esas comprobaciones implícitas es facultativa, a menos que se especifique explícitamente un manejador definido por el usuario).

CHILL permite escribir programas independientemente de la máquina. El propio lenguaje es independiente de la máquina; sin embargo, determinados sistemas de compilación pueden exigir la provisión de objetos específicos definidos por la implementación. Debe señalarse que, en general, los programas que contienen esos objetos no serán portables.

1.2 Examen general del lenguaje

Un programa CHILL consta esencialmente de tres partes:

- una descripción de los objetos de datos;
- una descripción de las acciones que han de efectuarse sobre los objetos de datos;
- una descripción de la estructura del programa.

Los objetos de datos se describen mediante sentencias de datos (sentencias de declaración y de definición); las acciones se describen mediante sentencias de acción, y la estructura del programa se determina mediante sentencias de estructuración del programa.

Los objetos de datos manipulables de CHILL son valores y localizaciones en las que pueden almacenarse valores. Las acciones definen las operaciones que han de efectuarse sobre los objetos de datos y el orden en que los valores se almacenan y se extraen de las localizaciones. La estructura del programa determina el tiempo de vida y la visibilidad de los objetos de datos.

CHILL permite una amplia comprobación estática de la utilización de objetos de datos en un contexto dado.

En las subcláusulas siguientes se ofrece un resumen de los diversos conceptos CHILL. Cada subcláusula es una introducción a una cláusula de igual título, que describe el concepto en detalle.

1.3 Modos y clases

Una localización tiene asociado un modo. El modo de una localización define el conjunto de valores que pueden residir en esa localización, así como otras propiedades asociadas con ella (obsérvese que no todas las propiedades de una localización son determinables por su modo solamente). Propiedades de las localizaciones son: tamaño, estructura interna, propiedad de "lectura solamente", referenciabilidad, etc. Propiedades de los valores son: representación interna, ordenación, operaciones aplicables, etc.

Un valor tiene asociada una clase. La clase de un valor determina los modos de las localizaciones que pueden contener dicho valor.

CHILL proporciona las siguientes categorías de modos:

- modos discretos: modos (simbólicos) entero, carácter, booleano, conjunto e intervalos correspondientes;
- modos reales: modos de coma flotante e intervalos de los mismos;
- modos conjuntistas: conjuntos de elementos de algún modo discreto;
- modos referencia: referencias ligadas, referencias libres y descriptores utilizados como referencias a localizaciones;
- modos compuestos: modos cadena, matriz y estructura;
- modos procedimiento: procedimientos considerados como objetos de datos manipulables;
- modos ejemplar: identificaciones de procesos;
- modos sincronización: modos suceso y tampón para sincronización de procesos y comunicación;
- modos entrada-salida: modos asociación, acceso y texto para operaciones de entrada-salida;
- modos temporización: modos duración y tiempo absoluto para supervisión de tiempo;
- modos moreta: modos módulo, región y tarea para orientación a objeto con herencia individual.

CHILL proporciona denotaciones para un conjunto de modos normalizados. Los modos definidos en el programa pueden introducirse mediante definiciones de modo. Algunas construcciones del lenguaje tienen asociado un denominado modo dinámico. Un modo dinámico es un modo en el que algunas de sus propiedades solamente pueden determinarse en forma dinámica. Los modos dinámicos son siempre modos parametrizados con parámetros determinados en la ejecución. Un modo no dinámico se denomina modo estático.

Con modos moreta, CHILL soporta la programación orientada a objetos de una manera muy versátil. Hay tres géneros de modos para objetos:

- modos módulo los valores de estos modos se comportan de una manera muy similar a los módulos y se parecen, por tanto, en gran medida, a los objetos utilizados en programas clásicos orientados a objetos (por ejemplo Smalltalk, C++, Eiffel, Java);
- modos región los valores de estos modos se comportan de una manera muy similar a regiones. Tales objetos no se encuentran por lo general en programas clásicos orientados a objetos.
- modos tarea: Los valores de estos modos tienen esencialmente la misma estructura que las regiones, pero tienen su propio hilo de control, y la comunicación entre estos y otros objetos se efectúa en forma asíncrona.

Las clases no tienen denotación en CHILL. Se introducen solamente en el metalenguaje para describir condiciones de contexto estáticas y dinámicas.

1.4 Localizaciones y sus accesos

Las localizaciones son lugares (abstractos) donde pueden almacenarse valores o de donde pueden obtenerse valores. Para almacenar u obtener un valor hay que acceder (conseguir acceso) a una localización.

Las sentencias de declaración definen los nombres que deben utilizarse para acceder a una localización. Hay:

1) declaraciones de localización;

2) declaraciones de identidad-loc.

Las primeras crean localizaciones y establecen nombres de acceso para las localizaciones de nueva creación. Las últimas establecen nuevos nombres de acceso para las localizaciones creadas en otra parte.

Además de las declaraciones de localización, pueden crearse nuevas localizaciones mediante llamadas a rutina incorporada *GETSTACK* o *ALLOCATE* que da valores de referencia (véase más adelante) a la localización así creada.

Una localización puede ser **referenciable**. Esto significa que existe para la localización un valor de referencia correspondiente. Este valor de referencia se obtiene como resultado de la operación de referenciación aplicada a la localización **referenciable**. Desreferenciando un valor de referencia se obtiene la localización referida. CHILL requiere que ciertas localizaciones sean **referenciables** y otras no **referenciables**, pero para otras localizaciones se deja a la implementación decidir si serán o no **referenciables**. La referenciabilidad debe ser una propiedad estáticamente determinable de las localizaciones.

Una localización puede ser de **lectura solamente**, lo que significa que solamente puede accederse a la misma para obtener un valor y no para almacenar en ella uno nuevo (excepto en la inicialización).

Una localización puede ser compuesta, lo que significa que tiene sublocalizaciones a las que puede accederse por separado. Una sublocalización no es necesariamente **referenciable**. Una localización que contenga al menos una sublocalización de **lectura solamente** se dice que tiene la **propiedad de lectura solamente**. Los métodos de acceso que dan sublocalizaciones (o subvalores) son la indexación y la segmentación para cadenas y matrices, y la selección para estructuras.

Una localización tiene asociado un modo. Si este modo es dinámico, la localización se denomina localización de modo dinámico.

Las propiedades siguientes de una localización, aunque son estáticamente determinables, no forman parte del modo:

referenciabilidad: si existe o no un valor de referencia para la localización;

clase de almacenamiento: si está o no estáticamente atribuida;

regionalidad: si la localización está o no declarada dentro de una región.

1.5 Valores y operaciones sobre los mismos

Los valores son objetos básicos sobre los que se definen operaciones específicas. Un valor tiene la forma de valor definido (CHILL) o **valor indefinido** (en sentido CHILL). La utilización de un valor indefinido en contextos específicos provoca una situación indefinida (en sentido CHILL), considerándose incorrecto el programa.

CHILL permite utilizar localizaciones en contextos en los que se requieren valores. En este caso, se accede a la localización, para obtener el valor contenido en la misma.

Un valor tiene asociada una clase. Los valores **fuertes** son valores que, además de su clase, tienen también asociado un modo. En ese caso, el valor es siempre uno de los valores definidos por el modo. La clase se utiliza para la verificación de compatibilidad y el modo para describir las propiedades del valor. Algunos contextos requieren que se conozcan esas propiedades, siendo entonces necesario un valor **fuerte**.

Un valor puede ser **literal**, en cuyo caso denota un valor discreto independiente de la implementación, conocido durante la compilación. Un valor puede ser **constante**, en cuyo caso entrega siempre el mismo valor, es decir, sólo tiene que evaluarse una vez. Cuando el contexto requiere un valor **literal** o **constante**, se supone que el valor ha de evaluarse antes de la ejecución, por lo que no puede generar una excepción durante la ejecución. Un valor puede ser **intrarregional**, en cuyo caso puede referirse, de alguna manera a localizaciones declaradas dentro de una región. Un valor puede ser compuesto, es decir, contener subvalores.

Las sentencias de definición de sinónimo establecen nuevos nombres que denotan valores **constantes**.

1.6 Acciones

Las acciones constituyen la parte algorítmica de un programa CHILL.

La acción de asignación almacena un valor (calculado) en una o más localizaciones. La llamada a procedimiento invoca un procedimiento, la llamada a rutina incorporada invoca una rutina incorporada (una rutina incorporada es un procedimiento cuya definición no tiene que estar escrita en CHILL y cuyo mecanismo de transferencia de parámetros y de resultado puede ser más general). Para retornar de una llamada a procedimiento y/o establecer el resultado de la misma se utilizan las acciones retornar y resultar.

Para controlar el flujo secuencial de acciones, CHILL proporciona el siguiente flujo de acciones de control:

– acción condicional: para una bifurcación (simple);

– acción de caso: para una bifurcación múltiple (ramificación). La selección de la rama puede basarse en varios valores; es similar a una tabla de decisión;

– acción hacer: para iteración o encorchetado;

– acción salir: para abandonar de manera estructurada una acción encorchetada o un módulo;

– acción causar: para causar una excepción específica;

– acción ir a: para transferencia incondicional a un punto etiquetado del programa.

Las sentencias de acción y de datos pueden agruparse para formar un módulo o un bloque principio-fin, que forma una acción (compuesta).

Para controlar el flujo de acciones concurrentes, CHILL proporciona las acciones arrancar, parar, demorar, continuar, enviar, demorar y elegir, y recibir y elegir, y las expresiones recibir y arrancar.

1.7 Entrada y salida

Las facilidades de entrada y salida CHILL proporcionan el medio para comunicar con una diversidad de dispositivos del mundo exterior.

El modelo de referencia entrada-salida distingue tres estados. En el estado libre no hay interacción con el mundo exterior.

Mediante una operación *ASSOCIATE* se pasa al estado manejo de ficheros. En el estado manejo de ficheros existen localizaciones de modo asociación, que designan objetos del mundo exterior. Mediante rutinas incorporadas es posible leer y modificar los atributos de asociaciones definidos por el lenguaje, es decir, los atributos **existente**, **legible**, **escribible**, **indexable**, **secuenciable** y **variable**. En el estado manejo de ficheros también se efectúa la creación y el borrado de ficheros.

Mediante la operación *CONNECT* se conecta una localización de modo acceso a una posición de un modo asociación, y se entra en el estado transferencia de datos. La operación *CONNECT* permite el posicionamiento de un índice de **base** en un fichero. En el estado transferencia de datos pueden inspeccionarse diversos atributos de localizaciones de modo acceso y pueden aplicarse las operaciones de transferencia de datos *READRECORD* y *WRITERECORD*.

Mediante las operaciones de transferencia de textos, los valores CHILL pueden representarse en una forma legible por el ser humano, la cual podrá transferirse hacia o desde un fichero o una localización CHILL.

1.8 Manejo de excepciones

Las condiciones semánticas dinámicas CHILL son aquellas condiciones (no independientes del contexto) que, en general, no pueden determinarse estáticamente. (Se deja a la implementación decidir si ha de generarse o no código para probar las condiciones dinámicas durante la ejecución, a menos que se haya especificado explícitamente un manejador adecuado.) La violación de una regla semántica dinámica produce una excepción durante la ejecución; sin embargo, si una implementación puede determinar estáticamente que se violará una condición dinámica, podrá rechazar el programa.

Las excepciones también pueden ser causadas por la ejecución de una acción causar, o condicionalmente, por la ejecución de una acción afirmar. Cuando aparece una excepción en un punto dado de un programa, se transfiere el control al manejador asociado a dicha excepción si es especificable (es decir, si la excepción tiene un nombre) y está especificado. Es posible determinar estáticamente si un manejador está o no especificado para una excepción, en un punto dado. Si no se ha especificado un manejador explícito, el control puede transferirse a un manejador de excepciones definido por la implementación.

Las excepciones tienen nombre, que puede ser un nombre de excepción definido en CHILL, un nombre de excepción definido por la implementación o un nombre de excepción definido por el programa. Obsérvese que cuando se especifica un manejador para un nombre de excepción, debe comprobarse la condición dinámica asociada.

1.9 Supervisión de tiempo

Las facilidades de supervisión de tiempo CHILL proporcionan el medio para reaccionar al transcurso del tiempo en el mundo exterior. Los procesos CHILL sólo podrán interrumpirse en puntos **temporizables** precisos, durante la ejecución. Cuando así sucede, se transfiere el control a un manejador adecuado.

Los programas pueden detectar el transcurso de un periodo de tiempo o sincronizarse con un punto de tiempo absoluto, o a intervalos precisos sin derivas acumuladas. Se proporcionan rutinas incorporadas de tiempo para convertir valores de tiempo y duración en valores enteros, poner un proceso en un estado de espera y detectar la expiración de una supervisión de tiempo.

1.10 Estructura del programa

Las sentencias de estructuración del programa son el bloque principio-fin, módulo, procedimiento, proceso, región y moreta. Las sentencias de estructuración del programa proporcionan los medios de controlar el tiempo de vida de las localizaciones y la visibilidad de los nombres.

El tiempo de vida de una localización es el tiempo durante el cual una localización existe en el programa. Las localizaciones pueden ser explícitamente declaradas (en una declaración de localización) o generadas (llamada a rutina incorporada *GETSTACK* o *ALLOCATE*), o pueden ser implícitamente declaradas o generadas como resultado de la utilización de construcciones del lenguaje.

Se dice que un nombre es **visible** en un cierto punto del programa, si puede utilizarse en ese punto. El alcance de un nombre comprende todos los puntos en los que es **visible**, es decir, donde el objeto designado se identifica por ese nombre.

Los bloques principio-fin determinan la visibilidad de los nombres y el tiempo de vida de las localizaciones.

Los módulos están previstos para restringir la visibilidad de los nombres a fin de protegerlos contra una utilización no autorizada. Mediante las sentencias de visibilidad, es posible ejercer un control sobre la visibilidad de los nombres en diversas partes del programa.

Un procedimiento es un subprograma (posiblemente parametrizado) que puede invocarse (llamarse) en diferentes lugares dentro de un programa. Puede retornar un valor (procedimiento que entrega un valor) o una localización (procedimiento que entrega una localización), o no entregar ningún resultado. En este último caso sólo puede llamarse al procedimiento mediante una acción de llamada a procedimiento.

Los procesos, localizaciones de tarea, regiones y localizaciones de región proporcionan los medios para conseguir una estructura de ejecuciones concurrentes.

Plantillas genéricas proporcionan medios que permiten construir módulos, regiones, procedimientos, procesos y modos moreta genéricos. Estas plantillas pueden ser parametrizadas por constantes, modos y procedimientos SYN. Se utilizan sentencias de ejemplificación genéricas para obtener módulos, regiones, procedimientos, procesos y modos moreta (no genéricos), que se denominan ejemplares genéricos. Un ejemplo genérico se obtiene de una plantilla genérica T reemplazando en T los parámetros genéricos formales por los correspondientes parámetros genéricos efectivos.

Un programa CHILL completo es una lista de unidades de programa que se considera circundada por una definición de proceso (imaginaria). Este proceso más externo es iniciado por el sistema bajo cuyo control se ejecuta el programa. Una unidad de programa puede ser un módulo, una región, una sentencia de definición de sínmodo moreta, una sentencia de definición de neomodo moreta o una plantilla genérica.

Están previstas construcciones para facilitar las diversas formas de desarrollar programas por piezas. Se utilizan un módulo de spec y una región de spec para definir las propiedades estáticas de una pieza de programa, y se usa un contexto para definir las propiedades estáticas de nombres tomados. Además, es posible especificar, mediante la facilidad remoto, que el texto de una pieza de programa se encuentra en otra parte.

1.11 Ejecución concurrente

CHILL permite la ejecución concurrente de unidades de programa. Un hilo (proceso o tarea) es la unidad de ejecución concurrente. La evaluación de una acción arrancar provoca la creación de un nuevo proceso de la definición de proceso indicada. Se considera entonces que el proceso se ejecuta concurrentemente con el proceso de arranque. CHILL permite la activación simultánea de uno o más procesos de igual o diferente definición. La acción parar ejecutada por un proceso provoca su terminación.

Un hilo está siempre en uno de dos estados: activo o demorado. La transición de activo a demorado se denomina demora del proceso. La transición del estado demorado al estado activo se denomina reactivación del hilo. La ejecución de acciones demorar sobre sucesos, o de acciones recibir sobre tampones, o señales, o de acciones enviar sobre tampones, o de una acción llamar a un procedimiento componente de una localización región, o de una acción llamar un procedimiento componente de una localización tarea en caso de que no haya suficiente espacio de almacenamiento para la ejecución pueden hacer que el hilo ejecutante sea demorado. La ejecución de una acción continuar sobre sucesos, o de acciones enviar sobre tampones o señales, o de acciones recibir sobre tampones, o la liberación de una localización región, o, al comienzo de la ejecución, de un procedimiento componente llamado externamente de una localización tarea puede volver nuevamente activo un hilo demorado.

Los tampones y sucesos son localizaciones de uso restringido. Las operaciones enviar, recibir y recibir y elegir se definen sobre tampones, las operaciones demorar, demorar y elegir y continuar se definen sobre sucesos. Los tampones constituyen un modo de sincronizar y transmitir información entre procesos. Los sucesos se utilizan solamente para sincronización. Las señales se definen en sentencias de definición de señal. Éstas denotan funciones para componer y descomponer listas de valores transmitidos entre procesos. Las acciones enviar y las acciones recibir y elegir permiten la comunicación de una lista de valores y la sincronización.

Una región es un tipo especial de módulo. Se utiliza para permitir un acceso mutuamente exclusivo a las estructuras de datos compartidos por varios procesos.

1.12 Propiedades semánticas generales

Las condiciones semánticas (no independientes del contexto) del CHILL son las condiciones de compatibilidad de modos y clases (verificación de modo) y las condiciones de visibilidad (verificación de alcance). Las reglas de verificación de modo determinan cómo pueden utilizarse los nombres, y las reglas de verificación de alcance determinan dónde pueden utilizarse los nombres.

Las reglas de verificación de modo se formulan en términos de requisitos de compatibilidad entre modos, entre clases y entre modos y clases. Los requisitos de compatibilidad entre modos y clases y entre las propias clases se definen en términos de relaciones de equivalencia entre modos. Si intervienen modos dinámicos, la verificación de modos es parcialmente dinámica.

Las reglas de alcance definen la visibilidad de nombres, que está determinada por la estructura del programa y por sentencias explícitas de visibilidad. Las sentencias explícitas de visibilidad influyen en el alcance de los nombres mencionados. Hay un lugar donde están definidos o declarados los nombres introducidos en un programa. Este lugar se denomina la ocurrencia de definición del nombre. Los lugares donde el nombre se utiliza se denominan ocurrencias aplicadas del nombre. Las reglas de vinculación de nombres asocian una ocurrencia de definición única con cada ocurrencia aplicada del nombre.

1.13 Opciones de implementación

CHILL permite utilizar modos enteros definidos por la implementación, rutinas incorporadas definidas por la implementación, nombres de **proceso** definidos por la implementación, manejadores de excepciones definidos por la implementación y nombres de excepción definidos por la implementación.

Un modo entero definido por la implementación debe designarse mediante un nombre de **modo** definido por la implementación. Se considera que este nombre debe definirse en una sentencia de definición de neomodo que no está especificada en CHILL. Dentro del cuadro de reglas sintácticas y semánticas CHILL está permitida la extensión de las operaciones aritméticas existentes definidas en CHILL a los modos enteros definidos por la implementación. Como ejemplos de modos enteros definidos por la implementación pueden citarse los enteros largos, y los enteros cortos.

Una rutina incorporada es un procedimiento cuya definición no tiene que estar escrita en CHILL y que puede tener un esquema de transferencia de parámetros y de transmisión de resultados más general que los procedimientos CHILL.

Un nombre de **proceso** incorporado es un nombre de proceso cuya definición no tiene que estar escrita en CHILL, y que puede tener un esquema de transferencia de parámetros más general que los procesos CHILL. Un proceso CHILL puede cooperar con los procesos definidos por la implementación o iniciar tales procesos.

Un manejador de excepciones definido por la implementación es un manejador añadido a una definición de proceso. Si este manejador recibe el control después de la ocurrencia de una excepción, la implementación decide qué acciones deben efectuarse. Si se viola una condición dinámica definida por la implementación se causa una excepción definida por la implementación.

2 Preliminares

2.1 Metalenguaje

La descripción de CHILL se compone de dos partes:

- la descripción de la sintaxis independiente del contexto;
- la descripción de las condiciones semánticas.

2.1.1 Descripción de la sintaxis independiente del contexto

La sintaxis independiente del contexto se describe utilizando una extensión de la forma de Backus-Naur. Las categorías sintácticas se indican mediante una o más palabras españolas escritas en cursiva y encerradas entre corchetes angulares (< y >). Este indicador se denomina símbolo no terminal. Para cada símbolo no terminal se da una regla de producción en la sección de sintaxis apropiada. Una regla de producción para un símbolo no terminal consta del símbolo no terminal a la izquierda del símbolo ::=, y una o más construcciones que consisten en símbolos no terminales y/o terminales al lado derecho. Dichas construcciones se separan mediante una barra vertical (|) para designar producciones alternativas para el símbolo no terminal.

A veces el símbolo no terminal incluye una parte subrayada. Esta parte subrayada no forma parte de la descripción independiente del contexto, sino que define una categoría semántica (véase 2.1.2).

Los elementos sintácticos pueden agruparse mediante llaves ({ y }). La repetición de los grupos encerrados en llaves se indica por un asterisco (*) o un signo más ($^+$). Un asterisco indica que el grupo es facultativo y puede repetirse ulteriormente cualquier número de veces; un signo más significa que el grupo tiene que estar presente y puede repetirse ulteriormente cualquier número de veces. Por ejemplo, { *A* }* representa cualquier secuencia de *A*'s, con inclusión de cero, mientras que { *A* }$^+$ representa cualquier secuencia de al menos una *A*. Si los elementos sintácticos se agrupan utilizando corchetes ([y]), el grupo es facultativo. Un grupo encerrado entre llaves o corchetes puede contener una o varias barras verticales que indican elementos sintácticos alternativos.

Se distingue entre sintaxis estricta, para la cual se dan directamente las condiciones semánticas, y sintaxis derivada. Se considera que la sintaxis derivada es una extensión de la sintaxis estricta, y la semántica para la sintaxis derivada se explica indirectamente en términos de la sintaxis estricta asociada.

Debe señalarse que la descripción de sintaxis independiente del contexto se ha elegido de modo que facilite la descripción semántica en esta Recomendación | Norma Internacional y no para que convenga a un algoritmo específico de análisis (por ejemplo, se han introducido ciertas ambigüedades independientes del contexto para mayor claridad). Las ambigüedades se resuelven utilizando la categoría semántica de los elementos sintácticos.

2.1.2 Descripción semántica

Cada categoría sintáctica (símbolo no terminal) se describe en los apartados **semántica**, **propiedades estáticas**, **propiedades dinámicas**, **condiciones estáticas** y **condiciones dinámicas**.

El apartado **semántica** describe los conceptos denotados por las categorías sintácticas (es decir, su significado y comportamiento).

El apartado **propiedades estáticas** define las propiedades semánticas de la categoría sintáctica determinables estáticamente. Se utilizan estas propiedades en la formulación de las condiciones estáticas y/o dinámicas en los apartados donde se utiliza la categoría sintáctica.

El apartado **propiedades dinámicas** define las propiedades de la categoría sintáctica que se conocen solamente en forma dinámica.

El apartado **condiciones estáticas** describe las condiciones comprobables estáticamente, dependientes del contexto, que deben satisfacerse cuando se utiliza la categoría sintáctica. En la sintaxis se expresan algunas condiciones estáticas, mediante una parte subrayada del símbolo no terminal (véase 2.1.1). Esta utilización requiere que el no terminal pertenezca a una categoría semántica específica. Por ejemplo, <*expresión* booleana> es idéntico a <*expresión*> en sentido independiente del contexto, pero semánticamente requiere que *expresión* pertenezca a una clase booleana.

El apartado **condiciones dinámicas** describe las condiciones dependientes del contexto que deben satisfacerse durante la ejecución. En algunos casos, las condiciones son estáticas si no intervienen modos dinámicos. En estos casos, la condición se menciona en el apartado **condiciones estáticas** y se hace referencia a ella en el apartado **condiciones dinámicas**. En otros casos, las condiciones dinámicas pueden comprobarse estáticamente; una implementación puede tratar esto como una violación de una condición estática.

En la descripción semántica pueden utilizarse diferentes tipos de caracteres de las siguientes maneras: se utilizan caracteres en cursiva (sin < y >) para indicar objetos sintácticos; los términos correspondientes en caracteres román indican objetos semánticos correspondientes (por ejemplo, una *localización* denota una localización). Las negritas se utilizan para denominar propiedades semánticas; algunas veces una propiedad puede expresarse sintácticamente y también semánticamente (por ejemplo, la oración "la *expresión* es **constante**" significa lo mismo que "la *expresión* es una *expresión constante*").

A menos que se especifique otra cosa, la semántica, las propiedades y las condiciones descritas en el apartado de una categoría sintáctica se cumplen, independientemente del contexto en que esa categoría sintáctica pueda aparecer en otros apartados.

Las propiedades de una categoría sintáctica *A* que tiene una regla de producción de la forma *A::=B*, donde *B* es una categoría sintáctica, son las mismas que las de *B* si no se especifica otra cosa.

En esta Recomendación | Norma Internacional se introducen nombres virtuales para describir modos, localizaciones y valores que no ocurren explícitamente en el texto de programa. En tales casos el nombre va precedido de un símbolo de 'y' comercial (&). Estos nombres se presentan con fines exclusivamente descriptivos.

2.1.3 Ejemplos

En la mayor parte de los apartados de sintaxis hay un apartado **ejemplos** que da uno o más ejemplos de las categorías sintácticas definidas. Estos ejemplos se han extraído de un conjunto de ejemplos de programas contenido en el apéndice IV. Las referencias indican por medio de qué reglas sintácticas se obtiene cada ejemplo y de qué ejemplo se toma.

Por ejemplo, *6.20 (d+5)/5 (1.2)* indica un ejemplo de la cadena terminal *(d+5)/5*, obtenido mediante la regla *(1.2)* del apartado sintaxis correspondiente, tomada del ejemplo de programa N.º *6* línea *20*.

2.1.4 Reglas de vinculación en el metalenguaje

A veces la descripción semántica menciona cadenas de nombre simple **especiales** CHILL (véase el apéndice III). Estas cadenas de nombre simple **especiales** se utilizan siempre con su significado CHILL, por lo que no están influidas por las reglas de vinculación de un programa CHILL efectivo.

2.2 Vocabulario

Los programas se representan utilizando el juego de caracteres CHILL (véase el apéndice I) con excepción de los literales de caracteres anchos, los literales de cadena de caracteres anchos y los comentarios. No se especifica la representación de un programa CHILL, lo que significa que también es posible utilizar una representación de carácter de más de un octeto. El alfabeto CHILL está representado por la categoría sintáctica <*carácter*>, de la que puede derivarse como producción terminal cualquier carácter que forme parte del juego de caracteres CHILL. Los caracteres del nivel 1 del juego universal de caracteres 2 (UCS-2) se representan por la categoría sintáctica <*carácter ancho*>, de la cual puede derivarse como producción terminal cualquier carácter pertinente al nivel 1 del UCS-2.

Los elementos léxicos de CHILL son:

- símbolos especiales;
- cadenas de nombre simple;
- literales.

Los símbolos se indican en el apéndice II. Pueden estar constituidos por un solo carácter o por combinaciones de caracteres.

Las cadenas de nombre simple se forman de acuerdo con la siguiente **sintaxis:**

sintaxis:

<cadena de nombre simple> ::= *(1)*
<*letra*> { <*letra*> | <*dígito*> | _ }* *(1.1)*

<letra> ::= *(2)*
A | B | C | D | E | F | G | H | I | J | K | L | M *(2.1)*
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z *(2.2)*
| a | b | c | d | e | f | g | h | i | j | k | l | m *(2.3)*
| n | o | p | q | r | s | t | u | v | w | x | y | z *(2.4)*

<digit> ::= *(3)*
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 *(3.1)*

semántica: El carácter de subrayado (_) forma parte de la cadena de nombre simple, es decir, la cadena de nombre simple *life_time* es diferente de la cadena de nombre simple *lifetime*. Las letras mayúsculas y minúsculas son diferentes, es decir, *Estado* y *estado* son dos cadenas de nombre simple diferentes.

El lenguaje tiene varias cadenas de nombre simple **especiales** con significados predeterminados (véase el apéndice III). Algunas de ellas están **reservadas**, es decir, no pueden utilizarse para otros fines.

Las cadenas de nombre simple **especiales** de una pieza deben estar representadas todas en mayúsculas o todas en minúsculas. Las cadenas de nombre simple **reservadas** solamente lo están en la representación elegida (por ejemplo, si se eligen las minúsculas, **row** estará reservado y **ROW** no).

condiciones estáticas: Una *cadena de nombre simple* no puede ser una de las cadenas de nombre simple **reservadas** (véase III.1).

2.3 Utilización de espacios

Está permitida una secuencia de uno o más espacios antes y después de cada elemento léxico. Tal secuencia se denomina un delimitador. Los elementos léxicos son también terminados por el primer carácter que no puede formar parte del elemento léxico. Por ejemplo, *IFBTHEN* se considerará una *cadena de nombre simple* y no el comienzo de una acción **IF** B **THEN**, //* se considerará que es el símbolo de concatenación (//) seguido de un asterisco (*), y no el símbolo de división (/) seguido de un paréntesis de apertura de comentario (/*).

2.4 Comentarios

sintaxis:

<comentario> ::= *(1)*
<comentario encorchetado> *(1.1)*
| *<comentario de fin de lína>* *(1.2)*

<comentario encorchetado> ::= *(2)*
/ <cadena de caracteres> */* *(2.1)*

<comentario de fin de línea> ::= *(3)*
– – <cadena de caracteres> <fin de línea> *(3.1)*

<cadena de caracteres> ::= *(4)*
*{ <carácter> }** *(4.1)*
| *{ <carácter ancho> }** *(4.2)*

NOTA – *fin de línea* denota el final de la línea en la que está el comentario.

semántica: Un *comentario* da información al lector de un programa. No tiene influencia sobre la semántica del programa.

Un *comentario* puede insertarse en todos los lugares donde estén permitidos espacios como delimitadores.

Un *comentario encorchetado* es terminado por la primera ocurrencia de la secuencia especial: */. Un *comentario de fin de línea* es terminado por la primera ocurrencia del fin de la línea.

ejemplo:

4.1 */* from collected algorithms from CACM n.º 93 */* *(2.1)*

2.5 Caracteres de formato

Los caracteres de formato retroceso (BS, *backspace*), retorno del carro (CR, *carriage return*), página siguiente (FF, *form feed*), tabulación horizontal (HT, *horizontal tabulation*), cambio de renglón (LF, *line feed*), y tabulación vertical (VT, *vertical tabulation*) del juego de caracteres CHILL (véase el apéndice I, posiciones FE_0 a FE_5) y el *fin de línea* no se mencionan en la descripción de la sintaxis independiente del contexto CHILL. Cuando se emplean, tienen el mismo efecto delimitador que un espacio. Los espacios y los caracteres de formato no pueden aparecer dentro de elementos léxicos (salvo los literales de cadena de caracteres).

2.6 Directivas de compilación

sintaxis:

<cláusula directiva> ::= *(1)*
< > <directiva> {, <directiva> } < >* *(1.1)*

<directiva> ::= *(2)*
<directiva de implementación> *(2.1)*

semántica: Una cláusula directiva da información al compilador. Esta información se especifica en un formato definido por la implementación.

Una directiva de implementación no puede influir en la semántica del programa, es decir, un programa con directivas de implementación es correcto, en sentido CHILL, si y sólo si es correcto sin estas directivas.

Una *cláusula directiva* es terminada por la primera ocurrencia del símbolo de fin de directiva (< >). Una *directiva* puede contener cualquier carácter del juego de caracteres (véase el apéndice I).

propiedades estáticas: Una *cláusula directiva* puede insertarse en cualquier lugar donde se permitan espacios. Tiene el mismo efecto delimitador que un espacio. Los nombres utilizados en una *cláusula directiva* siguen un esquema de vinculación de nombres definido por la implementación que no influye en las reglas de vinculación de nombres CHILL (véase 12.2).

2.7 Nombres y sus ocurrencias de definición

sintaxis:

<nombre> ::= *(1)*
<cadena de nombre> *(1.1)*
| *<nombre calificado>* *(1.2)*
| *<nombre de componente moreta>* *(1.3)*

<cadena de nombre> ::= *(2)*
<cadena de nombre simple> *(2.1)*
| *<cadena de nombre prefijada>* *(2.2)*

<cadena de nombre prefijada> ::= *(3)*
<prefijo> ! <cadena de nombre simple> *(3.1)*

<prefijo> ::= *(4)*
*<prefijo simple> { ! <prefijo simple> }** *(4.1)*

<prefijo simple> ::= *(5)*
<cadena de nombre simple> *(5.1)*

<ocurrencia de definición> ::= *(6)*
<cadena de nombre simple> *(6.1)*

<lista de ocurrencias de definición> ::= *(7)*
*<ocurrencia de definición> { , <ocurrencia de definición>}** *(7.1)*

<nombre de elemento de conjunto> ::= *(8)*
<cadena de nombre simple> *(8.1)*

<ocurrencia de definición de nombre de elemento de conjunto> ::= *(9)*
<cadena de nombre simple> *(9.1)*

<nombre de campo> ::= *(10)*
<cadena de nombre simple> *(10.1)*

<ocurrencia de definición de nombre de campo> ::= *(11)*
<cadena de nombre simple> *(11.1)*

<lista de ocurrencias de definición de nombre de campo>::= *(12)*
*<ocurrencia de definición de nombre de campo>{ , <ocurrencia de definición de nombre de campo> }** *(12.1)*

<nombre de excepción> ::= *(13)*
<cadena de nombre simple> *(13.1)*
| *<cadena de nombre prefijada>* *(13.2)*

<nombre de referencia de texto> ::= *(14)*
<cadena de nombre simple> *(14.1)*
| *<cadena de nombre prefijada>* *(14.2)*

<nombre de componente> ::= *(15)*
<cadena de nombre simple> *(15.1)*

<ocurrencia de definición de nombre de componente> ::= *(16)*
<cadena de nombre simple> *(16.1)*

<nombre calificado> ::= *(17)*
<cadena de nombre simple> ! *<nombre de componente>* *(17.1)*

<nombre de componente moreta> ::= *(18)*
<localización moreta> . { *<cadena de nombre simple>* | *<nombre calificado>*} *(18.1)*

semántica: Los nombres en un programa denotan objetos. Dada una ocurrencia de un *nombre* (formalmente: una ocurrencia de una producción terminal de *nombre*) en un programa, las reglas de vinculación de 12.2 proporcionan *ocurrencias de definición* (formalmente: ocurrencias de producciones terminales de *ocurrencia de definición*) a las que ese *nombre* (o esa ocurrencia de *nombre*) está **ligado**. El *nombre* denota entonces el objeto definido o declarado por las *ocurrencias de definición*. (Puede haber más de una *ocurrencia de definición* de un *nombre* en el caso de *nombres* con **cuasi** *ocurrencias de definición* y en el caso de *nombres* de componentes de modos moreta.)

Se dice que las *ocurrencias de definición* definen el *nombre*. Se dice que un *nombre* es una ocurrencia aplicada del nombre creado por la *ocurrencia de definición* a la que está **ligado**. La *cadena de nombre simple* más a la derecha, del *nombre*, es igual a la del nombre.

Análogamente, los nombres de campo están **ligados** a las ocurrencias de definición de nombre de campo y denotan los campos (de un modo estructura) definidos por esas ocurrencias de definición de nombre de campo. Los *nombres de componentes moreta* están ligados a *ocurrencias de definición de componente* y denotan los componentes (de un modo moreta) definidos por esas *ocurrencias de definición de nombre de componente*.

Los nombres de excepción se utilizan para identificar los manejadores de excepción de acuerdo con las reglas establecidas en la cláusula 8.

Los nombres de referencia de texto se utilizan para identificar descripciones de piezas de texto fuente en una forma definida por la implementación, según las reglas indicadas en 10.10.1.

Cuando un nombre está **ligado** a más de una ocurrencia de definición, cada una de las ocurrencias de definición a la que está **ligado** el nombre define o declara el mismo objeto (véanse las reglas precisas en 10.10 y 12.2.2).

Se utilizan *nombres calificados* para identificar componentes de *modos moreta*.

definición de notación: Dada una *cadena de nombre* NS, y una cadena de caracteres P, que es un *prefijo* o está vacía, el resultado de prefijar NS con P, que se escribe P ! NS, se define como sigue:

- si P está vacía, entonces P ! NS es NS;
- en otro caso, P ! NS es la cadena de nombre obtenida concatenando todos los caracteres de P, un operador de prefijación y todos los caracteres de NS.

Por ejemplo, si P es "*q ! r*" y NS es "*s ! n*", entonces P ! NS es "*q ! r ! s ! n*".

propiedades estáticas: Cada *cadena de nombre simple* tiene asociada una cadena de nombre **canónica** que es la propia *cadena de nombre simple*. Una *cadena de nombre* tiene asociada una cadena de nombre **canónica** que es lo siguiente:

- si la *cadena de nombre* es una *cadena de nombre simple*, entonces es la cadena de nombre **canónica** de esa *cadena de nombre simple*;
- si la *cadena de nombre* es una *cadena de nombre prefijada*, entonces es la concatenación de izquierda a derecha de todas las *cadenas de nombre simple* de la *cadena de nombre*, separadas por operadores de prefijación, es decir, los espacios intercalados, los comentarios y los determinantes de formato (si los hubiere) se dejan fuera.

En el resto de esta Recomendación | Norma Internacional:

- la cadena de nombre de un *nombre*, *nombre de excepción*, o *nombre de referencia de texto* se utiliza para denotar la cadena de nombre **canónica** de la *cadena de nombre* de ese *nombre*, *nombre de excepción* o *nombre de referencia de texto*, respectivamente.
- la cadena de nombre de una *ocurrencia de definición*, *nombre de campo*, *ocurrencia de definición de nombre de campo*, *nombre de componente moreta* u *ocurrencia de definición de componente moreta* se utiliza para denotar la cadena de nombre **canónica** de la *cadena de nombre simple* en esa *ocurrencia de definición*, *nombre de campo*, *ocurrencia de definición de nombre de campo*, *nombre de componente moreta* u *ocurrencia de definición de componente moreta*, respectivamente.

Las reglas de vinculación son tales que:

- los nombres con una cadena de nombre simple están **ligados** a ocurrencias de definición que tienen la misma cadena de nombre;
- los nombres con una cadena de nombre prefijada están **ligados** a ocurrencias de definición que tienen la misma cadena de nombre que la cadena de nombre simple situada más a la derecha en la cadena de nombre prefijada del nombre;
- los nombres de campo están **ligados** a ocurrencias de definición de nombre de campo que tienen la misma cadena de nombre que los nombres de campo.
- Los *nombres de componente moreta* están ligados a *ocurrencias de definición de nombre de componente moreta* que tienen la misma *cadena de nombre* que los *nombres de componente moreta*.

Un *nombre* hereda todas las propiedades estáticas asociadas al nombre definido por la *ocurrencia de definición* a la que está **ligado**. Un *nombre de campo* hereda todas las propiedades estáticas asociadas al nombre de campo definido por la *ocurrencia de definición de nombre de campo* a la que está **ligado**. Un *nombre de componente moreta* hereda todas las propiedades estáticas asociadas al *nombre de componente moreta* definido por la *ocurrencia de definición de nombre de componente moreta* a la que está **ligado**.

condiciones estáticas: La *cadena de nombre simple* denotada por un *nombre calificado* y seguido por ! debe ser un *nombre de modo moreta*.

Si un nombre de la forma "M ! nombre de componente" ocurre fuera de la definición del modo moreta M, el nombre de componente debe ser el nombre de un componente SYN, o de un componente SYNMODE, o de un componente NEWMODE de M.

3 Modos y clases

3.1 Generalidades

Una localización tiene asociado un modo; un valor tiene asociada una clase. El modo asociado a una localización define el conjunto de valores que dicha localización puede contener, los métodos de acceso a la localización y las operaciones permitidas con los valores. La clase asociada a un valor constituye una forma de determinar los modos de las localizaciones que puede contener el valor. Algunos valores son **fuertes**. Un valor **fuerte** tiene asociados una clase y un modo. Se requieren valores **fuertes** en aquellos contextos de valor en los que se necesita información de modo.

3.1.1 Modos

CHILL tiene modos estáticos (es decir, modos en los que todas las propiedades son estáticamente determinables) y modos dinámicos (es decir, modos en los que algunas propiedades se conocen solamente en el momento de la ejecución). Los modos dinámicos son siempre modos parametrizados con parámetros de ejecución.

Los modos estáticos son producciones terminales de la categoría sintáctica *modo*.

Los modos son también parametrizados por valores no denotados explícitamente en el texto del programa.

3.1.2 Clases

Las clases no tienen denotación en CHILL.

Existen los siguientes tipos de clases, y cualquier valor en un programa CHILL tendrá una clase de uno de estos tipos:

Para un modo M, existe la clase M-valuada. Todos los valores de esta clase y sólo éstos son **fuertes**, y el modo asociado al valor es M.

- Para un modo M existe una clase M-derivada.
- Para cualquier modo M existe la clase M-referencia.
- La clase **nula**.
- La clase **general**.

Las dos últimas clases son constantes, es decir, no dependen de un modo M. Se dice que una clase es dinámica si y sólo si es una clase M-valuada, una clase M-derivada o una clase M-referencia, donde M es un modo dinámico.

3.1.3 Propiedades de los modos y de las clases, y relaciones entre los mismos

Los modos CHILL tienen propiedades. Éstas pueden ser hereditarias o no hereditarias. Una propiedad hereditaria se hereda de un modo definidor y pasa a un nombre de **modo** por él definido. A continuación se ofrece un resumen de las propiedades que se aplican a todos los modos (todas, salvo la primera, se definen en 12.1):

- Un modo tiene una **novedad** (definida en 3.2.2, 3.2.3 y 3.3).
- Un modo puede tener la **propiedad de lectura solamente**.
- Un modo puede ser **parametrizable**.
- Un modo puede tener la **propiedad de referenciación**.
- Un modo puede tener la **propiedad parametrizada con marcadores**.
- Un modo M puede tener la **propiedad de no-valor**.

Las clases en CHILL pueden tener las siguientes propiedades (definidas en 12.1):

- Una clase puede tener un modo **raíz**.
- Una o más clases pueden tener una **clase resultante**.

Las operaciones CHILL están determinadas por los modos y las clases de localizaciones y valores. Esto se expresa por las reglas de verificación de modos definidas en 12.1, como cierto número de relaciones entre modos y clases. Existen las siguientes relaciones:

- Dos modos pueden ser **similares**.
- Dos modos pueden ser **v-equivalentes**.
- Dos modos pueden ser **equivalentes**.
- Dos modos pueden ser **l-equivalentes**.
- Dos modos pueden ser **iguales**.
- Dos modos pueden estar **ligados por novedad**.
- Dos modos pueden ser **de lectura compatible**.
- Dos modos pueden ser **de lectura dinámica compatible**.
- Dos modos pueden ser **equivalentes dinámicos**.
- Un modo puede ser **restringible** a un modo.
- Un modo puede ser **compatible** con una clase.
- Una clase puede ser **compatible** con una clase.

3.2 Definiciones de modos

3.2.1 Generalidades

sintaxis:

<definición de modo> ::= *(1)*
<lista de ocurrencias de definición> = <modo definidor> *(1.1)*

<modo definidor> ::= *(2)*
<modo> *(2.1)*

sintaxis derivada: Una *definición de modo* cuya *lista de ocurrencias de definición* consta de más de una *ocurrencia de definición* se deriva de varias definiciones de modo, una para cada *ocurrencia de definición*, separadas por comas, con el mismo *modo definidor*. Por **ejemplo:**

NEWMODE *dollar, pound = INT*;

se deriva de:

NEWMODE *dollar = INT, pound = INT*;

semántica: Una definición de modo define un nombre que denota el modo especificado. Las definiciones de modo ocurren en sentencias de definición de sínmodo y neomodo. Un sínmodo es **sinónimo** de su modo definidor. Un neomodo no es **sinónimo** de su modo definidor. La diferencia se define en términos de la propiedad **novedad**, que se utiliza en la verificación de modo (véase 12.1).

propiedades estáticas: Una *ocurrencia de definición* en una *definición de modo* define un nombre de **modo**.

Los nombres de **modo** predefinidos, los nombres de **modo** entero definidos por la implementación y los nombre de **modo** coma flotante definidos por la implementación (si los hubiere, véanse 3.4.2 y 3.5.1) son también nombres de **modo**.

Un nombre de **modo** tiene un modo **definidor** que es el *modo definidor* en la *definición de modo* que lo define. (Para los nombres de **modo** predefinidos y definidos por la implementación este **modo definidor** es un modo virtual.) Las propiedades hereditarias de un nombre de **modo** son las de su modo **definidor**.

Un conjunto de definiciones recursivas es un conjunto de definiciones de modo o definiciones de sinónimo (véase 5.1) tal que el *modo definidor* en cada *definición de modo* o el *valor constante* o *modo* en cada *definición de sinónimo* es, o contiene directamente, un nombre de **modo** o un nombre de **sinónimo** definido por una definición en el conjunto.

Un conjunto de definiciones recursivas de modo es un conjunto de definiciones recursivas que tienen solamente definiciones de modo.

Un modo que sea, o contenga, un nombre de **modo** definido en un conjunto de definiciones recursivas de modo se dice que denota un modo recursivo. Un camino en un conjunto de definiciones recursivas de modo es una lista de nombres de **modo**, cada uno de los cuales está indicado con un señalador tal que:

- todos los nombres del camino tienen una definición diferente;
- para cada nombre, su sucesor es, u ocurre directamente en, su modo definidor (el sucesor del último nombre es el primer nombre);
- el señalador indica unívocamente la posición del nombre en el modo definidor de su predecesor (el predecesor del primer nombre es el último nombre).

[Ejemplo: **NEWMODE** *M* = **STRUCT** *(i M, n* **REF** *M)*; contiene dos caminos: { M_i } y { M_n }.]

Un camino es **seguro** si y sólo si al menos uno de sus nombres está contenido en un *modo referencia*, un *modo descriptor* o un *modo procedimiento* en el lugar señalado.

condiciones estáticas: Para cualquier conjunto de definiciones recursivas de modo, todos los caminos deben ser **seguros.** (El primer camino del ejemplo anterior no es **seguro.**)

ejemplos:

1.15 *operand_mode = INT* *(1.1)*

3.3 *complex =* ***STRUCT*** *(re, im FLOAT)* *(1.1)*

3.2.2 Definiciones de sínmodo

sintaxis:

<sentencia de definición de sínmodo> ::= *(1)*
SYNMODE *<definición de modo>* {, *<definición de modo>* }*; *(1.1)*
| *<unidad de programa remoto>* *(1.2*

semántica: Una sentencia de definición de sínmodo define nombres de **modo** que son **sinónimos** de su modo definidor.

propiedades estáticas: Una *ocurrencia de definición* en una *definición de modo* en una *sentencia de definición de sínmodo* define un nombre de **sínmodo** (que es también un nombre de **modo**). Se dice que un nombre de **sínmodo** es **sinónimo** de un modo M (o recíprocamente, se dice que M es **sinónimo** del nombre de **sínmodo**) si y sólo si:

- el modo M es el modo **definidor** de nombre de **sínmodo**; o
- el modo **definidor** del nombre de **sínmodo** es en sí un nombre de **sínmodo sinónimo** del modo M.

Dos nombres de modo A y B son **sinónimos** si y sólo si:

- o bien A y B son el mismo nombre;
- o A es el modo **definidor** de B y B es un nombre de **sínmodo**;
- o B es el modo **definidor** de A y A es un nombre de **sínmodo**;
- o el nombre de modo **definidor** de A es **sinónimo** de B y A es un nombre de **sínmodo**.
- o el nombre de modo **definidor** de B es **sinónimo** de A y B es un nombre de **sínmodo**.

La **novedad** de un nombre de **sínmodo** es la de su modo **definidor**.

Si el modo **definidor** es un modo de intervalo discreto o un modo de intervalo de coma flotante, el modo **progenitor** del nombre de **sinónimo** es el de su modo **definidor**. Si el modo **definidor** es un modo cadena **variable**, el modo **componente** del nombre **de sinónimo** es el de su modo **definidor**.

ejemplos:

6.3 **SYNMODE** *month* = **SET** *(jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec)*; *(1.1)*

3.2.3 Definiciones de neomodo

sintaxis:

<sentencia de definición de neomodo> ::= *(1)*
NEWMODE *<definición de modo>* { , *<definición de modo>* }*; *(1.1)*
| *<unidad de programa remota>* *(1.2)*

semántica: Una sentencia de definición de neomodo define nombres de **modo** que no son **sinónimos** de su modo definidor.

propiedades estáticas: Una *ocurrencia de definición* en una *definición de modo* en una *sentencia de definición de neomodo* define un nombre de **neomodo** (que es también un nombre de **modo**).

La **novedad** del nombre de **neomodo** es la *ocurrencia de definición* que lo define. Si el modo **definidor** del nombre de **neomodo** es un modo de intervalo discreto o un modo de intervalo de coma flotante, entonces el modo virtual *&nombre* se introduce como el modo **progenitor** del nombre de **neomodo**. El modo **definidor** del *&nombre* es el modo **progenitor** del modo intervalo discreto o del modo coma flotante, y la **novedad** de *&nombre* es la del nombre de **neomodo**.

Si el modo **definidor** es un modo cadena **variable**, el modo virtual *&nombre* se introduce como el modo **componente** del nombre de **neomodo**. El modo definidor del *&nombre* es el modo **componente** del modo cadena **variable**, y la **novedad** del *&nombre* es la del nombre de **neomodo**.

Si la *ocurrencia de definición* de la definición de modo es una **cuasi** *ocurrencia de definición*, la **novedad** es una **cuasi novedad**, y en otro caso es una **novedad real**.

condiciones estáticas: Si la **novedad** es una **cuasi novedad**, a lo sumo una **novedad real** debe estar **ligada por novedad** a ella.

ejemplos:

11.6 **NEWMODE** *línea = INT (1:8)*; *(1.1)*

11.12 **NEWMODE** *cuadro* = **ARRAY** *(línea)* **ARRAY** *(columna) cuadrado* *(1.1)*

3.3 Clasificación de modos

sintaxis:

<modo> ::= *(1)*
[**READ**] *<modo no compuesto>* *(1.1)*
| [**READ**] *<modo compuesto>* *(1.2)*
| *<indicación de modo genérico formal>* *(1.3)*

<modo no compuesto> ::= *(2)*
<modo discreto> *(2.1)*
| *<modo real>* *(2.2)*
| *<modo conjuntista>* *(2.3)*
| *<modo referencia>* *(2.4)*
| *<modo procedimiento>* *(2.5)*
| *<modo ejemplar>* *(2.6)*
| *<modo sincronización>* *(2.7)*
| *<modo entrada-salida>* *(2.8)*
| *<modo temporización>* *(2.9)*

semántica: Un modo define un conjunto de valores y las operaciones permitidas sobre los mismos. Un modo puede ser un modo de **lectura solamente**, lo que indica que no puede accederse a la localización de ese modo para almacenar un valor. Un modo tiene una **novedad**, que indica si fue o no introducido mediante una sentencia de definición de neomodo.

propiedades estáticas: Un modo tiene las siguientes propiedades hereditarias:

- Es un modo de **lectura solamente** si es un modo de **lectura solamente** explícito o implícito.
- Es un modo de **lectura solamente** explícito si **READ** está especificado o es un modo matriz **parametrizado**, un modo cadena **parametrizado** o un modo estructura **parametrizada**, donde el nombre de modo matriz **origen**, el nombre de modo cadena **origen** o el nombre de modo estructura **variable origen**, respectivamente, que hay en él es un modo de **lectura solamente**.
- Es un modo de **lectura solamente** implícito si no es un modo de **lectura solamente** explícito y si:
  - es el modo **elemento** del modo de un modo cadena de **lectura solamente** o un modo matriz de **lectura solamente** (véanse 3.13.2 y 3.13.3);
  - es un modo **campo** de un modo estructura de **lectura solamente** o es el modo de un campo **marcador** de un modo estructura **parametrizado** (véase 3.13.4).

Un *modo* tiene las mismas propiedades que el *modo no compuesto* o el *modo compuesto* que hay en él. En los puntos siguientes se definen las propiedades de los nombres de **modo** predefinidos y de los *modos* que no son *nombres de modo*; las propiedades de los *nombres de modo* se definen en 3.2. Los modos de **sólo lectura** tienen las mismas propiedades que sus correspondientes modos que no son de **lectura solamente**, excepto la **propiedad** de **lectura solamente** (véase 12.1.1.1).

Un modo tiene las siguientes propiedades no hereditarias:

- Una **novedad** que es o bien **nula** o la *ocurrencia de definición* en una *definición de modo* en una *sentencia de definición de neomodo*. La **novedad** de un modo que no es un *nombre de modo* (ni un *nombre de modo* **READ**) se define como sigue:
  - si es un modo cadena **parametrizado**, un modo matriz **parametrizado** o un modo estructura **parametrizada**, su **novedad** es la de su modo cadena **origen**, modo matriz **origen** o modo estructura **variable origen**, respectivamente;
  - si es un modo intervalo discreto o modo intervalo de coma flotante, su **novedad** es la de su modo **progenitor**;
  - en otro caso, su **novedad** es **nula**.

  La **novedad** de un modo que es un *nombre de modo* (*nombre de modo* **READ**) se define en 3.2.2 y 3.2.3.
- Un **tamaño** que es el valor entregado por *SIZE (&M)*, donde *&M* es un nombre de **sínmodo** virtual **sinónimo** del *modo*.

3.4 Modos discretos

3.4.1 Generalidades

sintaxis:

<modo discreto> ::= *(1)*
<modo entero> *(1.1)*
| *<modo booleano>* *(1.2)*
| *<modo carácter>* *(1.3)*
| *<modo conjunto>* *(1.4)*
| *<modo intervalo discreto>* *(1.5)*

semántica: Un modo discreto define conjuntos y subconjuntos de valores totalmente ordenados.

3.4.2 Modos enteros

sintaxis:

<modo entero> ::= *(1)*
<nombre de modo entero> *(1.1)*

nombres predefinidos: El nombre *INT* está predefinido como un nombre de **modo entero**.

semántica: Un modo entero define un conjunto de valores enteros con signo, comprendidos entre límites definidos por la implementación, sobre los cuales se definen las operaciones de ordenación y aritméticas usuales (véase 5.3). Una implementación puede definir otros modos enteros con límites diferentes (por ejemplo, *LONG_INT, SHORT_INT, UNSIGNED_INT*) que pueden también utilizarse como modos **progenitores** para intervalos (véase 13.2). El modo *&INT* se introduce como el modo virtual que contiene todos los valores de todos los modos enteros **predefinidos** definidos por la implementación. La representación interna de un valor entero es el propio valor entero. Obsérvese que *&INT* no es un modo **predefinido** (aunque puede tener los mismos límites que los de un modo entero **predefinido**).

propiedades estáticas: Un modo entero tiene las siguientes propiedades hereditarias:

- Un **límite superior** y un **límite inferior**, que son los literales que designan respectivamente los valores más alto y más bajo definidos por el modo entero. Están definidos por la implementación.
- Un **número de valores**, que es **límite superior – límite inferior** + 1.

ejemplo:

1.5 INT *(1.1)*

3.4.3 Modos booleanos

sintaxis:

<modo booleano> ::= *(1)*
<nombre de modo booleano> *(1.1)*

nombres predefinidos: El nombre *BOOL* está predefinido como un nombre de **modo booleano**.

semántica: Un modo booleano define los valores lógicos de verdad (*TRUE* y *FALSE*) con las operaciones booleanas usuales (véase 5.3). Las representaciones internas de *FALSE* y *TRUE* son, respectivamente, los valores enteros 0 y 1. La representación define también la ordenación de los valores.

propiedades estáticas: Un modo booleano tiene las siguientes propiedades hereditarias:

- Un **límite superior** que es *TRUE*, y un **límite inferior** que es *FALSE*.
- Un **número de valores** que es 2.

ejemplo:

5.4 BOOL *(1.1)*

3.4.4 Modos carácter

sintaxis:

<modo carácter> ::= *(1)*
<nombre de modo carácter> *(1.1)*

nombres predefinidos: Los nombres *CHAR* y *WCHAR* están predefinidos como nombres de **modo carácter**.

semántica: Un modo carácter define los valores de carácter que se describen en el juego de caracteres CHILL (véase el apéndice I para *CHAR*, o ISO/CEI 10646-1 para *WCHAR*. Estos alfabetos definen la ordenación de los caracteres y los valores enteros que son sus representaciones internas.

propiedades estáticas: Un modo carácter tiene las siguientes propiedades hereditarias:

- Un **límite superior** y un **límite inferior**, que son los literales de carácter que denotan respectivamente los valores más alto y más bajo definidos por *CHAR o WCHAR*, respectivamente.
- Un **número de valores** que es 256 en caso de *CHAR*, y que viene dado por ISO/CEI 10646-1 en el caso de *WCHAR*.

ejemplo:

8.4 CHAR *(1.1)*

3.4.5 Modos conjunto

sintaxis:

<modo conjunto> ::= *(1)*
SET (*<lista de conjunto>*) *(1.1)*
| *<nombre de modo conjunto>* *(1.2)*

<lista de conjunto> ::= *(2)*
<lista de conjunto numerada> *(2.1)*
| *<lista de conjunto no numerada>* *(2.2)*

<lista de conjunto numerada> ::= *(3)*
*<elemento de conjunto numerado> { , <elemento de conjunto numerado> }** *(3.1)*

<elemento de conjunto numerado> ::= *(4)*
<ocurrencia de definición de nombre de elemento de conjunto> =
<expresión literal entera> *(4.1)*

<lista de conjunto no numerada> ::= *(5)*
*<elemento de conjunto> { , <elemento de conjunto> }** *(5.1)*

<elemento de conjunto> ::= *(6)*
<ocurrencia de definición de nombre de elemento de conjunto> *(6.1)*

semántica: Un modo conjunto define un conjunto de valores nominados e innominados. Los valores nominados se designan por los nombres definidos por *ocurrencias de definición* en la *lista de conjunto*; los valores innominados son los restantes. La representación interna de los valores nominados es el valor entero asociado con ellos. Esta representación define también la ordenación de los valores.

El **número de valores** máximo de un modo conjunto está definido por la implementación.

propiedades estáticas: Una *ocurrencia de definición* en una *lista de conjunto* define un nombre de **elemento de conjunto**. Un nombre de **elemento de conjunto** tiene un modo **conjunto** asociado, que es el modo conjunto.

Un modo conjunto tiene las siguientes propiedades hereditarias:

- Un conjunto de nombres de **elemento de conjunto**, que es el conjunto de nombres definido por las *ocurrencias de definición* en su *lista de conjunto*.
- Cada nombre de **elemento de conjunto** de un modo conjunto tiene asociado un valor de representación interna que, en caso de un *elemento de conjunto numerado*, es el valor entregado por la *expresión literal entera* en el mismo; en otro caso, es uno de los valores 0, 1, 2, etc., de acuerdo con su posición en la *lista de conjunto no numerada*. Por **ejemplo:** en **SET** *(a, b)*, *a* tiene asociado un valor de representación 0, y *b* un valor de representación 1.
- Un **límite superior** y un **límite inferior**, que son sus nombres de **elemento de conjunto**, con los valores de representación más alto y más bajo, respectivamente.

- Un **número de valores**, que es el más alto de los valores asociados a los nombres de **elemento de conjunto** más 1.
- Es un modo conjunto **numerado** si la *lista de conjunto* en el mismo es una *lista de conjunto numerada*; en otro caso, es un modo conjunto **no numerado**.

condiciones estáticas: Para cada par de *expresiones literal entera* e_1, e_2 en la *lista de conjunto*, *NUM* (e_1) y *NUM* (e_2) deben entregar resultados no negativos diferentes.

ejemplos:

11.7 **SET** *(ocupado, libre)* *(1.1)*

6.3 *mes* *(1.2)*

3.4.6 Modos intervalo discreto

sintaxis:

<modo intervalo discreto> ::= *(1)*
<nombre de modo discreto> (*<intervalo de literal>*) *(1.1)*
| **RANGE** (*<intervalo de literal>*) *(1.2)*
| **BIN** (*<expresión literal entera>*) *(1.3)*
| *<nombre de modo intervalo discreto>* *(1.4)*

<intervalo literal> ::= *(2)*
<límite inferior> : *<límite superior>* *(2.1)*

<límite inferior> ::= *(3)*
<expresión literal discreta> *(3.1)*

<límite superior> ::= *(4)*
<expresión literal discreta> *(4.1)*

sintaxis derivada: La notación **BIN** *(n)* se deriva de **RANGE** *(0 : 2^n – 1)*. Por ejemplo, **BIN** *(2+1)* corresponde a **RANGE** *(0 : 7)*.

semántica: Un modo intervalo discreto define el conjunto de valores comprendido entre los límites especificados por el *intervalo literal* (límites incluidos). El intervalo se toma de un modo **progenitor** específico que determina las operaciones con los valores de intervalo y la ordenación de los mismos.

propiedades estáticas: Un modo intervalo discreto tiene la siguiente propiedad no hereditaria: tiene un modo **progenitor**, definido como sigue:

- Si el modo intervalo discreto es de la forma:

  <nombre de modo discreto> (*<intervalo literal>*)

  entonces, si el *nombre de modo discreto* no es un modo intervalo discreto, el modo **progenitor** es el *nombre de modo discreto*; en otro caso, es el modo **progenitor** del *nombre de modo discreto*.

- Si el modo intervalo discreto es de la forma:

  RANGE (*<intervalo literal>*)

  entonces el modo **progenitor** depende de la **clase resultante** de las clases del *límite superior* y del *límite inferior* del *intervalo literal*:

  – si es una clase M-derivada, donde M es un modo entero, entonces el modo **progenitor** es un modo entero **predefinido** elegido en la implementación de tal modo que contenga el intervalo de valores entregado por *intervalo literal*;

  – en otro caso es el modo **raíz** de la **clase resultante**.

- Si el modo intervalo discreto es un *nombre de modo intervalo discreto* que es un nombre de **sínmodo**, entonces su modo **progenitor** es el del modo **definidor** del nombre de **sínmodo**; en otro caso, es un nombre de **neomodo**, y su modo **progenitor** es el modo **progenitor** introducido virtualmente (véase 3.2.3).

Un modo intervalo discreto tiene las siguientes propiedades hereditarias:

- Un modo intervalo discreto tiene un **límite superior** y un **límite inferior**, que son los literales que denotan los valores entregados por *límite inferior* y *límite superior*, respectivamente, en *intervalo literal*.
- Un **número de valores**, que es el valor entregado por *NUM (U) – NUM (L)* + 1, donde *U* y *L* designan el **límite superior** y el **límite inferior** del modo intervalo discreto, respectivamente.
- Es un modo intervalo discreto **numerado** si su modo **progenitor** es un modo conjunto **numerado**.

condiciones estáticas: Las clases del *límite superior* y *límite inferior* deben ser **compatibles** entre sí y con el *nombre de modo discreto*, si está especificado.

El *límite inferior* debe entregar un valor menor o igual que el valor proporcionado por el *límite superior*, y ambos valores deben pertenecer al conjunto de valores definido por el *nombre de modo discreto*, si se ha especificado.

La *expresión literal entera* en el caso de **BIN** debe entregar un valor no negativo.

Si el modo **progenitor** es un modo entero, debe existir un modo entero **predefinido** que contenga el conjunto de valores incluido entre el **límite inferior** y el **límite superior**.

Si el modo intervalo discreto es de la forma:

RANGE (<*intervalo literal*>) o <*nombre de modo discreto*> (<*intervalo literal*>)

la evaluación del 1.*límite inferior,* 2.*límite superior*, no podrá depender directa ni indirectamente del valor del 1.**límite inferior**, 2.**límite superior** del modo intervalo discreto. Si el modo intervalo discreto es de la forma:

BIN (<*expresión literal entera*>)

entonces la evaluación de la *expresión literal entera* no podrá depender directa ni indirectamente del valor del **límite superior** del modo intervalo discreto.

ejemplos:

9.5 *INT* (*2:max*) *(1.1)*

11.12 *line* *(1.4)*

3.5 Modos reales

sintaxis:

<*modo real*> ::= *(1)*
<*modo coma flotante*> *(1.1)*
| <*modo intervalo de coma flotante*> *(1.2)*

semántica: Un modo real especifica un conjunto de valores numéricos que aproxima un intervalo continuo de números reales.

3.5.1 Modos de coma flotante

sintaxis:

<*modo coma flotante*> ::= *(1)*
<*nombre de modo coma flotante*> *(1.1)*

nombres predefinidos: El nombre *FLOAT* está predefinido como un nombre de **modo coma flotante**.

semántica: Un modo coma flotante define un número de aproximaciones numéricas a un intervalo de valores reales, junto con su exactitud relativa mínima, entre límites definidos por la implementación en el cual están definidas las operaciones usuales de ordenación y aritméticas (véase 5.3). Este conjunto contiene solamente los valores que pueden ser representados en la implementación. Una implementación puede definir otros modos de coma flotante con diferentes límites y/o **precisión** (por ejemplo, *LONG_FLOAT, SHORT_FLOAT*) que pueden también utilizarse como modos **progenitores** para intervalos (véase 13.3). El modo *&FLOAT* se introduce como el modo virtual que contiene todos los valores de todos los modos predefinidos de coma flotante definidos por la implementación. La representación interna de un valor de coma flotante es el propio valor de coma flotante. Obsérvese que *&FLOAT* no es un modo **predefinido** (aunque puede tener los mismos límites que los de un modo coma flotante **predefinido**).

propiedades estáticas: Un modo de coma flotante tiene las siguientes propiedades hereditarias:

- Un **límite superior** y un **límite inferior** que son los literales que denotan respectivamente el valor más alto y el más bajo definidos por el modo coma flotante. Son definidos por la implementación.
- Una **precisión** que es el número máximo de cifras decimales significativas definidas por el modo.

- Un **límite inferior positivo** y un **límite superior negativo** que son los literales que denotan respectivamente el valor positivo más pequeño y el valor negativo más grande representables exactamente en el modo coma flotante, excluido el cero.

ejemplo:

FLOAT *(1.1)*

3.5.2 Modos intervalo de coma flotante

sintaxis:

<modo intervalo de coma flotante> ::= *(1)*
<nombre de modo coma flotante> (<intervalo de valor float>) *(1.1)*
| **RANGE** *(<intervalo de valor float> [, <cifras significativas>])* *(1.2)*
| *<nombre de modo intervalo de coma flotante>* *(1.3)*

<intervalo de valor float> ::= *(2)*
<límite float inferior> : <límite float superior> *(2.1)*

<límite float inferior> ::= *(3)*
<expresión literal de coma flotante> *(3.1)*

<límite float superior> ::= *(4)*
<expresión literal de coma flotante> *(4.1)*

<cifras significativas> ::= *(5)*
<expresión literal de entero> *(5.1)*

semántica: Un modo intervalo de coma flotante define el conjunto de valores comprendidos entre los límites especificados (incluidos los límites) por *intervalo de valor float* con el número de cifras significativas especificado por *cifras significativas*. El intervalo se toma de un modo **progenitor** específico que determina las operaciones sobre los valores del intervalo y la ordenación de dichos valores. Por ejemplo, **RANGE** (–10,0*E*1 : 10,0*E*1, 2) denota los valores: –10,0, –9,9, ..., –0,11, –0,1, 0, 0,1, ..., 10,0.

propiedades estáticas: Un modo intervalo de coma flotante tiene la siguiente propiedad no hereditaria: tiene un modo **progenitor**, definido como sigue:

- Si el modo intervalo de coma flotante es de la forma:

  <nombre de modo coma flotante> (*<intervalo de valor float>*)

  entonces si el *nombre de modo coma flotante* no es un modo intervalo de coma flotante, el modo **progenitor** es el *nombre de modo coma flotante*; en otro caso es el modo **progenitor** del *nombre de modo coma flotante*.

- Si el modo intervalo de coma flotante es de la forma:

  RANGE (*<intervalo de valor float>* [, *<cifras significativas>*])

  entonces el modo **progenitor** depende de la **clase resultante** de las clases del límite float superior y límite float inferior en el intervalo de literal:

  – si es una clase M-derivada, donde M es un modo coma flotante, entonces el modo **progenitor** es un modo coma flotante **predefinido** elegido en la implementación de tal modo que contenga el intervalo de valores entregado por *intervalo de valores float*, con la **precisión** definida más abajo;

  – en otro caso es el modo **raíz** de la **clase resultante**.

- Si el modo intervalo de coma flotante es un *nombre de modo intervalo de coma flotante* que es un nombre de **sínmodo**, entonces su modo **progenitor** es el de un modo **definidor** del nombre de **sínmodo**; en otro caso es un nombre de **neomodo** y entonces su modo **progenitor** es el modo **progenitor** introducido virtualmente (véase 3.2.3).

Un modo intervalo de coma flotante tiene las siguientes propiedades hereditarias:

- Un **límite superior** y un **límite inferior** que son los literales que denotan los valores entregados por *límite float inferior* y *límite float superior*, respectivamente, en el *intervalo de valores float*.

- Una **precisión** que es, si el modo intervalo de coma flotante es de la forma:

  RANGE (*<intervalo de valor float>* [, *<cifras significativas>*])

  – el valor entregado por cifras significativas si están especificadas;

  – en otro caso, la **precisión** más elevada de las **precisiones** de *límite float inferior* y *límite float superior*.

  En otro caso es la del *nombre de modo coma flotante* o la del *nombre de modo intervalo de coma flotante*.

condiciones estáticas: *Límite float inferior* debe entregar un valor que sea menor que o igual al valor entregado por *límite float superior*, y ambos valores deben pertenecer al conjunto de valores definidos por *nombre de modo coma flotante*, si está especificado.

Debe existir un modo coma flotante **predefinido** que contenga el **límite superior** y el **límite inferior** con la **precisión** especificada.

El valor entregado por *cifra significativa* debe ser mayor que cero.

La evaluación del 1.*límite inferior,* 2.*límite superior*, no podrá depender directa ni indirectamente del valor del 1.**límite inferior**, 2.**límite superior** del modo intervalo de coma flotante.

3.6 Modos conjuntistas

sintaxis:

<modo conjuntista> ::= *(1)*
POWERSET *<modo miembro>* *(1.1)*
| *<nombre de modo conjuntista>* *(1.2)*

<modo miembro> ::= *(2)*
<modo discreto> *(2.1)*

semántica: Un modo conjuntista define valores que son conjuntos de valores de su modo miembro. Los modos conjuntistas se extienden sobre todos los subconjuntos del modo miembro. Con los valores conjuntistas se definen los usuales operadores teóricos de conjuntos (véase 5.3).

El **número de valores** máximo de un modo miembro está definido por la implementación.

propiedades estáticas: Un modo conjuntista tiene la siguiente propiedad hereditaria:

- Un modo **miembro** que es el *modo miembro*.

ejemplos:

8.4 **POWERSET** *CHAR* *(1.1)*

9.5 **POWERSET** *INT (2:max)* *(1.1)*

9.6 *number_list* *(1.2)*

3.7 Modos referencia

3.7.1 Generalidades

sintaxis:

<modo referencia> ::= *(1)*
<modo referencia ligada> *(1.1)*
| *<modo referencia libre>* *(1.2)*
| *<modo descriptor>* *(1.3)*

semántica: Un modo referencia define referencias (direcciones o descriptores) a localizaciones **referenciables**. Por definición, las referencias ligadas corresponden a localizaciones de un modo estático dado o a un conjunto dado de modos moreta; las referencias libres pueden corresponder a localizaciones de cualquier modo estático, los descriptores corresponden a localizaciones de un modo dinámico.

La operación de desreferenciación se define sobre valores de referencia (véanse 4.2.3, 4.2.4 y 4.2.5), y entrega la localización referenciada.

Dos valores de referencia son iguales si y sólo si se refieren ambos a la misma localización o si no se refieren a localización alguna (por ejemplo, son el valor *NULL*).

3.7.2 Modos referencia ligada

sintaxis:

<modo referencia ligada> ::= *(1)*
REF *<modo referenciado>* *(1.1)*
| *<nombre de modo referencia ligada>* *(1.2)*

<modo referenciado> ::= *(2)*
<modo> *(2.1)*

semántica: Un modo referencia ligada define valores de referencia a localizaciones del modo referenciado especificado.

Si el modo referenciado es un modo no moreta M, el modo referencia ligada define valores de referencia a localizaciones de M.

Si el modo referenciado es un modo moreta MM, el modo referencia ligada define valores de referencia a localizaciones de MM o cualquier sucesor de MM.

propiedades estáticas: Un modo referencia ligada tiene la siguiente propiedad hereditaria:

- Un modo **referenciado** que es el *modo referenciado*.

ejemplo:

10.42 **REF** *cell* *(1.1)*

3.7.3 Modos referencia libre

sintaxis:

<modo referencia libre> ::= *(1)*
<nombre de modo referencia libre> *(1.1)*

nombres predefinidos: El nombre *PTR* está predefinido como un nombre de **modo referencia libre**.

semántica: Un modo referencia libre define valores de referencia a localizaciones de cualquier modo estático.

ejemplo:

19.8 *PTR* *(1.1)*

3.7.4 Modos descriptor

sintaxis:

<modo descriptor> ::= *(1)*
ROW *<modo cadena>* *(1.1)*
| **ROW** *<modo matriz>* *(1.2)*
| **ROW** *<modo estructura variable>* *(1.3)*
| *<nombre de modo descriptor>* *(1.4)*

semántica: Un modo descriptor define valores de referencia relativos a localizaciones de un modo dinámico (que son localizaciones de algún modo parametrizado con parámetros no **constantes**).

Un valor descriptor puede referirse a:

- localizaciones cadena con **longitud de cadena no constante**;
- localizaciones matriz con un **límite superior no constante**;
- localizaciones estructura parametrizada con parámetros no **constantes**.

propiedades estáticas: Un modo descriptor tiene la siguiente propiedad hereditaria:

- Un modo **origen referenciado** que es el *modo cadena*, el *modo matriz*, o el *modo estructura variable*, respectivamente.

condición estática: El *modo estructura variable* debe ser **parametrizable**.

ejemplo:

8.6 **ROW CHARS** *(max)* *(1.1)*

3.8 Modos procedimiento

sintaxis:

<modo procedimiento> ::= *(1)*
PROC ([*<lista de parámetros>*]) [*<espec de resultado>*]
[**EXCEPTIONS** (*<lista de excepciones>*)] *(1.1)*
| *<nombre de modo procedimiento>* *(1.2)*

<lista de parámetros> ::= *(2)*
<espec de parámetro> { , *<espec de parámetro>* }* *(2.1)*

<espec de parámetro> ::= *(3)*
<modo> [*<atributo de parámetro>*] *(3.1)*

<atributo de parámetro> ::= *(4)*
IN | **OUT** | **INOUT** | **LOC** [**DYNAMIC**] *(4.1)*

<espec de resultado> ::= *(5)*
RETURNS (*<modo>* [*<atributo resultado>*]) *(5.1)*

<atributo de resultado> ::= *(6)*
[**NONREF**] **LOC** [**DYNAMIC**] *(6.1)*

<lista de excepciones> ::= *(7)*
<nombre de excepción> { , *<nombre de excepción>* }* *(7.1)*

semántica: Un modo procedimiento define valores de procedimiento (**general**), es decir, los objetos designados por nombres de **procedimiento general**, que son nombres definidos en sentencias de definición de procedimiento. Los valores de procedimiento indican piezas de código en un contexto dinámico. Los modos procedimiento permiten manipular dinámicamente un procedimiento, por ejemplo, pasarlo en forma de parámetro a otros procedimientos, enviarlo en forma de valor de mensaje a un tampón, almacenarlo en una localización, etc.

Los valores de procedimiento pueden ser llamados (véase 6.7).

Dos valores de procedimiento son iguales si y sólo si denotan el mismo procedimiento en el mismo contexto dinámico, o si ninguno denota procedimiento alguno (es decir, son el valor *NULL*).

propiedades estáticas: Un modo procedimiento tiene las siguientes propiedades hereditarias:

- Una lista de **especs de parámetro**, cada una de las cuales compuesta por un modo y posiblemente un atributo de parámetro. Las **especs de parámetro** se definen mediante la *lista de parámetros*.
- Una **spec de resultado** facultativa, compuesta por un modo y un atributo de resultado facultativo. La **spec de resultado** se define por la *espec de resultado*.
- Una lista posiblemente vacía de nombres de **excepción**, que son los mencionados en la *lista de excepciones*.

condiciones estáticas: Todos los nombres mencionados en la *lista de excepciones*, deben ser diferentes.

Sólo si se especifica **LOC** en la *espec de parámetro* o en la *espec de resultado*, el *modo* en ella puede tener la **propiedad de no-valor**.

Si se especifica **DYNAMIC** en la *espec de parámetro* o en la *espec de resultado*, el *modo* en ella debe ser **parametrizable**.

3.9 Modos ejemplar

sintaxis:

<modo ejemplar> ::= *(1)*
<nombre de modo instancia> *(1.1)*

nombres predefinidos: El nombre *INSTANCE* está predefinido como un nombre de **modo ejemplar**.

semántica: Un modo ejemplar define valores que identifican procesos. La creación de un nuevo proceso (véanse 5.2.15, 6.13 y 11.1) proporciona un valor de ejemplar único como identificación del proceso creado.

Dos valores de ejemplar son iguales si y sólo si identifican el mismo proceso o no identifican ningún proceso (es decir, son el valor *NULL*).

ejemplo:

15.39 INSTANCE *(1.1)*

3.10 Modos sincronización

3.10.1 Generalidades

sintaxis:

<modo sincronización> ::= *(1)*
<modo evento> *(1.1)*
| *<modo tampón>* *(1.2)*

semántica: Un modo sincronización proporciona un medio para la sincronización y la comunicación entre procesos (véase la cláusula 11). No existe en CHILL ninguna expresión que designe un valor definido por un modo sincronización. En consecuencia, no hay operaciones definidas sobre los valores.

3.10.2 Modos evento

sintaxis:

<modo evento> ::= *(1)*
EVENT [(*<longitud de evento>*)] *(1.1)*
| *<nombre de modo evento>* *(1.2)*

<longitud de evento> ::= *(2)*
<expresión literal entera> *(2.1)*

semántica: Una localización de modo suceso proporciona un medio para la sincronización entre procesos. Las operaciones definidas en las localizaciones de modo evento son la acción continuar, la acción demorar y la acción demorar y elegir, descritas en 6.15, 6.16 y 6.17 respectivamente.

La *longitud de evento* especifica el máximo número de procesos que pueden resultar demorados en una localización evento; ese número no está limitado si no se especifica *longitud de evento*.

Una localización modo evento que contiene el valor **indefinido** es un evento "vacío", es decir, no tiene asociados procesos demorados.

propiedades estáticas: Un modo evento tiene la siguiente propiedad hereditaria:

- Una **longitud de evento** facultativa, que es el valor entregado por *longitud de evento*.

condiciones estáticas: La *longitud de evento* debe entregar un valor positivo.

La evaluación de la *longitud de evento* no debe depender directa ni indirectamente del valor de la **longitud de evento** del modo evento.

ejemplo:

14.10 **EVENT** *(1.1)*

3.10.3 Modos tampón

sintaxis:

<modo tampón> ::= *(1)*
BUFFER [(*<longitud de tampón>*)] *<modo elemento tampón>* *(1.1)*
| *<nombre de modo tampón>* *(1.2)*

<longitud de tampón> ::= *(2)*
<expresión literal entera> *(2.1)*

<modo elemento tampón> ::= *(3)*
<modo> *(3.1)*

semántica: Una localización de modo tampón proporciona un medio para la sincronización y comunicación entre procesos. Las operaciones definidas sobre localizaciones tampón son la acción enviar, la acción recibir y elegir y la expresión recibir, descritas en 6.18 y 6.19, respectivamente.

La *longitud de tampón* especifica el máximo número de valores que pueden almacenarse en una localización suceso; ese número no está limitado si no se especifica *longitud de tampón*.

Una localización modo tampón que contiene el valor **indefinido** es un evento "vacío", es decir, no tiene asociados procesos demorados, ni tampoco hay mensajes en el tampón.

propiedades estáticas: Un modo tampón tiene las siguientes propiedades hereditarias:

- Una **longitud de tampón** facultativa, que es el valor entregado por *longitud de tampón*.
- Un modo **elemento tampón**, que es el *modo elemento tampón*.

condiciones estáticas: La *longitud de tampón* debe entregar un valor no negativo.

El *modo elemento tampón* no debe tener la **propiedad de no-valor**.

La evaluación de la *longitud de tampón* no debe depender directa ni indirectamente del valor de la **longitud de tampón** del modo tampón.

ejemplos:

16.30 **BUFFER** *(1) user_messages* *(1.1)*

16.34 *user_buffers* *(1.2)*

3.11 Modos entrada-salida

3.11.1 Generalidades

sintaxis:

<modo entrada-salida> ::= *(1)*
<modo asociación> *(1.1)*
| *<modo acceso>* *(1.2)*
| *<modo texto>* *(1.3)*

semántica: Un modo de entrada-salida proporciona un medio para las operaciones de entrada-salida definidas en la cláusula 7. No existe en CHILL una expresión que denote un valor definido por un modo entrada-salida. En consecuencia, no hay operaciones definidas sobre los valores.

ejemplo:

20.17 *ASSOCIATION* *(1.1)*

3.11.2 Modos asociación

sintaxis:

<modo asociación> ::= *(1)*
<nombre de modo asociación> *(1.1)*

nombres predefinidos: El nombre *ASSOCIATION* está predefinido como un nombre **de modo asociación**.

semántica: Las localizaciones de modo asociación proporcionan un medio para representar una relación con un objeto del mundo exterior. Esta relación se denomina asociación en CHILL; pueden crearse asociaciones por la rutina incorporada *ASSOCIATE* y terminarlas por *DISSOCIATE*.

Una localización modo asociación que contiene el valor **indefinido** está "vacía", es decir, no contiene una asociación.

3.11.3 Modos acceso

sintaxis:

<modo acceso> ::= *(1)*
ACCESS [(*<modo índice>*)] [*<modo registro>* [***DYNAMIC***]] *(1.1)*
| *<nombre de modo acceso>* *(1.2)*

<modo registro> ::= *(2)*
<modo> *(2.1)*

<modo índice> ::= *(3)*
<modo discreto> *(3.1)*
| *<intervalo de literal>* *(3.2)*

sintaxis derivada: La notación de modo índice *intervalo de literal* se deriva del modo discreto **RANGE** (*intervalo de literal*).

semántica: Una localización de modo acceso proporciona un medio para posicionar un fichero y para transferir valores de un programa CHILL a un fichero del mundo exterior, y viceversa.

Un modo acceso puede definir un *modo registro*; este modo registro define el modo **raíz** de la clase de los valores que pueden transferirse vía una localización de ese modo acceso a un fichero, o desde un fichero. El modo del valor transferido puede ser dinámico, es decir, el **tamaño** del registro puede variar cuando se especifica el atributo **DYNAMIC** en la denotación de modo acceso, o cuando *modo registro* es un modo cadena **variable**. En este último caso no es necesario especificar **DYNAMIC**.

Un modo acceso puede definir también un *modo índice*; dicho modo índice define el tamaño de una "ventana" al fichero (o parte del mismo), a partir de la cual es posible leer (o escribir) registros a voluntad. Esta ventana puede posicionarse en un fichero (**indexable**) por la operación conectar. Si no se especifica ningún *modo índice*, los registros sólo podrán transferirse secuencialmente.

Una localización modo acceso que contiene el valor **indefinido** está "vacía", es decir, no contiene una asociación.

propiedades estáticas: Un modo acceso tiene las siguientes propiedades hereditarias:

- Un modo **registro** facultativo, que es el *modo registro* si está presente. Es un modo **registro dinámico** si se especifica **DYNAMIC** o si el *modo registro* es un modo cadena **variable**; en otro caso, es un modo **registro estático**.
- Un modo **índice** facultativo, que es el *modo índice*.
- **Límite superior** y **límite inferior** facultativos que son el **límite superior** y el **límite inferior** del *modo índice*, está presente.

condiciones estáticas: El *modo registro* facultativo debe tener la **propiedad de no-valor**.

Si se especifica **DYNAMIC**, el modo **registro** debe ser **parametrizable**, y no debe ser un modo estructura **sin marcador**.

El *modo índice* no debe ser un modo conjunto **numerado** ni un modo intervalo discreto **numerado**.

Si el *modo índice* es un *intervalo de literal* de la forma:

<límite inferior> : <límite superior>

la evaluación del 1.*límite inferior,* 2.*límite superior*, no podrá depender directa ni indirectamente del valor del 1.**límite inferior**, 2.**límite superior** del modo acceso.

ejemplos:

20.18 **ACCESS** *(index_set) record_type* *(1.1)*

22.20 **ACCESS** *string* **DYNAMIC** *(1.1)*

20.18 *record_type* *(2.1)*

20.18 *index_set* *(3.1)*

3.11.4 Modos texto

sintaxis:

<modo texto> ::= *(1)*
<modo texto estrecho> *(1.1)*
| *<modo texto ancho>* *(1.2)*

<modo texto estrecho> ::= *(2)*
TEXT (*<longitud de texto>*) [*<modo índice>*] [**DYNAMIC**] *(2.1)*

<modo texto ancho> ::= *(3)*
WTEXT (*<longitud de texto>*) [*<modo índice>*] [**DYNAMIC**] *(3.1)*

<longitud de texto> ::= *(4)*
<expresión literal entera> *(4.1)*

semántica: Una localización de modo texto proporciona un medio para transferir valores representados en forma legible para el ser humano, desde un programa CHILL a un fichero en el mundo exterior, y viceversa. Una localización de modo texto tiene un **registro de texto** y una sublocalización **acceso**. La sublocalización **registro de texto** está inicializada con una cadena vacía.

Un modo texto tiene una **longitud de texto**, que define la longitud máxima de los registros que pueden ser transferidos, y posiblemente un modo **índice**, que tiene el mismo significado que para los modos acceso. El atributo **longitud efectiva** de una localización modo texto es la **longitud efectiva** de su **registro de texto**.

Una localización modo texto que contiene el valor **indefinido** tiene una sublocalización **registro de texto** que contiene la cadena vacía y una sublocalización acceso que contiene el valor **indefinido**.

propiedades estáticas: Un modo texto tiene las siguientes propiedades hereditarias:

- Una **longitud de texto**, que es el valor entregado por *longitud de texto*.
- Un modo **registro de texto**, que es **CHARS** (<*longitud de texto*>) **VARYING** en caso de **TEXT** y que es **WCHARS** (<*longitud de texto*>) **VARING** en caso de **WTEXT**.
- Un modo **acceso**, que es **ACCESS** [(<*modo índice*>)] **CHARS** (<*longitud de texto*>). [**DYNAMIC**] en caso de **TEXT** y que es **WCHARS** (<*longitud de texto*>) [**DYNAMIC**] en caso de **WTEXT** (<*modo índice*> y **DYNAMIC** son parte del modo únicamente si están especificados).
- **Límite superior** y **límite inferior** facultativos que son el **límite superior** y **límite inferior** del *modo índice*, si está presente.

condiciones estáticas: Si el *modo índice* es un *intervalo de literal* de la forma:

<*límite inferior*> : <*límite superior*>

la evaluación del 1.*límite inferior,* 2.*límite superior*, no podrá depender directa ni indirectamente del valor del 1.**límite inferior**, 2.**límite superior** del modo texto.

ejemplo:

26.8 **TEXT** *(80)* **DYNAMIC** *(2.1)*

3.12 Modos temporización

3.12.1 Generalidades

sintaxis:

<*modo temporización*> ::= *(1)*
<*modo duración*> *(1.1)*
| <*modo tiempo absoluto*> *(1.2)*

semántica: Un modo de temporización proporciona un medio para la supervisión de tiempo de los procesos descritos en la cláusula 9. Los valores de temporización son creados por un conjunto de rutinas incorporadas. Los operadores relacionales se definen sobre valores de temporización.

3.12.2 Modos duración

sintaxis:

<*modo duración*> ::= *(1)*
<*nombre de modo duración*> *(1.1)*

nombres predefinidos: El nombre *DURATION* está predefinido como un nombre **de modo duración**.

semántica: Un modo duración define valores que representan periodos de tiempo. El conjunto de valores definido por el modo duración está definido por la implementación. Una implementación puede optar por representar los valores de duración como pares de precisión y valor. Los valores de duración se ordenan de manera intuitiva.

3.12.3 Modos tiempo absoluto

sintaxis:

<*modo tiempo absoluto*> ::= *(1)*
<*nombre de modo tiempo absoluto*> *(1.1)*

nombres predefinidos: El nombre *TIME* está predefinido como un nombre **de modo tiempo absoluto**.

semántica: Un modo tiempo absoluto define valores que representan puntos de tiempo. El conjunto de valores definido por el modo tiempo absoluto está definido por la implementación. Los valores de tiempo absoluto se ordenan de manera intuitiva.

3.13 Modos compuestos

3.13.1 Generalidades

sintaxis:

<modo compuesto> ::= *(1)*
<modo cadena> *(1.1)*
| *<modo matriz>* *(1.2)*
| *<modo estructura>* *(1.3)*
| *<modo moreta>* *(1.4)*

semántica:Un modo compuesto define valores compuestos, es decir, valores que constan de subcomponentes que pueden ser accedidos u obtenidos (véanse 4.2.6-4.2.10 y 5.2.6-5.2.10).

3.13.2 Modos cadena

sintaxis:

<modo cadena> ::= *(1)*
<tipo de cadena> (*<longitud de cadena>*) [**VARYING**] *(1.1)*
| *<modo cadena parametrizado>* *(1.2)*
| *<nombre de modo cadena>* *(1.3)*

<modo cadena parametrizado> ::= *(2)*
<nombre de modo cadena origen> (*<longitud de cadena>*) *(2.1)*
| *<nombre de modo cadena parametrizado>* *(2.2)*

<nombre de modo cadena origen> ::= *(3)*
<nombre de modo cadena> *(3.1)*

<tipo de cadena> ::= *(4)*
BOOLS *(4.1)*
| **CHARS** *(4.2)*
| **WCHARS** *(4.3)*

<longitud de cadena> ::= *(5)*
<expresión literal entera> *(5.1)*

semántica: Un modo cadena **fijo** define valores de cadena de bits o de caracteres de una longitud indicada o implicada por el modo cadena. Un modo cadena **variable** define valores de cadena de bits o de caracteres cuya **longitud efectiva** puede variar dinámicamente de 0 a la **longitud de cadena**. La longitud se conoce sólo durante la ejecución, a partir del valor del atributo **longitud efectiva**. Para un modo cadena **fijo** la **longitud efectiva** es siempre igual a la **longitud de cadena**. Las cadenas de caracteres son secuencias de valores de carácter; las cadenas de bits son secuencias de valores booleanos.

Los valores de cadena están vacíos o tienen elementos de cadena numerados de 0 en adelante.

Los valores de cadena de un modo cadena dado están bien ordenados según el orden de los valores componentes y la siguiente definición.

Dos cadenas s y t son iguales si y sólo si tienen la misma longitud l y $s(i) = t(i)$ para toda $0 \le i < l$. Una cadena s precede a t cuando:

- existe un índice j tal que $s(j) < t(j)$ y $s(0 : j - 1) = t(0 : j - 1)$, o
- *LENGTH (s) < LENGTH (t)* y $s = t(0$ **UP** *LENGTH (s))*.

El operador de concatenación está definido sobre valores de cadena. Los operadores lógicos usuales están definidos sobre valores de cadena de bits y operan entre sus elementos correspondientes (véase 5.3).

La longitud máxima de modos cadena está definida por la implementación.

propiedades estáticas: Un modo cadena tiene las siguientes propiedades hereditarias:

- Una **longitud de cadena**, que es el valor entregado por longitud de cadena.
- Un **límite superior** y un **límite inferior**, que son los valores entregados por longitud de cadena – 1 y 0, respectivamente.

- Un modo **elemento** que es *M* o **READ** *M*, donde *M* es *BOOL*, *CHAR* o *WCHAR* según que *tipo de cadena* especifique **BOOLS**, **CHARS** o **WCHARS**, o el modo **elemento** del *nombre de modo cadena origen*, respectivamente. El modo **elemento** será **READ** *M* si y sólo si el modo cadena es un modo **lectura solamente** implícito.
- Es un modo cadena **variable** si se especifica **VARYING**, o si el *nombre de modo cadena origen* es un modo cadena **variable**; en otro caso es un modo cadena **fijo**.

Un modo cadena es **parametrizado** si y sólo si es un *modo cadena parametrizado*.

Un modo cadena **parametrizado** tiene un modo cadena **origen** que es el modo denotado por *nombre de modo cadena origen*.

Un modo cadena **variable** tiene las siguientes propiedades no hereditarias: tiene un modo **componente** definido como sigue:

- Si el modo cadena **variable** es de la forma:

  <*tipo de cadena*> (<*longitud de cadena*>) **VARYING**

  entonces es <*tipo de cadena*> (<*longitud de cadena*>).
- Si el modo cadena **variable** es de la forma:

  <*nombre de modo cadena origen*> (<*longitud de cadena*>)

  entonces el modo **componente** es *&nombre* (*longitud de cadena*), donde *&nombre* es un nombre de **sínmodo** introducido virtualmente, **sinónimo** del modo **componente** del *nombre de modo cadena origen*.
- Si el modo cadena **variable** es un *nombre de modo cadena* que es un nombre de **sínmodo**, entonces su modo **componente** es el del modo **definidor** del nombre de **sínmodo**; en otro caso es un nombre de **neomodo**, y entonces su modo **componente** es un modo **componente** introducido virtualmente (véase 3.2.3).

condiciones estáticas: La *longitud de cadena* debe entregar un valor no negativo.

El valor entregado por la *longitud de cadena* contenida directamente en un *modo cadena parametrizado* debe ser menor o igual que la **longitud de cadena** del *nombre de modo cadena origen*. Esta condición sólo se aplica a los *modos cadena* **parametrizados** que no estén introducidos virtualmente.

La evaluación de la *longitud de cadena* no debe depender directa ni indirectamente del valor de la **longitud de cadena** del modo cadena.

ejemplos:

7.51 **CHARS** *(20)* *(1.1)*

22.22 **CHARS** *(20)* **VARYING** *(1.1)*

3.13.3 Modos matriz

sintaxis:

<*modo matriz*> *::=* *(1)*
    ARRAY (<*modo índice*> { , <*modo índice*> }*)
    <*modo elemento*> { <*organización de elementos*> }* *(1.1)*
  | <*modo matriz parametrizado*> *(1.2)*
  | <*nombre de modo matriz*> *(1.3)*

<*modo matriz parametrizado*> *::=* *(2)*
    <*nombre de modo matriz origen*> (<*índice superior*>) *(2.1)*
  | <*nombre de modo matriz parametrizado*> *(2.2)*

<*nombre de modo matriz origen*> *::=* *(3)*
    <*nombre de modo matriz*> *(3.1)*

<*índice superior*> *::=* *(4)*
    <*expresión literal discreta*> *(4.1)*

<*modo elemento*> *::=* *(5)*
    <*modo*> *(5.1)*

sintaxis derivada: Un *modo matriz* que contenga más de un modo índice (lo que denota una matriz multidimensional) es la sintaxis derivada para un *modo matriz* que tiene un *modo elemento*, que a su vez es un *modo matriz*. Por ejemplo:

ARRAY (*1:20,1:10*) *INT*

se deriva de:

ARRAY (**RANGE** *(1:20)*) **ARRAY** (**RANGE** *(1:10)*) *INT*

La ocurrencia de más de una *organización de elementos* está permitida sólo si se utiliza esta sintaxis derivada. El número de ocurrencias de *organización de elementos* será menor o igual que el número de ocurrencias del *modo índice*. En este caso, la *organización de elementos* situada más a la izquierda se asocia con el *modo elemento* más interno, etc.

semántica: Un modo matriz define valores compuestos, que son listas de valores definidos por su modo elemento. Mediante la especificación de la *organización de elementos* puede controlarse la organización física de una localización o un valor matricial (véase 3.13.5). Dos valores matrices son iguales si y sólo si tienen el mismo **número de elementos** y los valores de los elementos correspondientes son iguales.

El **número de elementos** máximo de modos matriz se define por la implementación.

propiedades estáticas: Un modo matriz tiene las siguientes propiedades hereditarias:

- Un modo **índice**, que es el *modo índice* si no es un *modo matriz parametrizado*; en otro caso, el modo **índice** es el modo intervalo discreto construido como sigue:

  &nombre (*límite inferior* : *límite superior*)

  donde *&nombre* es un nombre virtual de **sínmodo**, **sinónimo** del modo **índice** del *nombre de modo matriz origen, límite inferior* es el límite inferior del modo **índice** del *nombre de modo matriz origen*, y *límite superior* es el *índice superior*.

- Un **límite superior** y un **límite inferior**, que son el **límite superior** y el **límite inferior** respectivamente de su modo **índice**.

- Un modo **elemento** que es *M* o **READ** *M*, donde *M* es el *modo elemento*, o el modo **elemento** del *nombre de modo matriz origen*, respectivamente. El modo **elemento** será **READ** *M* si y sólo si *M* no es un modo de **lectura solamente** y el *modo matriz* es un modo de **lectura solamente**. El modo **elemento** es un modo de **lectura solamente** implícito si es **READ** *M*.

- Una **organización de elementos**, que, si se trata de un *modo matriz parametrizado*, es la **organización de elementos** de su *nombre de modo matriz origen*; en otro caso, es la *organización de elementos* especificada, o la establecida por defecto en la implementación, que es **PACK** o **NOPACK**.

- Un **número de elementos**, que es el valor entregado por:

  NUM (*límite superior*) – *NUM* (*límite inferior*) + 1

  donde *límite superior* y *límite inferior* son respectivamente el **límite superior** y el **límite inferior** de su modo **índice**.

- Es un modo **con correspondencia** si la *organización de elementos* se especifica y es un *paso*.

Un modo matriz es **parametrizado** si y sólo si es un *modo matriz parametrizado*.

Un modo matriz **parametrizado** tiene un modo matriz **origen** que es el modo denotado por *nombre de modo matriz origen*.

condiciones estáticas: La clase de *índice superior* debe ser **compatible** con el modo **índice** del *nombre de modo matriz origen* y el valor entregado por éste debe estar en el intervalo definido por ese modo **índice**.

Si el modo matriz es un *modo matriz parametrizado*, la evaluación del *índice superior* no debe depender directa ni indirectamente del valor del **límite superior** del modo matriz. Si el modo matriz no es ni un *modo matriz parametrizado* ni un *nombre de modo matriz*, y si el *modo matriz* es un *intervalo de literal* de la forma:

<*límite inferior*> : <*límite superior*>

la evaluación del 1.*límite inferior*, 2.*límite superior*, no debe depender directa ni indirectamente del valor del 1.**límite inferior**, 2.**límite superior**, del modo matriz.

ejemplos:

5.27 **ARRAY** *(1:16)* **STRUCT** *(c4, c2, c1 BOOL)* *(1.1)*

11.12 **ARRAY** *(line)* **ARRAY** *(column) square* *(1.1)*

11.17 *board* *(1.3)*

3.13.4 Modos estructura

sintaxis:

<modo estructura> ::= *(1)*
STRUCT (*<campo>* { , *<campo>* }*) *(1.1)*
| *<modo estructura parametrizada>* *(1.2)*
| *<nombre de modo estructura>* *(1.3)*

<campo> ::= *(2)*
<campo fijo> *(2.1)*
| *<campo alternativo>* *(2.2)*

<campo fijo> ::= *(3)*
<lista de ocurrencias de definición de nombre de campo> <modo>
[*<organización de campo>*] *(3.1)*

<campo alternativo> ::= *(4)*
CASE [*<lista de marcadores>*] **OF**
<alternativa variable> { , *<alternativa variable>* }*
[**ELSE** [*<campo variable>* { , *<campo variable>* }*]] **ESAC** *(4.1)*

<alternativa variable> ::= *(5)*
[*<especificación de etiqueta de caso>*]:
[*<campo variable>* { , *<campo variable>* }*] *(5.1)*

<lista de marcadores> ::= *(6)*
<nombre de campo marcador> { , *<nombre de campo marcador>* }* *(6.1)*

<campo variable> ::= *(7)*
<lista de ocurrencia de definición de nombre de campo> <modo>
[*<organización de campo>*] *(7.1)*

<modo estructura parametrizada> ::= *(8)*
<nombre de modo estructura variable origen> (*<lista de expresiones literales>*) *(8.1)*
| *<nombre de modo estructura parametrizada>* *(8.2)*

<nombre de modo estructura variable origen> ::= *(9)*
<nombre de modo estructura variable> *(9.1)*

<lista de expresiones literales> ::= *(10)*
<expresión literal discreta> { , *<expresión literal discreta>* }* *(10.1)*

sintaxis derivada: Una ocurrencia de *campo fijo*, o una ocurrencia de *campo variable*, cuando la *lista de ocurrencias de definición de nombre de campo* contenga más de una *ocurrencia de definición de nombre de campo*, es sintaxis derivada para varias ocurrencias de *campo fijo* o para varias ocurrencias de *campo variable* con una *ocurrencia de definición de nombre de campo*, respectivamente, cada uno con el *modo* especificado y una *organización de campo* facultativa. En el caso de *organización de campo*, esta *organización* no debe ser *pos*. Por ejemplo:

STRUCT (*I,J BOOL* **PACK**)

se deriva de:

STRUCT (*I BOOL* **PACK**, *J BOOL* **PACK**)

semántica: Los modos estructura definen valores compuestos consistentes en una lista de valores, seleccionables por un nombre de componente. Cada valor se define mediante un modo asociado al nombre del componente. Los valores de estructura pueden residir en localizaciones estructura (compuesta), en las que el nombre del componente sirve de acceso a la sublocalización. Los componentes de un valor o localización estructura se denominan campos, y sus nombres, nombres de **campo**.

Hay estructuras **fijas**, estructuras **variables** y estructuras **parametrizadas**.

Las estructuras **fijas** constan solamente de campos fijos, es decir, campos que están siempre presentes y que son accesibles sin ninguna comprobación dinámica.

Las estructuras **variables** tienen campos variables, es decir, campos que no están siempre presentes. Para las estructuras **variables marcadas**, la presencia de estos campos solamente es conocida en el momento de la ejecución por medio de uno o varios valores de ciertos campos fijos asociados denominados campos **marcadores**. Las estructuras **variables sin marcadores** no tienen campos **marcadores**. Puesto que la composición de una estructura **variable** puede cambiar durante la ejecución, la dimensión de la localización correspondiente a una estructura variable se basa la mayor parte del conjunto (caso más desfavorable) de alternativas variables.

En un *campo de alternativa*, la *alternativa variable* elegida es aquella para la cual concuerdan los valores dados en la especificación de la etiqueta del caso; si ningún valor concuerda, se elige la *alternativa variable* que sigue a **ELSE** (que estará presente).

Una estructura **parametrizada** se determina a partir de un modo estructura **variable** para el cual se ha especificado estáticamente la elección de alternativas variables, mediante expresiones **literales**. La composición es fija desde el punto de creación de la estructura parametrizada y no puede cambiar durante la fase de ejecución. Los campos **marcadores**, si existen, son de **lectura solamente** y se inicializan automáticamente con los valores especificados. Para una localización estructura parametrizada puede asignarse una cuantía precisa de almacenamiento en el punto de declaración o generación. Obsérvese que también existen modos estructura **parametrizada** dinámicos. En 3.14.4 se define su semántica.

Puede controlarse la organización de un valor o de una localización de estructura mediante una especificación de organización de campo (véase 3.13.5).

Dos valores estructura son iguales si y sólo si lo son sus valores componentes correspondientes. Sin embargo, si los valores de estructura son valores de estructura **variables sin marcador**, el resultado de la comparación se define en la implementación.

En el caso de un modo con la **propiedad parametrizada con marcador**, el valor **indefinido** denota un valor en el que los subvalores de campo marcadores son iguales a los correspondientes valores de parámetros y todos los demás son iguales al valor **indefinido**.

propiedades estáticas:

generalidades: Un modo estructura tiene las siguientes propiedades hereditarias:

- Es un modo estructura **fijo** si es un *modo estructura* que no contiene directamente una ocurrencia de *campo alternativo*.
- Es un modo estructura **variable** si es un *modo estructura* y contiene al menos una ocurrencia de *campo alternativo*.
- Es un modo estructura **parametrizada** si es un *modo estructura parametrizada*.
- Tiene un conjunto de nombres de **campo**. Más adelante se define este conjunto para los distintos casos. Se dice que un nombre es un nombre de **campo** si y sólo si está definido en una *lista de ocurrencias de definición de nombre de campo* en *campos fijos* o en *campos variables* en un *modo estructura*.

  Cada *campo fijo*, *campo variable*, y por tanto cada nombre de **campo** de un modo estructura, tiene asociado un modo **campo** que es *M* o **READ** *M*, siendo *M* el *modo* en el *campo fijo* o *campo variable*. El modo **campo** es **READ** *M* si *M* no es un modo de **lectura solamente**, bien el modo estructura es un modo de **lectura solamente** o bien, el campo es un campo **marcador** de un modo estructura **parametrizada**. El modo **campo** es un modo de **lectura solamente** implícito, si es **READ** *M*.

  Un *campo fijo*, *campo variable*, y por tanto un nombre de **campo** de un modo estructura dado, tiene asociada una **organización de campo**, que es la *organización de campo*, en el *campo fijo* o *campo variable*, si está presente; en otro caso, es la organización de campo por defecto, que es **PACK** o **NOPACK**.
- Es un modo **con correspondencia** si sus nombres de **campo** tienen una *organización de campo* que es *pos*.

estructuras fijas: Un modo estructura **fija** tiene la siguiente propiedad hereditaria:

- Un conjunto de nombres de **campo**, que es el conjunto de nombres definidos por cualquier *lista de ocurrencias de definición de nombre de campo* en *campos fijos*. Estos nombres de **campo** son nombres de **campo fijo**.

estructuras variables: Un modo estructura **variable** tiene las siguientes propiedades hereditarias:

- Un conjunto de nombres de **campo**, que es la unión del conjunto de nombres definidos por cualquier lista de ocurrencias de definición de nombre de campo en campos fijos con el conjunto de nombres definido por cualquier lista de ocurrencias de definición de nombre de campo en campos alternativos. Los nombres de **campo** definidos por una lista de ocurrencias de definición de nombre de campo en campos fijos son los nombres de **campo fijo** del modo estructura **variable**; sus otros nombres de **campo** son los nombres de **campo variable**.
- Un nombre de **campo** de un modo estructura **variable** es un nombre de **campo marcador** si y sólo si aparece en cualquier lista de marcadores de un campo alternativo. Los campos alternativos que no tengan especificados marcadores son *campos alternativos* **sin marcadores**.
- Un modo estructura **variable** es un modo estructura **variable sin marcadores** si todas las ocurrencias de sus *campos alternativos* son **sin marcador**. En otro caso, es un modo estructura **variable con marcadores**.

- Un modo estructura **variable** es un modo estructura **variable parametrizable** si es un modo estructura **variable con marcadores** o un modo estructura **variable sin marcadores** en el que para cada una de las ocurrencias de *campo alternativo* se da una *especificación de etiqueta de caso* para todas las ocurrencias de *alternativa variable* de la misma.
- Un modo estructura **variable parametrizable** tiene asociada una lista de clases, determinada como sigue:
  - si se trata de un modo estructura **variable con marcadores**, la lista de clases M_i – valuada, donde M_i son los modos de los nombres de **campo marcador** en el orden en que se definen en *campos fijos*;
  - si se trata de un modo estructura **variable sin marcadores**, la lista está formada por las distintas **listas resultantes de clases** de cada uno de los *campos alternativos*, concatenados en el orden en que ocurren estos *campos alternativos*. La **lista resultante de clases** de una ocurrencia de *campo alternativo*, es la **lista resultante de clases** de la lista de ocurrencias de *especificación de la etiqueta de caso* de la misma (véase 12.3).

estructuras parametrizadas: Un modo estructura **parametrizada** tiene las siguientes propiedades hereditarias:

- Un modo estructura **variable origen**, que es el modo denotado por *nombre de modo estructura variable origen*.
- Un conjunto de nombres de **campo**, que es la unión del conjunto de nombres de **campo fijo** de su modo estructura **variable origen** con el conjunto de aquellos nombres de **campo variable** de su modo estructura **variable origen** definidos en las ocurrencias de *alternativa variable* seleccionadas por la lista de valores, definida por la *lista de expresiones literales*.
- El conjunto de nombres de **campo marcador** de un *modo estructura parametrizada* es el conjunto de nombres de **campo marcador** de su modo estructura **variable origen**.
- Una lista de valores asociados, definida por la *lista de expresiones literales*.
- Es un modo estructura **parametrizada con marcadores** si su modo estructura **variable origen** es un modo estructura **variable con marcadores**; en otro caso, el modo estructura **parametrizada** es **sin marcadores**.

Para los modos de estructura **parametrizada** dinámicos, véase 3.14.4.

condiciones estáticas:

generalidades: Todos los nombres de **campo** de un modo estructura deben ser diferentes.

Si un campo cualquiera tiene una organización que es *pos*, todos los campos deben tener una organización de campo que sea *pos*.

estructuras variables: Un nombre de **campo marcador** debe ser un nombre de **campo fijo** que debe estar definido textualmente con antelación a todas las ocurrencias de *campo alternativo* en cuya *lista de marcadores* se mencione. (En consecuencia, un campo **marcador** precede a todos los campos **variables** que dependen del mismo.) El modo de un nombre de **campo marcador** debe ser un modo discreto.

El *modo de un campo variable* no puede tener ni la **propiedad de no-valor** ni la **propiedad parametrizada con marcadores**.

En un modo estructura **variable**, las ocurrencias de *campo alternativo* deben ser o todas **con marcadores** o todas **sin marcadores**. En los campos alternativos con marcadores, puede especificarse la especificación de etiqueta de caso en cada alternativa variable. En los *campos alternativos* **sin marcadores**, puede omitirse la *especificación de etiqueta de caso* en todas las ocurrencias de *alternativa variable* juntas, o debe especificarse para todas las ocurrencias de *alternativa variable*.

Si se da la *especificación de etiqueta de caso*, para uno cualquiera de los *campos alternativos* correspondientes a un modo estructura **variable sin marcadores**, todos sus *campos alternativos* deben tener la *especificación de etiqueta de caso*.

Para los *campos alternativos*, deben cumplirse las condiciones de selección de caso (véase 12.3) y deben asegurarse los mismos requisitos de completud, coherencia y compatibilidad que para la acción de caso (véase 6.4). Cada uno de los nombres de **campo marcador** de la *lista de marcadores* (si existe) sirve como selector de caso con las clases M-valuadas, donde M es el modo del nombre del **campo marcador**. En el caso de campos alternativos **sin marcadores**, se ignoran las comprobaciones relativas al selector de caso.

Para un modo estructura **variable parametrizable**, ninguna de las clases de su lista de clases asociada puede ser la clase **general**. (Esta condición se cumple automáticamente con el modo estructura **variable con marcadores**.)

estructuras parametrizadas: El *nombre de modo estructura variable origen* debe ser **parametrizable**.

Debe haber tantas expresiones **literales** en la *lista de expresiones literales* como clases en la lista de clases del *nombre de modo estructura variable origen*. La clase de cada expresión **literal** debe ser **compatible** con la clase correspondiente (en posición) de la lista de clases. Si la última clase es una clase M-valuada, el valor entregado por la expresión **literal** debe ser uno de los valores definidos por M.

ejemplos:

3.3 **STRUCT** *(re, im INT)* *(1.1)*

11.7 **STRUCT** *(status* **SET** *(occupied, free),*
CASE *status* **OF**
(occupied): p piece,
(free):

ESAC*)* *(1.1)*

2.6 *fraction* *(1.3)*

11.7 *status* **SET** *(occupied, free)* *(3.1)*

11.8 *status* *(6.1)*

11.9 *p piece* *(7.1)*

3.13.5 Descripción de la organización de los modos matriz y los modos estructura

sintaxis:

<organización de elementos> ::= *(1)*
PACK | **NOPACK** | *<paso>* *(1.1)*

<organización de campo> ::= *(2)*
PACK | **NOPACK** | *<pos>* *(2.1)*

<paso> ::= *(3)*
STEP (*<pos>* [, *<tamaño de paso>*]) *(3.1)*

<pos> ::= *(4)*
POS (*<palabra>* , *<bit inicial>* , *<longitud>*) *(4.1)*
| **POS** (*<palabra>* [, *<bit inicial>* [: *<bit final>*]]) *(4.2)*

<palabra> ::= *(5)*
<expresión literal entera> *(5.1)*

<tamaño de paso> ::= *(6)*
<expresión literal entera> *(6.1)*

<bit inicial> ::= *(7)*
<expresión literal entera> *(7.1)*

<bit final> ::= *(8)*
<expresión literal entera> *(8.1)*

<longitud> ::= *(9)*
<expresión literal entera> *(9.1)*

semántica: Es posible controlar la organización de una matriz o una estructura proporcionando información de empaquetamiento o de correspondencia en su modo. La información de empaquetamiento puede ser **PACK** o **NOPACK**; la información de correspondencia es *paso* en el caso de modos matriz, o *pos* en el caso de campos de modos estructura. La ausencia de *organización de campo* u *organización de matriz* en un modo matriz o estructura se interpretará siempre como información de empaquetamiento, es decir, como **PACK** o **NOPACK**.

Si se especifica **PACK** para los elementos de una matriz o los campos de una estructura, esto significa que se optimiza la utilización de espacio de memoria para los elementos de matriz o los campos de estructura, mientras que **NOPACK** implica la optimización del tiempo de acceso a los elementos de la matriz o a los campos de la estructura. **NOPACK** implica también la **referenciabilidad**.

La información **PACK**, **NOPACK** se aplica sólo para un nivel, es decir, se aplica a los elementos de la matriz o a los campos de la estructura, no a los posibles componentes del elemento de la matriz o del campo de la estructura. La información de organización va siempre asociada al modo más próximo al cual puede aplicarse y que no tenga ya asociada una organización. Por ejemplo, si el empaquetamiento por defecto es **NOPACK**:

STRUCT (*f* **ARRAY** (*0:1*) *m* **PACK**)

es equivalente a:

STRUCT (*f* **ARRAY** (*0:1*) *m* **PACK NOPACK**)

Es también posible controlar la organización precisa de una matriz o una estructura especificando la información de posición para sus componentes en el modo. Esta información de posición se proporciona de las siguientes maneras:

- Para los modos matriz, la información de posición se facilita conjuntamente para todos los elementos, en forma de un *paso* que sigue al modo matriz.
- Para los modos estructura, la información de posición se proporciona individualmente para cada campo en forma de una *pos*, que sigue al modo del campo.

La información de correspondencia con *pos* se da en términos de desplazamientos de palabras y bits. Una *pos* de la forma:

POS (<*palabra*> , <*bit inicial*> , <*longitud*>)

define un desplazamiento de bits de

NUM (*palabra*) * WIDTH + *NUM* (*bit inicial*)

y una longitud de *NUM* (*longitud*) bits, donde *WIDTH* es el número (definido por la implementación) de bits en una palabra y, *palabra* es una *expresión literal entera*.

Cuando *pos* está especificada en *organización de campo*, define que el campo correspondiente comienza con el desplazamiento de bits indicado a partir del comienzo de cada localización de ese modo, y ocupa la longitud dada.

Un *paso* de la forma:

STEP (<*pos*> , <*tamaño de paso*>)

define una serie de desplazamientos de bits b_i, tomando i valores de 0 a $n - 1$ y siendo n el **número de elementos** en la matriz, y

$b_i = i * NUM$ (*tamaño de paso*)

El j-ésimo elemento de la matriz empieza con un desplazamiento de bits $p + b_j$ a partir del comienzo de cada localización del modo matriz, siendo p el desplazamiento de bits especificado en *pos*. Cada elemento ocupa la longitud dada en *pos*.

Notaciones por defecto

La notación:

POS (<*palabra*> , <*bit inicial*> : <*bit final*>)

es semánticamente equivalente a:

POS (<*palabra*> , <*bit inicial*> , *NUM* (*bit final*) – *NUM* (*bit inicial*) + 1)

La notación:

POS (<*palabra*> , <*bit inicial*>)

es semánticamente equivalente a:

POS (<*palabra*> , <*bit inicial*> , *BSIZE*)

donde *BSIZE* es el mínimo número de bits que debe ocupar el componente para la cual se ha especificado *pos.*

La notación:

POS (<*palabra*>)

es semánticamente equivalente a:

POS (<*palabra*> , *0* , *BSIZE*)

La notación:

STEP (<*pos*>)

es semánticamente equivalente a:

STEP (<*pos*> , *SSIZE*)

donde *SSIZE* es la <*longitud*> especificada en *pos* o deducible de *pos* mediante las reglas mencionadas.

propiedades estáticas: Para cualquier localización de un modo matriz, la organización de elementos del modo determina la referenciabilidad (por lenguaje), de sus sublocalizaciones (con inclusión de submatrices, segmentos de matrices) como sigue:

- todas las sublocalizaciones son **referenciables**, o ninguna lo es;
- si la organización de elementos es **NOPACK**, todas las sublocalizaciones son **referenciables**.

Para una localización de un modo estructura, la referenciabilidad del campo de estructura seleccionado por un nombre de **campo** se determina mediante la organización de campo del nombre de **campo** como sigue:

- el nombre de **campo** es **referenciable** si la organización de campo es **NOPACK**.

condiciones estáticas: Si el modo **elemento** de un modo matriz dado, o el modo **campo** de un nombre de **campo** de un modo estructura dado, es él mismo un modo matriz o estructura, debe ser un modo **con correspondencia** si el modo matriz o estructura dado es **con correspondencia**.

NUM (*palabra*), *NUM* (*bit inicial*), *NUM* (*bit final*), *NUM* (*longitud*) y *NUM* (*tamaño de paso*) ≥ 0;

NUM (*bit inicial*) y *NUM* (*bit final*) ≤ *WIDTH*; *NUM* (*bit inicial*) ≤ *NUM* (*bit final*).

Cada implementación define para cada modo un número mínimo de bits que sus valores necesitan ocupar; esto se denomina la ocupación mínima de bits. Para modos discretos, es cualquier número de bits no inferior al logaritmo de base 2 del **número de valores** del modo. Para modos matriz, es el desplazamiento del elemento del índice más elevado, más sus bits ocupados. Para modos estructura, es el desplazamiento del bit ocupado más elevado.

Para cada *pos*, la *longitud* especificada no será inferior a la mínima ocupación de bits del modo de los componentes de campo o matriz asociados.

Para cada modo matriz **con correspondencia**, el *tamaño de paso* no debe ser inferior a la *longitud* dada o implicada en la *pos*.

Consistencia y factibilidad

Consistencia: No podrá especificarse ningún componente de una matriz u objeto de estructura de modo que ocupe bits ocupados por otro componente del mismo objeto, salvo en el caso de dos nombres de **campo variable** definidos en la misma ocurrencia de *campo alternativo*. Sin embargo, en este último caso, los nombres de **campo variable**, no pueden ambos estar definidos en la misma *alternativa variable* ni ir ambos a continuación de **ELSE**.

Factibilidad: En el lenguaje no se definen requisitos de factibilidad, salvo el que se deduce de la regla que expresa que la referenciabilidad de una sublocalización de cualquier localización (**referenciable** o no) se determina mediante la organización (de elementos o de campo) solamente, lo cual es una propiedad del modo de la localización. Esto impone algunas restricciones en el establecimiento de la correspondencia de componentes que tienen a su vez, componentes **referenciables**.

ejemplos:

17.5 **PACK** *(1.1)*

19.14 **POS** *(1,0:15)* *(4.2)*

3.14 Modos dinámicos

3.14.1 Generalidades

Un modo dinámico es un modo tal que algunas de sus propiedades se conocen solamente en el momento de la ejecución. Los modos dinámicos son siempre modos parametrizados con uno o más parámetros de ejecución. En esta Recomendación | Norma Internacional se introducen denotaciones virtuales, con fines de descripción. Estas denotaciones virtuales van precedidas del símbolo y comercial (&) para distinguirlas de las notaciones efectivas que puedan aparecer en el texto de un programa CHILL.

3.14.2 Modos cadena dinámicos

denotación virtual: &<*nombre de modo cadena origen*> (<*expresión entera*>)

semántica: Un modo cadena dinámico es un modo cadena parametrizado de longitud no **constante**.

propiedades estáticas: Los modos cadena dinámicos tienen las mismas propiedades que los modos cadena, excepto las descritas a continuación.

propiedades dinámicas:

- Un modo cadena dinámico tiene una **longitud de cadena** dinámica, que es el valor entregado por *expresión entera*.
- Un modo cadena dinámico tiene un **límite superior** y un **límite inferior**, que son los valores entregados por **longitud de cadena** –1 y 0, respectivamente.

3.14.3 Modos matriz dinámicos

denotación virtual: *&<nombre de modo matriz dinámico>* (*<expresión discreta>*)

semántica: Un modo matriz dinámico es un modo matriz parametrizado de **límite superior** no **constante**.

propiedades estáticas: Los modos matriz dinámicos tienen las mismas propiedades que los modos matriz, excepto las descritas a continuación.

propiedades dinámicas:

- Un modo matriz dinámico tiene un **límite superior** dinámico, que es el valor entregado por *expresión discreta*, y un **número de elementos** dinámico, que es el valor entregado por:

  NUM (*expresión discreta*) – *NUM* (*límite inferior*) + 1

  donde *límite inferior* es el **límite inferior** del *nombre de modo matriz origen*.

3.14.4 Modos estructura parametrizada dinámicos

denotación virtual: *&<nombre de modo estructura variable origen>* (*<lista de expresiones>*)

semántica: Un modo estructura **parametrizada** dinámico es un modo estructura **parametrizada** con parámetros no **constantes**.

propiedades estáticas: Las propiedades estáticas de un modo estructura **parametrizada** dinámico son las de un modo estructura **parametrizado** estático, excepto lo siguiente:

- El conjunto de nombres de **campo** de un modo estructura **parametrizada** dinámico es el conjunto de los nombres de **campo** de su modo estructura **variable origen**.

propiedades dinámicas:

- Un modo estructura **parametrizada** dinámico tiene asociada una lista de valores, que es la lista de valores entregada por las expresiones de la *lista de expresiones*.

3.15 Modos moreta

3.15.1 Generalidades

sintaxis:

<modo moreta> ::=	*(1)*
<modo módulo>	*(1.1)*
| *<modo región>*	*(1.2)*
| *<modo tarea>*	*(1.3)*
| *<ejemplificación de modo moreta genérico>*	*(1.4)*
| *<modo interfaz>*	*(1.5)*
| *<nombre de modo moreta>* [(*<lista de parámetros efectivos>*)]	*(1.6)*

semántica:

modo módulo – una localización de *modo módulo* tiene las mismas propiedades que un *módulo* sin una *lista de sentencias de acción*.

modo región – una localización de *modo región* tiene las mismas propiedades que una *región*.

modo tarea – una localización de *modo tarea* tiene esencialmente la misma estructura que una *localización de modo módulo sin definiciones de proceso*. El acceso directo a los componentes de una de una localización, cuyo modo es un *modo tarea*, es mutuamente exclusivo. Un localización cuyo modo es un *modo tarea* puede ejecutarse concurrentemente con otros procesos (véase 11.1).

instanciación de modo moreta genérico – una *instanciación de modo moreta genérico* se obtiene estáticamente por una instanciación de una plantilla de modo moreta genérico (véase 10.11).

modo interfaz – un *modo interfaz* consta de especificaciones y firmas solamente.

condiciones estáticas:

Los *modos moreta* no son parametrizables.

Los *modos moreta* y las *plantillas de modos moreta genéricos* no pueden anidarse.

(1.1) – (1.5) sólo se permiten en definiciones de sínmodo y de neomodo; esto es, no se permiten modos moreta anónimos.

3.15.2 Modos módulo

sintaxis:

<modo módulo> ::= *(1)*
    <especificación de modo módulo> *(1.1)*
  | *<cuerpo de modo módulo>* *(1.2)*

<especificación de modo módulo> ::= *(2)*
    MODULE SPEC [[**ASSIGNABLE** [**FINAL**] | **ABSTRACT**] |
    [**NOT_ASSIGNABLE** [**ABSTRACT** | **FINAL**]]]
    <cláusula de herencia de módulo>
    {*<componente de especificación de módulo>*}***** [*<parte invariable>*]
    END [*<cadena de nombre simple>*] *(2.1)*

<cuerpo de modo módulo> ::= *(3)*
    MODULE BODY [[**ASSIGNABLE** [**FINAL**] | **ABSTRACT**] |
    [**NOT_ASSIGNABLE** [**ABSTRACT** | **FINAL**]]]
    <cláusula de herencia de módulo>
    {*<componente de cuerpo de módulo>*}***** [*<parte invariable>*]
    END [*<manejador>*] [*<cadena de nombre simple>*] *(3.1)*

<cláusula de herencia de módulo> ::= *(4)*
    [*<herencia de módulo>*] [*<cláusula de implementación>*] *(4.1)*

<módulo de herencia> ::= *(5)*
    BASED_ON *<nombre de modo módulo>* *(5.1)*

<cláusula de implementación> ::= *(6)*
    IMPLEMENTS *<nombre de modo interfaz>* { , *<nombre de modo interfaz>* }***** *(6.1)*

<componente de especificación de módulo> ::= *(7)*
    <componente de módulo común> *(7.1)*
  | *<sentencia de declaración>* *(7.2)*
  | *<sentencia de firma de procedimiento guardado simple>* *(7.3)*
  | *<sentencia de definición de procedimiento guardado en línea>* *(7.4)*
  | *<sentencia de especificación de proceso>* *(7.5)*
  | *<sentencia de definición de señal>* *(7.6)*
  | *<sentencia de otorgamiento>* *(7.7)*

<componente de cuerpo de módulo> ::= *(8)*
    <componente de módulo común> *(8.1)*
  | *<sentencia de definición de procedimiento guardado simple>* *(8.2)*
  | *<sentencia de definición de proceso>* *(8.3)*

<componente de módulo común> ::= *(9)*
    <sentencia de definición de sinónimo> *(9.1)*
  | *<sentencia de definición de sínmodo>* *(9.2)*
  | *<sentencia de definición de neomodo>* *(9.3)*
  | *<sentencia de toma>* *(9.4)*

<parte invariable> ::= *(10)*
    INVARIANT *<expresión booleana>* *(10.1)*

semántica: Un módulo define valores compuestos que consisten en una lista de componentes selecionables por nombres de componente.

Los valores de módulo pueden residir en localizaciones de **módulo** (compuestas).

Un *modo módulo* se define dando dos partes distintas: una *especificación de modo módulo* y un *cuerpo de modo módulo*.

La parte **especificación** define la interfaz de los valores de un *modo módulo*.

La parte **cuerpo** define el comportamiento de los valores de un *modo módulo*.

La *expresión boolena* de la *parte invariable* debe ser verdadera antes y después de cualquier llamada de cualquier procedimiento componente **público** o de un proceso componente **público**.

propiedades estáticas: Si se especifica el atributo **ASSIGNABLE**, el modo es un modo módulo **asignable**. Un modo módulo **asignable** puede utilizarse de la misma forma que un modo para el cual no se especifica **READ** (véase 3.3).

Si se especifica el atributo **NOT_ASSIGNABLE**, el nodo tiene la propiedad **no_asignable** que indica que la localización de ese modo no puede ser accedida para almacenar el valor y no puede ser accedida para copiar su valor.

Si no se especifica **ASSIGNABLE** ni **NOT_ASSIGNABLE**, el modo es **no_asignable** por defecto.

Si se especifica el atributo **ABSTRACT**, el modo es un modo **abstracto**.

Si se da una *herencia de módulo*, el modo MD que se define se deriva inmediatamente del modo MB dado en la *herencia de módulo* y MB es un modo base inmediato de MD.

Si se da una *cláusula de implementación* IC, el modo MD que se define se deriva inmediatamente de los modos dados en IC, y estos modos son modos de base inmediatos de MD.

El efecto de la *cláusula de herencia de módulo* es que el modo derivado se comporta como si contuviera todos los componentes de sus modos de base inmediatos excepto los procedimientos componentes constructor y destructor de estos modos de base. Si cualquiera de estos modos de base es, en sí, un modo derivado, esta herencia de componentes debe entenderse de una manera transitiva. Para la visibilidad, véase 12.2.

Un *componente especificación de módulo* contenido en una *especificación de modo módulo* M_S o SEIZEd en M_S, que es concedido por M_S, se denomina un componente **público** del modo de M_S.

Un *componente de especificación de módulo* contenido en una *especificación de modo módulo* M_S o SEIZEd en M_S, que no está otorgado por M_S, se denomina un componente **interno** del modo M_S.

Un *componente cuerpo de módulo* C contenido en un *cuerpo de modo módulo* M_B o SEIZEd en M_B, se denomina un componente **privado** del modo de M_B si C no es un componente **público** ni un componente **interno** del modo de M_B.

Un modo módulo **abstracto** tiene la propiedad **no_asignable.**

condiciones estáticas: Un modo módulo no puede utilizarse como el modo en una *definición de sinónimo.*

Para cada *especificación de modo módulo* debe haber un *cuerpo de modo módulo* con la misma cadena de nombre en la *ocurrencia de definición*.

Si está especificada, la *cadena de nombre simple* después de **END** debe ser igual a la cadena de nombre de la ocurrencia de definición de esta definición de modo. Esto se cumple para *especificación de modo módulo* y para *cuerpo de modo módulo*.

Si uno de los atributos **ASSIGNABLE**, **NOT_ASSIGNABLE**, **ABSTRACT** o **FINAL** se especifica en un módulo *especificación de modo módulo*, debe especificarse también en el *cuerpo de modo módulo* correspondiente.

Si una *especificación de modo módulo* contiene una *cláusula de herencia de módulo*, el *cuerpo de modo módulo* correspondiente debe contener la misma *cláusula de herencia de módulo*.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en un una *firma de procedimiento guardado simple*, el procedimiento tiene la propiedad **incompleto**.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en una *sentencia de firma de procedimiento guardado simple*, el procedimiento tiene la propiedad **público**.

Para cada *sentencia de firma de procedimiento guardado* **simple**, **completo** S de una *especificación de modo módulo*, el correspondiente *cuerpo de modo módulo* debe contener una *sentencia de definición de procedimiento guardado simple* correspondiente D, donde la *firma de procedimiento guardado* de S concuerda con la *definición de procedimiento guardado* de D (véase 12.1.3).

Si P es una *firma de procedimiento guardado* **simple**, **incompleto** de una *especificación de modo módulo*, el *cuerpo de modo módulo* correspondiente no debe contener una *definición de procedimiento guardado* que concuerde con P.

Para cada *especificación de proceso* de una *especificación de modo módulo*, el *cuerpo de modo módulo* correspondiente debe contener una *definición de proceso* correspondiente (véase 12.1.3).

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *sentencia de firma de procedimiento guardado simple*, este procedimiento debe ser **público**.

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *firma de procedimiento guardado simple* PD contenida en una *especificación de modo módulo* M, el modo base inmediato MB de M debe contener o haber heredado una *firma de procedimiento guardado simple* **pública** PB, donde PB concuerda con PD, y PB no es ni un constructor ni un destructor, y PB no es SEIZEd.

Un *modo módulo* es un modo módulo **abstracto** si contiene al menos un *procedimiento de componente* **incompleto** (véase 10.4). En este caso debe especificarse el atributo **ABSTRACT**.

Un nombre de modo módulo **abstracto** sólo puede utilizarse como el modo de módulo *nombre* en una *herencia de módulo* o como un *modo referenciado*.

Si un modo módulo M tiene al menos un (sub)componente con la **propiedad no-valor**, M también tiene la **propiedad no-valor** y el atributo **ASSIGNABLE** no debe especificarse (véase 12.1.1.5).

Si un *modo módulo* M contiene el atributo **FINAL**, M se denomina un modo módulo **final**. Un modo módulo **final** no puede utilizarse como un modo base en una *herencia* moreta.

Un modo módulo **final** no contiene un procedimiento componente **incompleto**.

3.15.3 Modos región

sintaxis:

<modo región> ::= *(1)*
    <especificación de modo región> *(1.1)*
  | *<cuerpo de modo región>* *(1.2)*

<especificación de modo región> ::= *(2)*
    REGION SPEC [**ABSTRACT** | **FINAL**] [*<herencia de región>*]
    {*<componente especificación de región>*}***** [*<parte invariable>*]
    END [*<cadena de nombre simple>*] *(2.1)*

<cuerpo de modo región> ::= *(3)*
    REGION BODY [**ABSTRACT** | **FINAL**] [*<cláusula de herencia de región>*]
    {*<comunicación cuerpo de región>*}***** [*<parte invariable>*]
    END [*<manejador>*] [*<cadena de nombre simple>*] *(3.1)*

<cláusula de herencia de región> ::= *(4)*
    [*<herencia de región>*] [*<cláusula de implementación>*] *(4.1)*

<herencia de región> ::= *(5)*
    BASED_ON {*<nombre de modo módulo>* | *<nombre de modo región>*} *(5.1)*

<componente especificación de región> ::= *(6)*
    <componente módulo común> *(6.1)*
  | *<sentencia de declaración>* *(6.2)*
  | *<sentencia de firma de procedimiento guardado simple>* *(6.3)*
  | *<sentencia de definición de señal>* *(6.4)*
  | *<sentencia de concesión>* *(6.5)*

<componente cuerpo de región> ::= *(7)*
    <componente módulo común> *(7.1)*
  | *<sentencia de definición de procedimiento guardado simple>* *(7.2)*

semántica: Un *modo región* define valores compuestos que consisten en una lista de componentes seleccionables por nombres de componentes.

Los valores de región pueden residir en localizaciones de región (compuestas).

Un *modo región* se define dando dos partes distintas: una *especificación de modo región* y un *cuerpo de modo región*.

La parte **especificación** define la interfaz de los valores del *modo región*.

La parte **cuerpo** define el componente de los valores del *modo región*.

La *expresión booleana* de la *parte invariable* debe ser verdadera antes y después de cualquier llamada de un procedimiento componente **público**.

propiedades estáticas: Un *modo región* siempre tiene la propiedad **no_asignable**.

Si el atributo **ABSTRACT** está especificado, el modo es un modo **abstracto**.

Si se da una *herencia de región*, el modo MD que se define se deriva inmediatamente del modo MB dado en la *herencia de región*, y MB es un modo base inmediato de MD.

Si se da una *cláusula de implementación* IC, el modo MD que se define se deriva inmediatamente de los modos dados en IC, y estos modos son modos de base inmediatos de MD.

El efecto de la *cláusula de herencia de región* es que el modo derivado se comporta como si contuviera todos los componentes de sus modos de base inmediatos excepto los procedimientos de componentes constructor y destructor de estos modos de base. Si cualquiera de estos modos de base es, en sí, un modo derivado, esta herencia de componentes debe entenderse de manera transitiva. Para la visibilidad, véase 12.2.

Un *componente especificación de región* contenido en una *especificación de modo región* M_S o SEIZEd en M_S, que es otorgado por M_S, se denomina un componente **público** del modo de M_S.

Un *componente especificación de región* contenido en una *especificación de modo región* M_S o SEIZEd en M_S, que no es otorgado por M_S, se denomina un componente **interno** del modo de M_S.

Un *componente de cuerpo de región* C contenido en un *cuerpo de modo región* M_B o SEIZEd en M_B, se denomina un componente **privado** del modo de M_B, si C no es un componente **público** ni un componente **interno** del modo de M_B.

condiciones estáticas: Un modo región no puede utilizarse como el modo en una *definición de sinónimo*.

Para cada *especificación de modo región*, debe haber un *cuerpo de modo región* con la misma cadena de nombre de la *ocurrencia de definición*.

Si se especifica, la *cadena de nombre simple* después de **END** debe ser igual a la cadena de nombre de la ocurrencia de definición de esta definición de modo. Esto se cumple para *especificación de modo región* y para *cuerpo de modo región*.

Si el atributo **ABSTRACT** o **FINAL** está especificado en una *especificación de modo región*, debe especificarse también en el *cuerpo de modo región* correspondiente.

Si una *especificación de modo región* contiene una *cláusula de herencia de región*, el *cuerpo de modo región* correspondiente debe contener la misma *cláusula de herencia de región*.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en una *firma de procedimiento guardado simple*, este procedimiento tiene la propiedad **incompleto**.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en una *sentencia de firma de procedimiento guardado simple*, este procedimiento debe ser **público**.

Para cada *sentencia de firma de procedimiento guardado* **simple**, **completo** S de una *especificación de modo región*, el *cuerpo de modo región* correspondiente debe contener una *sentencia de definición de procedimiento guardado simple* D (véase 12.1.3) en la que la *firma de procedimiento guardado* de S concuerda con la *definición de procedimiento guardado* de D.

Si P es una *firma de procedimiento guardado* **simple**, **incompleto** de una *especificación de modo región*, el *cuerpo de modo región* correspondiente debe contener una *definición de procedimiento guardado simple* que concuerda con P.

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *sentencia de firma de procedimiento guardado simple*, este procedimiento debe ser **público.**

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *firma de procedimiento guardado simple* PD contenida en una *especificación de modo región* M, el modo base inmediato MB de M debe contener o haber heredado una *firma de procedimiento guardado simple* **público** PB donde PB concuerda con PD, y PB no es un constructor ni un destructor, y PB no es SEIZEd.

Un *modo región* es un modo región **abstracto** si contiene al menos un *procedimiento componente* **incompleto** (véase 10.4). En este caso debe especificarse el atributo **ABSTRACT**.

Un nombre de modo región **abstracto** sólo puede utilizarse como el *nombre* de modo región en una *herencia de región* o como un *modo referenciado*.

Una especificación de *modo región* no debe otorgar ninguna localización.

Si el modo base de un *modo región* es un *modo módulo* M, M debe tener la propiedad **no_asignable**, no debe otorgar ninguna localización y no debe contener ningún *procedimiento componente guardado en línea* ni ningún *proceso componente*.

Si un *modo región* M contiene el atributo **FINAL**, M se denomina un modo región **final**. Un modo región final no puede utilizarse como un modo base en una *herencia de región*.

Un modo región **final** no puede contener un procedimiento componente **incompleto**.

3.15.4 Modos tarea

sintaxis:

<modo tarea> ::= *(1)*
    <especificación de modo tarea> *(1.1)*
  | *<cuerpo de modo tarea>* *(1.2)*

<especificación de modo tarea> ::= *(2)*
    TASK SPEC [**ABSTRACT** | **FINAL**] [*<cláusula de herencia de tarea>*]
    [*<parte invariable>*] {*<componente especificación de tarea>*}*
    END [*<nombre de cadena simple>*] *(2.1)*

<cuerpo de modo tarea> ::= *(3)*
    TASK BODY [**ABSTRACT** | **FINAL**] [*<cláusula de herencia de tarea>*]
    {*<componente cuerpo de tarea>*}* [*<parte invariable>*]
    END [*<manejador>*] [*<cadena de nombre simple>*] *(3.1)*

<cláusula de herencia de tarea> ::= *(4)*
    [*<herencia de tarea>*] [*<cláusula de implementación>*] *(4.1)*

<herencia de tarea> ::= *(5)*
    BASED_ON {*<nombre de modo módulo>* | *<nombre de modo tarea>*} *(5.1)*

<componente especificación de tarea> ::= *(6)*
    <componente especificación de región> *(6.1)*

<componente cuerpo de tarea> ::= *(7)*
    <componente cuerpo de región> *(7.1)*

semántica: Un *modo tarea* define valores compuestos que consisten en una lista de componentes seleccionables por nombres de componentes.

Los valores de tareas pueden residir en localizaciones de tarea (compuestas).

Un *modo tarea* se define dando dos partes distintas: una *especificación de modo tarea* y un *cuerpo de modo tarea*.

La parte **especificación** define la interfaz de los valores del *modo tarea*.

La parte **cuerpo** define el comportamiento de los valores del *modo tarea*.

La *expresión booleana* de la *parte invariable* debe ser verdadera antes y después de cualquier llamada de un procedimiento componente **público**.

propiedades estáticas: Un *modo tarea* siempre tiene la propiedad **no_asignable**.

Si el atributo **ABSTRACT** está especificado, el modo es un modo **abstracto**.

Si se da una *herencia de región*, el modo MD que se define se deriva inmediatamente del modo MB dado en la *herencia de tarea*, y MB es un modo base inmediato de MD.

Si se da una *cláusula de implementación* IC, el modo MD que se define se deriva inmediatamente de los modos dados en IC, y estos modos son modos de base inmediatos de MD.

El efecto de la *cláusula de herencia de tarea* es que el modo derivado se comporta como si contuviera todos los componentes de sus modos de base inmediatos excepto los procedimientos de componentes constructor y destructor de estos modos de base. Si cualquiera de estos modos de base es, en sí, un modo derivado, esta herencia de componentes debe entenderse de manera transitiva. Para la visibilidad, véase 12.2.

Un *componente especificación de tarea* contenido en una *especificación de modo tarea* M_S o SEIZEd en M_S, que es otorgado por M_S, se denomina un componente **público** del modo de M_S.

Un *componente especificación de tarea* contenido en una *especificación de modo tarea* M_S o SEIZEd en M_S, que no es otorgado por M_S, se denomina un componente **interno** del modo de M_S.

Un *componente de cuerpo de tarea* C contenido en un *cuerpo de modo región* M_B o SEIZEd en M_S, se denomina un componente **privado** del modo de M_B, si C no es un componente **público** ni un componente **interno** del modo de M_B.

condiciones estáticas: Un modo tarea no puede utilizarse como el modo en una *definición de sinónimo*.

Para cada *especificación de modo tarea*, debe haber un *cuerpo de modo tarea* con la misma cadena de nombre en la *ocurrencia de definición*.

Si se especifica, la *cadena de nombre simple* después de **END** debe ser igual a la cadena de nombre de la ocurrencia de definición de esta definición de modo. Esto se cumple para *especificación de modo tarea* y para *cuerpo de modo tarea*.

Si el atributo **ABSTRACT** o **FINAL** se especifica en una *especificación de modo tarea*, debe especificarse también en el *cuerpo de modo tarea* correspondiente.

Si una *especificación de modo tarea* contiene una *cláusula de herencia de tarea*, el *cuerpo de modo tarea* correspondiente debe contener la misma *cláusula de herencia de tarea*.

Todos los procedimientos componentes públicos de un modo tarea deben tener solamente parámetros IN y no deben tener una *espec de resultado*.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en una *firma de procedimiento guardado simple*, este procedimiento tiene la propiedad **incompleto**.

Si el atributo **INCOMPLETE** (véase 10.4) está especificado en una *sentencia de firma de procedimiento guardado simple*, este procedimiento debe ser **público**.

Para cada *sentencia de firma de procedimiento guardado* **simple**, **completo** S de una *especificación de modo tarea*, el *cuerpo de modo tarea* correspondiente debe contener una *sentencia de definición de procedimiento guardado simple* D (véase 12.1.3) en la que la *firma de procedimiento guardado* de S concuerda con la *definición de procedimiento guardado* de D.

Si P es una *firma de procedimiento guardado* **simple**, **incompleto** de una *especificación de modo tarea*, el *cuerpo de modo tarea* correspondiente debe contener una *definición de procedimiento guardado simple* que concuerda con P.

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *sentencia de firma de procedimiento guardado simple*, este procedimiento debe ser **público**.

Si el atributo **REIMPLEMENT** (véase 10.4) se especifica en una *firma de procedimiento guardado simple* PD contenida en una *especificación de modo tarea* M, el modo base inmediato de M debe contener o haber heredado una *firma de procedimiento guardado simple* **público** PB donde PB concuerda con PD, y PB no es un constructor ni un destructor, y PB no es SEIZEd.

Un *modo tarea* es un modo tarea **abstracto** si contiene al menos un *procedimiento componente* **incompleto** (véase 10.4). En este caso debe especificarse el atributo **ABSTRACT**.

Un nombre de modo tarea **abstracto** sólo puede utilizarse como el *nombre* de modo región en una *herencia de tarea* o como un *modo referenciado*.

Una especificación de *modo tarea* no debe otorgar ninguna localización.

Si el modo base inmediato de un *modo tarea* es un *modo módulo* M, M debe tener la propiedad **no_asignable**, no debe otorgar ninguna localización, no debe contener ningún *procedimiento componente guardado en línea* ni ningún *proceso componente*, y debe contener solamente procedimientos **públicos** que cumplen las restricciones de procedimientos componentes **públicos** de modos tarea.

Si un *modo tarea* M contiene el atributo **FINAL**, M se denomina un modo tarea **final**. Un modo tarea **final** no puede utilizarse como un modo base en una *herencia de tarea*.

Un modo tarea **final** no puede contener un procedimiento componente **incompleto**.

3.15.5 Modos interfaz

sintaxis:

<modo interfaz> ::= *(1)*
INTERFACE [*<herencia de interfaz>*] {*<componente interfaz>*}*
END [*<cadena de nombre simple>*] *(1.1)*

<herencia de interfaz> ::= *(2)*
BASED_ON *<nombre de modo interfaz>* { , *<nombre de modo interfaz>* }* *(2.1)*

<componente interfaz> ::= *(3)*
<componente módulo común> *(3.1)*
| *<sentencia de declaración>* *(3.2)*
| *<sentencia de firma de procedimiento guardado simple>* *(3.3)*
| *<sentencia de especificación de proceso>* *(3.4)*
| *<sentencia de definición de señal>* *(3.5)*

semántica: Un *modo interfaz* define un modo moreta que sólo puede utilizarse como un modo base en la definición de otros modos moreta y es el modo referenciado de un modo referencia ligada.

propiedades estáticas: Si se da *herencia de interfaz* II, el modo MD que se define se deriva inmediatamente de los modos dados en II, y estos modos son modos base inmediatos de MD.

El efecto de la *cláusula de herencia de interfaz* es que el modo derivado se comporta como si contuviera todos los componentes de sus modos base inmediatos. Si cualquiera de estos modos base es, en sí, un modo derivado, esta herencia de componentes debe entenderse de una manera transitiva. Para la visibilidad, véase 12.2.

Todos los *componentes de interfaz* (incluidos los SEIZEd) son implícitamente GRANTed (otorgados), por lo que todos se denominan componentes **públicos**.

Un *modo interfaz* es un modo **abstracto**.

condiciones estáticas: Un modo interfaz no puede utilizarse como el modo en una *definición de sinónimo*.

Si se especifica, la *cadena de nombre simple* después de **END** debe ser igual a la cadena de nombre de la ocurrencia de definición de esta definición de modo.

El atributo **INCOMPLETE** (véase 10.4) debe especificarse en todas las *firmas de procedimiento guardado simple*; por tanto, todos los procedimientos tienen la propiedad **incompleto**.

El atributo **REIMPLEMENT** (véase 10.4) no debe especificarse en una *sentencia de firma de procedimiento guardado simple* de un *componente interfaz*.

4 Localizaciones y sus accesos

4.1 Declaraciones

4.1.1 Generalidades

sintaxis:

<sentencia de declaración> ::= *(1)*
DCL *<declaración>* { , *<declaración>* }* ; *(1.1)*

<declaración> ::= *(2)*
<declaración de localización> *(2.1)*
| *<declaración de identidad-loc>* *(2.2)*

semántica: Una sentencia de declaración declara que uno o más nombres constituyen un acceso a una localización.

ejemplos:

6.9 **DCL** *j INT := julian_day_number,*

d, m, y INT; *(1.1)*

11.36 *starting_square* **LOC** *:= b(m.lin_1)(m.col_1)* *(2.2)*

4.1.2 Declaraciones de localización

sintaxis:

<declaración de localización> ::= *(1)*
<lista de ocurrencias de definición> *<modo>* [**STATIC**] [*<inicialización>*] *(1.1)*

<inicialización> ::= *(2)*
<inicialización ligada al dominio> *(2.1)*
| *<inicialización ligada al tiempo de vida>* *(2.2)*
| *<inicialización ligada a moreta>* *(2.3)*

<inicialización ligada al dominio> ::= *(3)*
<símbolo de asignación> <valor> [*<manejador>*] *(3.1)*

<inicialización ligada al tiempo de vida> ::= *(4)*
INIT *<símbolo de asignación> <valor constante>* *(4.1)*

<inicialización ligada a moreta> ::= *(5)*
([*<lista de parámetros efectivos constructor>*]) [*<manejador>*] *(5.1)*

semántica: Una declaración de localización crea tantas localizaciones como *ocurrencias de definición* hay especificadas en la *lista de ocurrencias de definición*.

Con la *inicialización ligada al dominio* se evalúa el *valor* cada vez que se entra en el dominio en que está situada la declaración (véase el 10.2), asignándose a las localizaciones el valor entregado. Antes de evaluar el *valor*, la localización o localizaciones contienen el valor **indefinido**.

Con la *inicialización ligada al tiempo de vida*, se asigna a la, o las localizaciones el valor entregado por el *valor constante* sólo una vez, al comienzo del tiempo de vida de la localización (véanse 10.2 y 10.9).

Si el *modo* es un *modo moreta*, primero todas las inicializaciones en los componentes se efectúan en orden textual. Si se especifica una lista de parámetros (que puede ser una lista vacía), el **constructor** correspondiente del modo se aplica a la localización últimamente creada. Si el *modo* es un *modo tarea*, se empieza la tarea perteneciente a la localización últimamente creada.

Especificar que no hay *inicialización* es semánticamente equivalente a la especificación de una *inicialización ligada al tiempo de vida* con el valor **indefinido** (véase 5.3.1).

El significado del valor **indefinido** como inicialización para una localización que tiene asociado un modo con la **propiedad parametrizada con marcadores** o la **propiedad de no valor** es el siguiente:

- **propiedad parametrizada con marcadores**: la(s) sublocalización(es) campo **marcador** creada(s) se inicializa(n) con el correspondiente valor de parámetro.
- **propiedad de no-valor**:
  - El suceso y/o (sub)localización(es) tampón creado(s) se inicializa(n) a "vacío", es decir, no se asocian procesos demorados al suceso o tampón, ni hay mensajes en la memoria tampón.
  - La región y/o (sub)localización(es) tarea creada(s) se inicializa(n) a "vacío", es decir, no se les asocian procesos demorados.
  - La(s) (sub)localización(es) de asociación creada(s) se inicializa(n) a "vacío", es decir, no contiene(n) una asociación.
  - La(s) (sub)localización(es) de acceso creada(s) se inicializa(n) a "vacío", es decir, no está(n) conectada(s) a una asociación.
  - La(s) (sub)localización(es) tienen una sublocalización **registro de texto** que se inicializa con una cadena vacía en una sublocalización **acceso** que está inicializada con "vacío", es decir, no está conectada a una asociación.
- Las semántica de **STATIC** y de *manejador* pueden verse en 10.9 y en la cláusula 8, respectivamente.

Si el tiempo de vida de una localización **moreta** L termina y el modo de la localización contiene un destructor, este destructor se aplica a L (véase 10.2).

propiedades estáticas: Una *ocurrencia de definición* en una *declaración de localización* define un nombre de **localización**. El modo asociado a nombre de **localización** es el *modo* especificado en la *declaración de localización*. Un nombre de **localización** es **referenciable**.

condiciones estáticas: La clase del *valor* o *valor constante* debe ser **compatible** con el *modo*, y el valor entregado debe ser uno de los valores definidos por el *modo*, o el valor **indefinido**.

Si el *modo* tiene la **propiedad de lectura solamente**, debe especificarse la *inicialización*. Si el *modo* tiene la **propiedad de no-valor**, no debe especificarse la *inicialización ligada al dominio*.

Si se especifica la *inicialización*, el *valor* debe ser **regionalmente seguro** para la localización (véase 11.2.2).

condiciones dinámicas: En el caso de *inicialización ligada al dominio*, se aplican las condiciones de asignación de *valor* con respecto al *modo* (véase 6.2).

ejemplos:

5.7 *k2, x, w, t, s, r BOOL* *(1.1)*

6.9 *:= julian_day_number* *(3.1)*

8.4 **INIT** *:= ['A':'Z']* *(4.1)*

4.1.3 Declaraciones de identidad-loc

sintaxis:

<declaración de identidad-loc> ::= *(1)*
<lista de ocurrencias de definición> <modo> **LOC** [**DYNAMIC**]
<símbolo de asignación> <localización> [*<manejador>*] *(1.1)*

semántica: Una declaración de identidad-loc crea tantos nombres de acceso a la localización especificada como *ocurrencias de definición* hay especificadas en la *lista de ocurrencias de definición*. El modo de la localización sólo puede ser dinámico si se especifica **DYNAMIC**.

Si la *localización* se evalúa dinámicamente, se efectúa esta evaluación cada vez que se entra en el dominio en el que está situada la declaración de identidad-loc. En este caso, un nombre declarado denota una localización **indefinida** antes de la primera evaluación durante el tiempo de vida del acceso denotado por el nombre declarado (véanse 10.2 y 10.9).

propiedades estáticas: Una *ocurrencia de definición* en una *declaración de identidad-loc* define un nombre de **identidad-loc**. El modo asociado a un nombre de **identidad-loc** es, si no se especifica **DYNAMIC**, el *modo* especificado en la *declaración de identidad-loc*; en otro caso, es la versión dinámicamente parametrizada del mismo que tiene los mismos parámetros que el modo de la *localización*.

No está permitido crear una localización de un modo moreta con **DYNAMIC**.

Un nombre **de identidad-loc** es **referenciable** si y sólo si lo es la *localización* especificada.

condiciones estáticas: Si se especifica **DYNAMIC** en la *declaración de identidad-loc*, el *modo* debe ser **parametrizable**. El *modo* especificado debe ser de **lectura dinámica compatible** con el modo de la *localización* si se especifica **DYNAMIC**, y en otro caso, de **lectura compatible** con el modo de la *localización*.

La **localización** no debe ser un *elemento de cadena* ni *segmento de cadena* en el que el *modo* de la *localización* *cadena* es un modo cadena **variable**.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* o *TAGFAIL* si **DYNAMIC** está especificado y falla la citada comprobación de **lectura dinámica compatible**.

ejemplo:

11.36 *starting square* **LOC** *:= b(m.lin_1)(m.col_1)* *(1.1)*

4.2 Localizaciones

4.2.1 Generalidades

sintaxis:

<localización> ::= *(1)*
<nombre de acceso> *(1.1)*
| *<referencia ligada desreferenciada>* *(1.2)*
| *<referencia libre desreferenciada>* *(1.3)*
| *<descriptor desreferenciado>* *(1.4)*
| *<elemento de cadena>* *(1.5)*
| *<segmento de cadena>* *(1.6)*
| *<elemento de matriz>* *(1.7)*
| *<segmento de matriz>* *(1.8)*
| *<campo de estructura>* *(1.9)*
| *<llamada a procedimiento que entrega una localización>* *(1.10)*
| *<llamada a rutina incorporada que entrega una localización>* *(1.11)*
| *<conversión de localización>* *(1.12)*
| *<localización moreta predefinida>* *(1.13)*

semántica: Una localización es un objeto que puede contener valores. Debe accederse a localizaciones para almacenar u obtener un valor.

propiedades estáticas: Una *localización* tiene las siguientes propiedades:

- Un modo, definido en los puntos correspondientes. Este modo es estático o dinámico.
- Es **estático** o no (véase 10.9).
- Es **intrarregional** o **extrarregional** (véase 11.2.2).
- Es **referenciable** o no. La definición del lenguaje requiere que ciertas localizaciones sean **referenciables** y otras no **referenciables**, como se define en los puntos correspondientes. Una implementación puede extender la referenciabilidad a otras localizaciones, salvo cuando esté explícitamente desautorizado.

4.2.2 Nombres de acceso

sintaxis:

<nombre de acceso> ::= *(1)*
<nombre de localización> *(1.1)*
| *<nombre de identidad-loc>* *(1.2)*
| *<nombre de enumeración de localización>* *(1.3)*
| *<nombre de localización hacer-con>* *(1.4)*

semántica: Un nombre de acceso entrega una localización. Un nombre de acceso será uno de los siguientes:

- un nombre de **localización**, es decir, un nombre explícitamente declarado en una *declaración de localización* o declarado implícitamente en un *parámetro formal* sin el atributo **LOC**;
- un nombre de **identidad-loc**, es decir, un nombre explícitamente declarado en una *declaración de identidad-loc* o implícitamente en un *parámetro formal* con el atributo **LOC**;
- un nombre de **enumeración de localización**, es decir un *contador de bucle* en una *enumeración de localización*;
- un nombre de **localización hacer-con**, es decir un nombre de **campo** utilizado como acceso directo en una *acción* con una *parte con*.

Si la localización designada por un *nombre de localización hacer-con* es un campo variable de una localización estructura variable sin marcadores, la semántica se define en la implementación.

propiedades estáticas: El modo (posiblemente dinámico) asociado a un *nombre de acceso* es el modo del *nombre de localización*, *nombre de identidad-loc*, *nombre de enumeración de localización*, *nombre de localización hacer-con*, respectivamente.

Un *nombre de acceso* es **referenciable** si y sólo si es un *nombre de localización*, un *nombre de identidad-loc* **referenciable**, un *nombre de enumeración de localización* **referenciable**, o un *nombre de localización hacer-con* **referenciable**.

condiciones dinámicas: Cuando se accede mediante un *nombre de identidad-loc*, éste no debe denotar una localización **indefinida**.

Cuando se accede mediante un *nombre de identidad-loc* a una localización que es un campo **variable**, se aplican las condiciones de acceso de campo variable para la localización (véase 4.2.10). El acceso mediante un *nombre de localización hacer-con* origina una excepción *TAGFAIL* si la localización denotada es un campo **variable** y no se satisfacen las condiciones de acceso de campo variable para la localización.

ejemplos:

4.12 *a* *(1.1)*

11.39 *starting* *(1.2)*

15.35 *each* *(1.3)*

5.10 *c1* *(1.4)*

4.2.3 Referencias ligadas desreferenciadas

sintaxis:

<referencia ligada desreferenciada> ::= *(1)*
<valor primitivo de referencia ligada> –> [*<nombre de modo>*] *(1.1)*

semántica: Una referencia ligada desreferenciada entrega la localización que está referenciada por el valor de referencia ligada.

propiedades estáticas: El modo asociado a una *referencia ligada desreferenciada* es el *nombre de modo* si se ha especificado; en otro caso, es el modo **referenciado** del *valor primitivo de referencia ligada*. Una *referencia ligada desreferenciada* es **referenciable**.

condiciones estáticas: El *valor primitivo de referencia ligada* debe ser **fuerte**. Si se especifica el *nombre de modo* facultativo, debe ser de **lectura compatible** con el modo **referenciado** del modo del *valor primitivo de referencia ligada*.

condiciones dinámicas: El tiempo de vida de una localización referenciada no debe haber terminado.

Se produce la excepción *EMPTY* si el *valor primitivo de referencia ligada* entrega el valor *NULL*.

Si la localización referenciada es un campo **variable**, deben cumplirse las condiciones de acceso de campo variable para la localización (véase 4.2.10).

ejemplo:

10.54 p –> *(1.1)*

4.2.4 Referencias libres desreferenciadas

sintaxis:

<referencia libre desreferenciada> ::= *(1)*
<valor primitivo de referencia libre> –> <nombre de modo> *(1.1)*

semántica: Una referencia libre desreferenciada da la localización que está referenciada por el valor de referencia libre.

propiedades estáticas: El modo asociado a una *referencia libre desreferenciada* es el *nombre de modo*. Una *referencia libre desreferenciada* es **referenciable**.

condiciones estáticas: El *valor primitivo de referencia libre* debe ser **fuerte**.

condiciones dinámicas: El tiempo de vida de una localización referenciada no debe haber terminado.

Se produce la excepción *EMPTY* si el *valor primitivo de referencia libre* entrega el valor *NULL*.

El *nombre de modo* debe ser de **lectura compatible** con el modo de la localización referenciada.

Si la localización referenciada es un campo **variable**, deben cumplirse las condiciones de acceso de campo variable para la localización (véase 4.2.10).

4.2.5 Descriptores desreferenciados

sintaxis:

<descriptor desreferenciado> ::= *(1)*
<valor primitivo de descriptor> → *(1.1)*

semántica: Un descriptor desreferenciado da la localización que es referenciada por el valor de descriptor.

propiedades estáticas: El modo dinámico asociado a un *descriptor referenciado* se construye como sigue:

&nombre de modo origen (*<parámetro>* { , *<parámetro>* }*)

donde el *nombre de modo origen* es un nombre **sínmodo** virtual **sinónimo** del modo **origen referenciado** del modo del *valor primitivo de descriptor*, y en el cual los parámetros, según el modo **origen referenciado** son:

- la **longitud de cadena** dinámica, en el caso de un modo cadena;
- el **límite superior** dinámico, en el caso de un modo matriz;
- la lista de valores asociados con el modo de la localización estructura parametrizada, en el caso de un modo estructura **variable**.

Un *descriptor referenciado* es **referenciable**.

condiciones estáticas: El *valor primitivo de descriptor* debe ser **fuerte**.

condiciones dinámicas: El tiempo de vida de la localización referenciada no debe haber terminado.

Se produce la excepción *EMPTY* si el *valor primitivo de descriptor* entrega *NULL*.

Si la localización referenciada es un campo **variable**, deben cumplirse las condiciones de acceso de campo variable para la localización (véase 4.2.10).

ejemplo:

8.11 input –> *(1.1)*

4.2.6 Elementos de cadena

sintaxis:

<elemento de cadena> ::= *(1)*
<localización cadena> (<elemento de arranque>) *(1.1)*

<elemento de arranque> ::= *(2)*
<expresión entera> *(2.1)*

semántica: Un elemento de cadena entrega una (sub)localización, que es el elemento de la localización cadena especificada indicada por *elemento de arranque*.

propiedades estáticas: El modo ligado al *elemento de cadena* es el modo **elemento** del modo de la *localización de cadena*.

Si el modo de la *localización cadena* es un modo cadena **variable**, entonces el *elemento de cadena* no es **referenciable**.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si no se cumple la relación siguiente:

$$0 \leq NUM(start\ element) \leq L - 1$$

donde L es la **longitud efectiva** de la *localización cadena*.

ejemplo:

18.16 string –> (i) *(1.1)*

4.2.7 Segmentos de cadena

sintaxis:

<segmento de cadena> ::= *(1)*
<localización cadena>(elemento de la izquierda> : <elemento de la derecha>) *(1.1)*
| *<localización cadena>(<elemento de arranque>* **UP** *<tamaño de segmento>)* *(1.2)*

<elemento de la izquierda> ::= *(2)*
<expresión entera> *(2.1)*

<elemento de la derecha> ::= *(3)*
<expresión entera> *(3.1)*

<tamaño de segmento> ::= *(4)*
<expresión entera> *(4.1)*

semántica: Un segmento de cadena entrega una localización cadena (posiblemente dinámica), que es la parte de la localización cadena especificada indicada por *elemento de la izquierda* y *elemento de la derecha* o *elemento de arranque* y *tamaño de segmento*. La longitud (posiblemente dinámica) del tamaño de segmento se determina a partir de las expresiones especificadas.

Un *segmento de cadena* en el que el *elemento de la derecha* entrega un valor inferior al que entrega el *elemento de la izquierda*, o en que el *tamaño de segmento* entrega un valor no positivo, denota una cadena vacía.

propiedades estáticas: El modo (posiblemente dinámico) asociado a un *tamaño de segmento* es un modo cadena **parametrizado** construido como sigue:

&nombre (*tamaño de segmento*)

donde *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo (posiblemente dinámico) de la *localización cadena* si es un modo cadena **fijo**, y en otro caso, **sinónimo** del modo **componente**, y donde *tamaño de segmento* es:

NUM (*elemento de la derecha*) – *NUM* (*elemento de la izquierda*) + 1

o

NUM (*tamaño de segmento*).

Sin embargo, si se denota una cadena vacía, *tamaño de cadena* es 0. El modo asociado a un *tamaño de segmento* es estático si *tamaño de segmento* es **literal**, es decir, *elemento de la izquierda* y *elemento de la derecha* son **literales** o *tamaño de segmento* es **literal**; en otro caso, el modo es dinámico.

Si el modo de la *localización cadena* es un modo cadena **variable**, entonces el *segmento de cadena* no es **referenciable**.

condiciones estáticas: Deben cumplirse las siguientes relaciones :

$0 \leq NUM$ (*elemento de la izquierda*) $\leq L - 1$

$0 \leq NUM$ (*elemento de la derecha*) $\leq L - 1$

$0 \leq NUM$ (*elemento de arranque*) $\leq L - 1$

NUM (*elemento de arranque*) + NUM (*tamaño de segmento*) $\leq L$

donde L es la **longitud efectiva** de la *localización cadena*. Si L y el valor de todas las *expresiones enteras* son conocidos estáticamente, las relaciones pueden comprobarse estáticamente.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si falla una parte dinámica de la comprobación de las relaciones mencionadas.

ejemplos:

18.26 blanks (count : 9) *(1.1)*

18.23 string –> (scanstart **UP** *10)* *(1.2)*

4.2.8 Elementos de matriz

sintaxis:

<elemento de matriz> ::= *(1)*
<localización matriz> (*<lista de expresiones>*) *(1.1)*

<lista de expresiones> ::= *(2)*
<expresión> { , *<expresión>* }* *(2.1)*

sintaxis derivada: La notación: (*<lista de expresiones>*) es la sintaxis derivada para:

(*<expresión>*) { (*<expresión>*) }*

en la que hay tantas expresiones entre paréntesis como expresiones hay en la *lista de expresiones*. En consecuencia, un *elemento de matriz* tiene sólo una expresión (índice) en la sintaxis estricta.

semántica: Un elemento de matriz entrega una (sub)localización que es el elemento de la localización matriz especificada indicada por *expresión*.

propiedades estáticas: El modo asociado a un *elemento de matriz* es el modo **elemento** del modo de la *localización matriz*.

Un *elemento de matriz* es **referenciable** si la **organización de elementos** del modo de la *localización matriz* es **NOPACK**.

condiciones estáticas: La clase de *expresión* debe ser **compatible** con el modo **índice** del modo de la *localización matriz*.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si no se cumple la relación siguiente:

$L \leq$ *expresión* $\leq U$

donde L y U son respectivamente el **límite inferior** y el **límite superior** (posiblemente dinámicos) del modo de la *localización matriz*.

ejemplo:

11.36 b(m.lin_1)(m.col_1) *(1.1)*

4.2.9 Segmentos de matriz

sintaxis:

<segmento de matriz> ::= *(1)*
<localización matriz> (<elemento inferior> : <elemento superior>) *(1.1)*
| *<localización matriz> (<primer elemento>* **UP** *<tamaño de segmento>)* *(1.2)*

<elemento inferior> ::= *(2)*
<expresión> *(2.1)*

<elemento superior> ::= *(3)*
<expresión> *(3.1)*

<primer elemento> ::= *(4)*
<expresión> *(4.1)*

semántica: Un segmento de matriz proporciona una localización matriz (posiblemente dinámica), que es la parte de la localización matriz especificada indicada por *elemento inferior* y *elemento superior* o *primer elemento entero* y *segmento de matriz*. El **límite inferior** del segmento de matriz es igual al límite inferior de la matriz especificada; el **límite superior** (posiblemente dinámico) se determina a partir de las expresiones especificadas.

propiedades estáticas: El modo asociado a un *segmento de matriz* es un modo matriz **parametrizado** construido como sigue:

&nombre (índice superior)

donde *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo (posiblemente dinámico) de la *localización matriz*, e *índice superior* es una expresión cuya clase es **compatible** con las clases de *elemento inferior* y *elemento superior* y entrega un valor tal que:

NUM (*índice superior*) = *NUM* (*L*) + *NUM* (*elemento superior*) – *NUM* (*elemento inferior*)

o una expresión cuya clase es **compatible** con la clase del *primer elemento*, y entrega un valor tal que:

NUM (*índice superior*) = *NUM* (*L*) + *NUM* (*tamaño de segmento*) – 1

donde *L* es el **límite inferior** del modo de la *localización matriz*.

El modo asociado a un *segmento de matriz* es estático si *índice superior* es **literal**, es decir, *elemento inferior* y *elemento superior* son ambos **literales**, o si *tamaño de segmento* es **literal**; en otro caso, el modo es dinámico.

Un *segmento de matriz* es **referenciable** si la **organización de elementos** del modo de la *localización matriz* es **NOPACK**.

condiciones estáticas: Las clases de *elemento inferior* y *elemento superior*, o la clase de *primer elemento* deben ser **compatibles** con el modo **índice** de la *localización matriz*.

Deben cumplirse las siguientes relaciones:

$L \leq NUM(\text{elemento inferior}) \leq NUM(\text{elemento superior}) \leq U$

$1 \leq NUM(\text{tamaño de segmento}) \leq NUM(U) - NUM(L) + 1$

$NUM(L) \leq NUM(\text{primer elemento}) \leq NUM(\text{primer elemento}) +$
$NUM(\text{tamaño de segmento}) - 1 \leq NUM(\text{U})$

donde *L* y *U* son respectivamente el **límite inferior** y el **límite superior** del modo de la *localización matriz*. Si *U* y el valor de todas las *expresiones* son conocidas estáticamente, las relaciones pueden comprobarse estáticamente.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si falla una parte dinámica de la comprobación de las relaciones antes mencionadas.

ejemplo:

17.27 res (0 : count – 1) *(1.1)*

4.2.10 Campos de estructura

sintaxis:

<campo de estructura> ::= *(1)*
<localización estructura> . <nombre de campo> *(1.1)*

semántica: Un campo de estructura entrega una (sub)localización que es el campo de la localización estructura especificada por *nombre de campo*. Si la *localización estructura* tiene un modo estructura **variable sin marcadores**, y el *nombre de campo* es un nombre **de campo variable**, la semántica se define en la implementación.

propiedades estáticas: El modo del *campo de estructura* es el modo del *nombre de campo*.

Un *campo de estructura* es **referenciable** si **la organización de campo** del *nombre de campo* es **NOPACK**.

condiciones estáticas: El *nombre de campo* debe ser un nombre del conjunto de nombres **de campo** del modo de la *localización estructura*.

condiciones dinámicas: Una *localización* debe denotar:

- una localización de modo estructura **variable con marcadores** en la cual el valor o los valores de campo **marcador** indican que el campo no existe;
- una localización de modo estructura **parametrizada** dinámico en la cual la lista asociada de valores indica que el campo no existe.

Las condiciones mencionadas se denominan condiciones de acceso a campo variable para la localización. Se produce la excepción *TAGFAIL* si dichas condiciones no se cumplen para la *localización estructura*.

ejemplo:

10.57 last –>.info *(1.1)*

4.2.11 Llamadas a procedimiento que entrega una localización

sintaxis:

<llamada a procedimiento que entrega una localización> ::= *(1)*
<llamada a procedimiento que entrega una localización> *(1.1)*

semántica: Una llamada a procedimiento de localización entrega la localización retornada por el procedimiento.

propiedades estáticas: El modo asociado a una *llamada a procedimiento que entrega una localización* es el modo de la **espec de resultado** de la *llamada a procedimiento que entrega una localización*, si no tiene especificado **DYNAMIC**; en otro caso, es la versión del mismo dinámicamente parametrizada que tiene los mismos parámetros que el modo de la localización entregada.

La *llamada a procedimiento que entrega una localización* es **referenciable** si no está especificado **NONREF** en la **espec de resultado** de la *llamada a procedimiento que entrega una localización*.

condiciones dinámicas: La *llamada a procedimiento que entrega una localización* no debe entregar una localización **indefinida**, y el tiempo de vida de la localización entregada no debe haber terminado.

4.2.12 Llamadas a rutina incorporada que entrega una localización

sintaxis:

<llamada a rutina incorporada que entrega una localización> ::= *(1)*
<llamada a rutina incorporada que entrega una localización> *(1.1)*

semántica: Una llamada a rutina incorporada que entrega una localización entrega la localización retornada de la llamada a rutina incorporada.

propiedades estáticas: El modo asociado a la *llamada a rutina incorporada que entrega una localización* es el modo de la **espec de resultado** de la *llamada a rutina incorporada que entrega una localización*.

condiciones dinámicas: La *llamada a rutina incorporada que entrega una localización* no entregará una localización **indefinida**, y el tiempo de vida de la localización entregada no debe haber terminado.

4.2.13 Conversiones de localización

sintaxis:

<conversión de localización> ::= *(1)*
<nombre de modo> (<localización modo estático>) *(1.1)*

semántica: Una conversión de localización entrega la localización denotada por *localización modo estático*. Sin embargo, dicha conversión prevalece sobre las reglas de verificación de modo y de compatibilidad de CHILL, y asocia explícitamente un modo a la localización.

La semántica dinámica precisa de una conversión de localización se define en la implementación.

propiedades estáticas: El modo de una *conversión de localización* es el *nombre de modo*.

Una *conversión de localización* es **referenciable**.

condiciones estáticas: La *localización modo estático* debe ser **referenciable**.

Debe cumplirse la siguiente relación:

SIZE (*nombre de modo*) = *SIZE* (*localización modo estático*)

4.2.14 Localización moreta predefinida

sintaxis:

< localización moreta predefinida> ::= *(1)*
SELF *(1.1)*

semántica: En un procedimiento y/o proceso componente **P** de un *modo moreta*, **SELF** denota la localización moreta ML a la que **P** se está aplicando concurrentemente. El modo de **SELF** es el modo de ML.

condiciones estáticas: La utilización de **SELF** sólo está permitida dentro de la definición de un modo moreta.

5 Valores y operaciones sobre los mismos

5.1 Definiciones de sinónimos

sintaxis:

<sentencia de definición de sinónimo> ::= *(1)*
SYN *<definición de sinónimo>* { , *<definición de sinónimo>* }* ; *(1.1)*

<definición de sinónimo> ::= *(2)*
<lista de ocurrencias de definición> [*<modo>*] = *<valor constante>* *(2.1)*

sintaxis derivada: Una *definición de sinónimo*, en la que la *lista de ocurrencias de definición* consta de más de una *ocurrencia de definición*, se deriva de varias ocurrencias de *definición de sinónimo*, una por cada *ocurrencia de definición* con el mismo *valor constante* y *modo*, si existe. Por ejemplo, **SYN** *i , j = 3;* se deriva de **SYN** *i = 3, j = 3;*.

semántica: Una definición de sinónimo define un nombre que denota el valor **constante** especificado.

propiedades estáticas: Una *ocurrencia de definición* en una *definición de sinónimo* define un nombre de **sinónimo**.

La clase del nombre de **sinónimo**, si se especifica un *modo*, es la clase M-valuada, donde M es el *modo*; en otro caso, es la clase del *valor constante*.

Un nombre de **sinónimo** es **indefinido** si y sólo si el *valor constante* es un valor **indefinido** (véase 5.3.1).

Un nombre **de sinónimo** es **literal** si y sólo si el *valor constante* es **literal**.

condiciones estáticas: Si se especifica un *modo*, éste debe ser **compatible** con la clase del *valor constante*, y el valor entregado por el *valor constante* debe ser uno de los valores definidos por el *modo*.

La evaluación del *valor constante* no debe depender directa ni indirectamente del valor **constante** del nombre de **sinónimo**.

ejemplos:

1.17 **SYN** *neutral_for_add = 0,*

neutral_for_mult = 1; *(1.1)*

2.18 *neutral_for_add fraction* = [*0,1*] *(2.1)*

5.2 Valor primitivo

5.2.1 Generalidades

sintaxis:

<valor primitivo> ::= *(1)*
<contenido de localización> *(1.1)*
| *<nombre de valor>* *(1.2)*
| *<literal>* *(1.3)*
| *<tupla>* *(1.4)*
| *<valor elemento de cadena>* *(1.5)*
| *<valor segmento de cadena>* *(1.6)*
| *<valor elemento de matriz>* *(1.7)*
| *<valor segmento de matriz>* *(1.8)*
| *<valor campo de estructura>* *(1.9)*
| *<conversión de expresión>* *(1.10)*
| *<conversión de representación>* *(1.11)*
| *<llamada a procedimiento que entrega un valor>* *(1.12)*
| *<llamada a rutina incorporada que entrega un valor>* *(1.13)*
| *<expresión arrancar>* *(1.14)*
| *<operador cero-ádico>* *(1.15)*
| *<expresión parentizada>* *(1.16)*

semántica: Un valor primitivo es el componente básico de una expresión. Algunos valores primitivos tienen una clase dinámica, es decir, una clase basada en un modo dinámico. Las verificaciones de compatibilidad para tales valores primitivos sólo pueden completarse en la ejecución. Los fallos de estas verificaciones provocarán la excepción *TAGFAIL* o *RANGEFAIL*.

propiedades estáticas: La clase del *valor primitivo* es la clase del *contenido de localización*, del *nombre de valor*, etc., respectivamente.

Un *valor primitivo* es un *valor primitivo* **constante** si y sólo si es un *nombre de valor* **constante**, un *literal*, una *tupla* **constante**, una *conversión de expresión* **constante**, una *conversión de representación* **constante**, una *llamada a rutina incorporada que tenga un valor* **constante**, o una *expresión parentizada* **constante**.

Un *valor primitivo* es **literal**, si y sólo si es un *nombre de valor* que es **literal**, un *literal* **discreto** o una *llamada a rutina incorporada que entrega un valor* que es **literal**.

5.2.2 Contenido de localización

sintaxis:

<contenido de localización> ::= *(1)*
<localización> *(1.1)*

semántica: Un *contenido de localización* entrega el valor contenido en la localización especificada. Para obtener el valor almacenado se accede a la localización.

propiedades estáticas: La clase de *contenido de localización* es la clase M-valuada, donde M es el modo (posiblemente dinámico) de la *localización*.

condiciones estáticas: El modo de la *localización* no debe tener la **propiedad de no-valor**.

condiciones dinámicas: El valor entregado no debe ser **indefinido** (véase 5.3.1).

ejemplo:

3.7 c2.im (1.1)

5.2.3 Nombres de valor

sintaxis:

<nombre de valor> ::= *(1)*
<nombre de sinónimo> *(1.1)*
| *<nombre de enumeración de valor>* *(1.2)*
| *<nombre de valor hacer-con>* *(1.3)*
| *<nombre de valor a recibir>* *(1.4)*
| *<nombre de procedimiento general>* *(1.5)*

semántica: Un nombre de valor entrega un valor. Un nombre de valor es uno de los siguientes:

- un nombre de **sinónimo**, es decir, un nombre definido en una *sentencia de definición de sinónimo*;
- un nombre de **enumeración de valor**, es decir, un nombre definido por un *contador de bucle* en una *enumeración de valor*;
- un nombre de **valor hacer-con**, es decir, un nombre de **campo** introducido como nombre de valor en la *acción hacer* con una *parte con*;
- un nombre de **valor a recibir**, es decir, un nombre introducido en una *acción recibir y elegir*;
- un nombre de **procedimiento general** (véase 10.4).

Si el valor denotado por un *nombre de valor hacer-con* es un campo variable de un valor de estructura variable sin marcador, la semántica es definida por la implementación.

propiedades estáticas: La clase de un *nombre de valor* es la clase del *nombre de sinónimo*, *nombre de enumeración de valor*, *nombre de valor hacer-con*, *nombre de valor a recibir*, o la clase M-derivada, donde M es el modo del *nombre de procedimiento general*, respectivamente.

Un *nombre de valor* es **literal** si y sólo si es un *nombre de sinónimo* que es **literal**.

Un *nombre de valor* es **constante** si es un *nombre de sinónimo* o un *nombre de procedimiento general* que designa un nombre de **procedimiento** que tiene asociada una *definición de procedimiento* que no está circundada por un bloque.

condiciones estáticas: El *nombre de sinónimo* no debe ser **indefinido**.

condiciones dinámicas: La evaluación de un *nombre de valor hacer-con* produce una excepción *TAGFAIL* si el valor denotado es un campo **variable** y no se cumplen las condiciones de acceso de campo variable para el valor.

ejemplos:

10.12 max (1.1)

8.8 i (1.2)

15.54 this_counter (1.4)

5.2.4 Literales

5.2.4.1 Generalidades

sintaxis:

<literal> ::= *(1)*
<literal entero> *(1.1)*
| *<literal coma flotante>* *(1.2)*
| *<literal booleano>* *(1.3)*
| *<literal de carácter>* *(1.4)*
| *<literal de conjunto>* *(1.5)*
| *<literal de vacío>* *(1.6)*
| *<literal de cadena de caracteres>* *(1.7)*
| *<literal de cadena de bits>* *(1.8)*

semántica: Un literal entrega un valor **constante**.

propiedades estáticas: La clase del *literal* es la clase del *literal entero*, *literal booleano*, etc., respectivamente. Un *literal* es **discreto** si es un *literal entero*, un *literal booleano*, un *literal de carácter* o un *literal de conjunto*.

La letra junto con el apóstrofo siguiente con que comienza un *literal entero*, *literal booleano*, *literal cadena de bits, literal de carácter ancho, o literal de cadena de caracteres anchos* (es decir *B'*, *D'*, *H'*, *O'*, *W'*, *b'*, *d'*, *h'*, *o', w'*) es una calificación de literal.

5.2.4.2 Literales enteros

sintaxis:

<literal entero> ::= *(1)*
<literal entero sin signo> *(1.1)*
| *<literal entero con signo>* *(1.2)*

<literal entero sin signo> ::= *(2)*
<literal entero decimal> *(2.1)*
| *<literal entero binario>* *(2.2)*
| *<literal entero octal>* *(2.3)*
| *<literal entero hexadecimal>* *(2.4)*

<literal entero con signo> ::= *(3)*
– *<literal entero sin signo>* *(3.1)*

<literal entero decimal> ::= *(4)*
[{ D | d } '] *<secuencia de dígitos>* *(4.1)*

<literal entero binario> ::= *(5)*
{ B | b } ' { 0 | 1 | _ }$^{+}$ *(5.1)*

<literal entero octal> ::= *(6)*
{ O | o } ' {*<dígito octal>* | _ }$^{+}$ *(6.1)*

<literal entero hexadecimal> ::= *(7)*
{ H | h>} ' { *<hexadecimal digit>* | _ }$^{+}$ *(7.1)*

<dígito hexadecimal> ::= *(8)*
<dígito> | A | B | C | D | E | F | a | b | c | d | e | f *(8.1)*

<dígito octal> ::= *(9)*
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 *(9.1)*

<secuencia de dígitos> ::= *(10)*
{ *<dígito>* | _ }$^{+}$ *(10.1)*

semántica: Un literal entero entrega un valor entero no negativo. Se ha previsto la notación decimal usual (base 10) así como la binaria (base 2), octal (base 8) y hexadecimal (base 16). El carácter de subrayado (_) no es significativo, es decir, sirve sólo para facilitar la lectura y no influye en el valor denotado.

Un literal entero con signo entrega un valor que es la inversa aditiva del entregado por el *literal entero sin signo* en el mismo.

propiedades estáticas: La clase de un *literal entero* es la clase *&INT*-derivada. Un *literal entero* es **constante** y **literal**.

condiciones estáticas: La cadena que sigue al apóstrofo (') y la *secuencia de dígitos* no deben constar únicamente de caracteres de subrayado.

El valor entregado por *literal entero* tiene que ser uno de los valores definidos en el modo *&INT*.

ejemplos:

6.11 *1_721_119* *(2.1)*
D'1_721_119 *(2.1)*
B'101011_110100 *(2.2)*
O'53_64 *(2.3)*
H'AF4 *(2.4)*

5.2.4.3 Literales coma flotante

sintaxis:

<literal coma flotante> ::= *(1)*
<literal coma flotante sin signo> *(1.1)*
| *<literal coma flotante con signo>* *(1.2)*

<literal coma flotante sin signo> ::= *(2)*
<secuencia de dígitos> . [*<secuencia de dígitos>*] [*<exponente>*] *(2.1)*
[*<secuencia de dígitos>*] . *<secuencia de dígitos>* [*<exponente>*] *(2.2)*

<literal coma flotante con signo> ::= *(3)*
– <literal coma flotante sin signo> *(3.1)*

<exponente> ::= *(4)*
E <secuencia de dígitos> *(4.1)*
E – <secuencia de dígitos> *(4.2)*

sintaxis derivada: Un *literal coma flotante* en el que falta 1. una *secuencia de dígitos*, 2. un *exponente* es sintaxis derivada para un *literal* en que 1. la *secuencia de dígitos* es 0, 2. el *exponente* es E1.

semántica: Un literal coma flotante entrega un valor coma flotante, expresado como un número decimal en notación científica.

Un literal como flotante con signo entrega un valor que es la inversa aditiva del entregado por el *literal coma flotante sin signo* en el mismo.

Si el literal coma flotante está comprendido entre el **límite superior** y el **límite inferior** de uno de los modos coma flotante **predefinidos** de la implementación pero no es representable exactamente, el valor del literal coma flotante es aproximado al valor entregado por una *conversión de representación* implícita al modo coma flotante **predefinido** elegido en la implementación para representar el *literal coma flotante*.

propiedades estáticas: La clase de un *literal coma flotante* es la clase *&FLOAT*-derivada. Un *literal coma flotante* es **constante** y **literal**.

La **precisión** de un *literal coma flotante* es la suma del número de dígitos decimales significativos entregados por las dos *secuencias de dígitos* que forman su mantisa.

condiciones estáticas: El valor entregado por *literal coma flotante* debe ser uno de los valores definidos por el modo *&FLOAT*.

ejemplos:

10,0E1 *(1.1)*

–365,0E–5 *(1.1)*

5.2.4.4 Literales booleanos

sintaxis:

<literal booleano> ::= *(1)*
<nombre de literal booleano> *(1.1)*

nombres predefinidos: Los nombres *FALSE* y *TRUE* son predefinidos como nombres **literales booleanos**.

semántica: Un literal booleano entrega un valor booleano.

propiedades estáticas: La clase de un *literal booleano* es la clase *BOOL*-derivada. Un *literal booleano* es **constante** y **literal**.

ejemplos:

5.42 *FALSE* *(1.1)*

5.2.4.5 Literales de carácter

sintaxis:

<literal de carácter> ::= *(1)*
<literal de carácter estrecho> *(1.1)*
| *<literal de carácter ancho>* *(1.2)*

<literal de carácter estrecho> ::= *(2)*
' { *<carácter>* | *<secuencia de control>*} ' *(2.1)*

<literal de carácter ancho> ::= *(3)*
{ W | w }' { *<carácter>* | *<secuencia de control>*} ' *(3.1)*

<secuencia de control> ::= *(4)*
^ *<expresión literal entera>* { , *< expresión literal entera>* }*) *(4.1)*
| ^ *<carácter no especial>* *(4.2)*
| ^^ *(4.3)*

semántica: Un literal de carácter entrega un valor de carácter.

Aparte de la representación imprimible, puede utilizarse la representación *secuencia de control*. Una secuencia de control en la que el carácter circunflejo (^) va seguido por un paréntesis de apertura denota la secuencia de caracteres cuyas representaciones son la *expresión literal entera* en la misma; en otro caso, si va seguida de otro carácter circunflejo, denota el carácter cuya representación se obtiene por la negación lógica de la posición b7 de la representación interna del carácter *no especial* en la misma (véanse 12.4.4 y apéndice I)

propiedades estáticas: La clase de un *literal de carácter estrecho* es la clase *CHAR*-derivada. La clase de un *literal de carácter ancho* es la clase *WCHAR*-derivada. Un *literal de carácter* es **constante** y **literal**.

condiciones estáticas: Una *secuencia de control* en un *literal de carácter* debe denotar sólo un carácter.

El valor entregado por una *expresión literal entera* en una *secuencia de control* debe pertenecer al intervalo de valores definidos por las representaciones de los caracteres en el juego de caracteres CHILL (véase el apéndice I) en el caso de *literal de carácter estrecho* o al conjunto de valores definidos por las presentaciones de caracteres en los juegos de caracteres de ISO/CEI 10646-1 en el caso de *literal de carácter ancho*.

ejemplo:

7.9 'M' *(2.1)*

5.2.4.6 Literales de conjunto

sintaxis:

<literal de conjunto> ::= *(1)*
[*<nombre de modo>* .] *<nombre de elemento de conjunto>* *(1.1)*

semántica: Un literal de conjunto entrega un valor de conjunto. Un literal de conjunto es un nombre definido en un modo conjunto.

propiedades estáticas: La clase de un *literal de conjunto* es la clase M-valuada, donde M es el *nombre de modo*, si se especifica. En otro caso, M depende del contexto en que ocurre el *literal de conjunto*, de acuerdo con la siguiente exposición:

- si el *literal de conjunto* se utiliza en un lugar en que puede utilizarse una *tupla* sin el *nombre de modo*, M se deriva siguiendo las mismas reglas definidas para las tuplas (véase 5.2.5);
- si el *literal de conjunto* se utiliza como un valor en una *tupla*, M es el modo de ese valor;
- si el *literal de conjunto* se utiliza en un intervalo de literal para definir un *modo intervalo discreto* de la forma:

  <nombre de modo discreto> (*<intervalo de literal>*)

  entonces M es el *nombre de modo discreto*;
- si el *literal de conjunto* es la *expresión de utilización*, la *expresión índice* o la *expresión escribir* en la rutina incorporada para entrada salida (véase 7.4), M es respectivamente *USAGE, WHERE*, el modo **índice** de la *localización de acceso*, el modo **registro** de la *localización de acceso*;
- si el *literal de conjunto* se utiliza en una *expresión condicional*, M se deriva de la misma manera que para la expresión en que está contenida;

- si el *literal de conjunto* es el *índice superior* en un *modo matriz parametrizado*, M es el *modo índice* correspondiente del modo matriz origen;
- si el *literal de conjunto* es una *expresión* en un *modo estructura parametrizada*, M es el modo **raíz** del correspondiente *nombre de campo marcador* en el modo estructura variable origen;
- si el *literal de conjunto* se utiliza en un *elemento de matriz* o *segmento de matriz*, M es el correspondiente *modo índice* en el *modo matriz*;
- si el *literal de conjunto* se utiliza en una *etiqueta de caso*, M se deriva del modo del correspondiente *nombre de campo marcador* (para *modo estructura*), del modo del selector correspondiente en la *lista de selectores de caso* (para *acción de elegir* o *expresión condicional*), o del *modo índice* (para *tupla*);
- si el *literal de conjunto* se utiliza como el *límite inferior* o el *límite superior* y se especifica un nombre *de modo discreto* en el *intervalo de literal* en que está contenido, M es el *nombre de modo discreto*.

Un *literal de conjunto* es **constante** y **literal**.

condiciones estáticas: El *nombre de modo* facultativo puede omitirse solamente en los contextos antes especificados.

El *nombre de elemento de conjunto* debe pertenecer al conjunto de nombre de **elemento de conjunto** de M.

ejemplos:

6.51 dec *(1.1)*

11.78 king *(1.1)*

5.2.4.7 Literal de vacío

sintaxis:

<literal de vacío> ::= *(1)*
<nombre de literal de vacío> ::= *(1.1)*

nombres predefinidos: El nombre *NULL* es predefinido como un nombre **de literal de vacío**.

semántica: El literal de vacío entrega el valor de referencia vacía, es decir, un valor que no se refiere a una localización, el valor de procedimiento vacío, es decir, un valor que no indica un procedimiento, o el valor de ejemplar vacío, es decir, un valor que no identifica un proceso.

propiedades estáticas: La clase del *literal de vacío* es la clase **nula**. Un *literal de vacío* es **constante**.

ejemplo:

10.43 NULL *(1.1)*

5.2.4.8 Literales de cadena de caracteres

sintaxis:

<literal de cadena de caracteres> ::= *(1)*
<literal de cadena de caracteres estrechos> *(1.1)*
| *<literal de cadena de caracteres anchos>* *(1.2)*

<literal de cadena de caracteres estrechos> ::= *(2)*
" { *<carácter no reservado>* | *<comillas>* | *<secuencia de control>*}* " *(2.1)*

<literal de cadena de caracteres anchos> ::= *(3)*
{ W' | w' } " { *<carácter ancho no reservado>* |
<comillas> | *<secuencia de control>*}* " *(3.1)*

<comillas> ::= *(4)*
"" *(4.1)*

semántica: Un literal de cadena de caracteres entrega un valor de cadena de caracteres que puede ser de longitud 0. Éste es una lista de valores para los elementos de la cadena; los valores se dan para los elementos en orden creciente de su índice, de izquierda a derecha. Para representar el carácter comillas (") dentro de un literal de cadena de caracteres, debe escribirse dos veces ("").

propiedades estáticas: La **longitud de cadena** de un *literal de cadena de caracteres* es el número de ocurrencias de *carácter no reservado*, *comillas* y caracteres denotados por ocurrencias de *secuencia de control*.

La clase de un *literal de cadena de caracteres* es la clase **CHARS** (*n*)-derivada, donde *n* es la **longitud de cadena** del *literal de cadena de caracteres estrechos*. La clase de un *literal de cadena de caracteres* es la clase *(n)* derivada **WCHARS**, donde *n* es la **longitud de cadena** del *literal de cadena de caracteres anchos*. Un *literal de cadena de caracteres* es **constante**.

ejemplo:

8.20 "A-B<ZAA9K' " *(2.1)*

5.2.4.9 Literales de cadena de bits

sintaxis:

<literal de cadena de bits> ::= *(1)*
<literal binario de cadena de bits> *(1.1)*
| *<literal octal de cadena de bits>* *(1.2)*
| *<literal hexadecimal de cadena de bits>* *(1.3)*

<literal binario de cadena de bits> ::= *(2)*
{ B | b } ' { 0 | 1 | _ }* ' *(2.1)*

<literal octal de cadena de bits> ::= *(3)*
{ O | o } ' { *<dígito octal>* | _ }* ' *(3.1)*

<literal hexadecimal de cadena de bits> ::= *(4)*
{ H | h } ' { *<dígito hexadecimal>* | _ }* ' *(4.1)*

semántica: Un literal de cadena de bits entrega un valor cadena de bits que puede ser de longitud 0. Pueden emplearse las notaciones binaria, octal o hexadecimal. El carácter de subrayado (_) no es significativo, es decir, sirve sólo para facilitar la lectura y no influye en el valor indicado.

Un literal de cadena de bits es una lista de valores de los elementos de la cadena; los valores de los elementos se dan en orden creciente de su índice, de izquierda a derecha.

propiedades estáticas: La **longitud de cadena** de un *literal de cadena de bits binario*, es el número de ocurrencias *0* y *1* en un *literal de cadena de bits*, o tres veces el número de ocurrencias de *dígito octal* en un *literal de cadena de bits octal*, o cuatro veces el número de ocurrencias de *dígito hexadecimal* en un *literal de cadena de bits hexadecimal*.

La clase de un *literal de cadena de bits* es la clase **BOOLS** *(n)*-derivada, donde *n* es la **longitud de cadena** del *literal de cadena de bits*. Un *literal de cadena de bits* es **constante**.

ejemplos:

B'101011_110100' *(1.1)*

O'53_64' *(1.2)*

H'AF4' *(1.3)*

5.2.5 Tuplas

sintaxis:

<tupla> ::= *(1)*
[*<nombre de modo>*] (: { *<tupla conjuntista>* |
<tupla de matriz> | *<tupla de estructura>* } :) *(1.1)*

<tupla conjuntista> ::= *(2)*
[{ *<expresión>* | *<intervalo>* } { , { *<expresión>* | *<intervalo>* } }*] *(2.1)*

<intervalo> ::= *(3)*
<expresión> : *<expresión>* *(3.1)*

<tupla de matriz> ::= *(4)*
<tupla de matriz no etiquetada> *(4.1)*
| *<tupla de matriz etiquetada>* (4.2)

<tupla de matriz no etiquetada> ::= *(5)*
<valor> { , *<valor>* }* *(5.1)*

<tupla de matriz etiquetada> ::= *(6)*
*<lista de etiquetas de caso> : <valor> { , <lista de etiquetas de caso> : <valor> }** *(6.1)*

<tupla de estructura> ::= *(7)*
<tupla de estructura no etiquetada> *(7.1)*
| *<tupla de estructura etiquetada>* *(7.2)*

<tupla de estructura no etiquetada> ::= *(8)*
*<valor> { , <valor> }** *(8.1)*

<tupla de estructura etiquetada> ::= *(9)*
*<lista de nombres de campo> : <valor> { , <lista de nombres de campo> : <valor> }** *(9.1)*

<lista de nombres de campo> ::= *(10)*
*<nombre de campo> { , . <nombre de campo> }** *(10.1)*

sintaxis derivada: Los corchetes de apertura y cierre, [y], de la tupla son sintaxis derivada para (: y :), respectivamente. Esto no se indica en la sintaxis para evitar confusión con la utilización de corchetes como metasímbolos.

semántica: Una tupla entrega un valor conjuntista, un valor matriz o un valor estructura.

En el caso de un valor conjuntista, éste consta de una lista de expresiones y/o intervalos, que denotan los valores miembros del valor conjuntista. Un intervalo designa aquellos valores comprendidos en el mismo o son uno de los valores entregados por las expresiones en el intervalo. Si la segunda expresión entrega un valor inferior al entregado por la primera, se dice que el intervalo es vacío, es decir, no designa valores. La tupla conjuntista puede designar el valor conjuntista vacío.

Si se trata de un valor matriz, es una lista de valores (posiblemente etiquetada) de los elementos de la matriz; en una tupla de matriz no etiquetada, los valores de los elementos se ordenan según su índice creciente, en la tupla de matriz etiquetada, se dan los valores de los elementos cuyos índices se especifican en la lista de etiquetas de caso que etiqueta al valor. Ésta puede utilizarse como notación abreviada para tuplas de matrices grandes, en las que hay muchos valores iguales. La etiqueta **ELSE** denota todos los valores índices no mencionados explícitamente. La etiqueta * denota todos los valores índice (para más detalles, véase 12.3).

Si se trata de un valor de estructura, es un conjunto (posiblemente etiquetado) para los campos de la estructura. En una tupla de estructura no etiquetada, se indican los valores de los campos en el mismo orden en que están especificados en el modo estructura asociado. En la tupla de estructura etiquetada, se dan los valores de los campos cuyos nombres de campo están especificados en la lista de nombres de campo para el valor.

No está definido el orden de evaluación de las expresiones y valores de una tupla, por lo que puede considerarse que se evalúan en un orden mixto.

propiedades estáticas: La clase de una *tupla* es la clase M-valuada donde M es el *nombre de modo*, si se ha especificado. En otro caso, M depende del contexto en el que aparece la *tupla*, de acuerdo con las normas siguientes:

- si la *tupla* es el *valor* o el *valor constante*, de una *inicialización* en una *declaración de localización*, M es el *modo* de la *declaración de localización*;
- si la *tupla* es el *valor* del lado derecho en una *acción de asignación simple*, M es el modo (posiblemente dinámico) de la *localización* de lado izquierdo;
- si la *tupla* es el *valor constante* de una *definición de sinónimo* con un *modo* especificado, M es dicho *modo*;
- si la *tupla* se utiliza en un *operando-2* y uno de los operandos es **fuerte**, M es el modo del operando **fuerte;**
- si la *tupla* es un *parámetro efectivo* en una *llamada a procedimiento* o en una *expresión arrancar*, donde **DYNAMIC** no está especificado en la correspondiente *espec de parámetro*, entonces M es el modo en la correspondiente *espec de parámetro*;
- si la *tupla* es el *valor* en una *acción retornar* o en una *acción resultar*, M es el modo de la **espec de resultado** del nombre **de procedimiento** de la *acción resultar* o de la *acción retornar* (véase 6.8);
- si la *tupla* es un *valor* de una *acción enviar*, entonces es el modo asociado especificado en la definición de señal del *nombre de señal* o el modo **elemento tampón** del modo de la *localización tampón*;
- si la *tupla* es una *expresión* de una *tupla de matriz*, M es el modo **elemento** del modo de la *tupla de matriz*;

- si la *tupla* es una *expresión* de una *tupla de estructura no etiquetada* o una *tupla de estructura etiquetada* cuya *lista de nombres de campo* asociada contiene solamente un *nombre de campo*, M es el modo de campo de la *tupla de estructura* para la que se especifica la tupla.
- si la *tupla* es el *valor* en una llamada a rutina incorporada *GETSTACK* o *ALLOCATE*, M es el modo denotado por *argumento de modo*.

Una *tupla* es **constante** si y sólo si lo es cada *valor* o *expresión* de la misma.

condiciones estáticas: El *nombre de modo* facultativo puede suprimirse sólo en los textos especificados anteriormente. Según se especifique una *tupla conjuntista*, una *tupla de matriz* o una *tupla de estructura*, deberán cumplirse las siguientes condiciones de compatibilidad:

a) *Tupla conjuntista*

1) El modo de la *tupla* debe ser un modo conjuntista.

2) La clase de cada *expresión* debe ser **compatible** con el modo **miembro** del modo de la *tupla*.

3) Para una tupla conjuntista **constante**, el valor entregado por cada *expresión* debe ser uno de los valores definidos por ese modo **miembro**.

b) *Tupla matricial*

1) El modo de la *tupla* debe ser un modo matriz.

2) La clase de cada *valor* debe ser **compatible** con el modo **elemento** del modo de la *tupla*.

3) En el caso de una *tupla de matriz no etiquetada*, deben producirse tantas ocurrencias de *valor* como **número de elementos** tenga el modo matriz de la *tupla*.

4) En el caso de una *tupla de matriz etiquetada*, deben cumplirse las condiciones de selección de caso para la lista de ocurrencias de *lista de etiquetas de caso* (véase 12.3). La **clase resultante** debe ser **compatible** con el modo **índice** del modo de la *tupla*. La lista de especificaciones de etiquetas de caso debe estar **completa**.

5) En el caso de una *tupla de matriz etiquetada*, los valores indicados explícitamente por cada etiqueta de caso en una *lista de etiquetas de caso* tienen que ser valores definidos por el modo **índice** de la *tupla*.

6) En una *tupla de matriz no etiquetada*, al menos una ocurrencia de *valor* debe ser una *expresión*.

7) Para una *tupla de matriz* **constante**, en la que el modo **elemento** del modo de la *tupla* es un modo discreto, cada *valor* especificado debe entregar un valor definido por ese modo **elemento**, a menos que se trate de un valor **indefinido**.

c) *Tupla de estructura*

1) El modo de la tupla debe ser un modo estructura.

2) Este modo no debe ser un modo estructura que tenga nombres de **campo** que sean **invisibles** (véase 12.2.5).

En el caso de una tupla de estructura no etiquetada:

– Si el modo de la *tupla* no es un modo estructura **variable** ni un modo estructura **parametrizada**:

3) Deben producirse tantas ocurrencias de *valor* como nombres de **campo** haya en la lista de nombres de **campo** del modo de la *tupla*.

4) La clase de cada *valor* debe ser **compatible** con el modo del nombre de **campo** correspondiente (en posición) al modo de la *tupla*.

– Si el modo de la *tupla* es un modo estructura **variable con marcadores** o un modo estructura **parametrizada con marcadores**:

5) Cada *valor* especificado por un campo **marcador** debe ser una *expresión literal discreta*.

6) Debe haber tantas ocurrencias de *valor* como nombres de **campo** haya indicados como existentes por el valor o valores entregados por las ocurrencias de *expresión literal discreta* especificadas para los campos **marcadores**.

7) La clase de cada *valor* debe ser **compatible** con el modo del nombre de **campo** correspondiente.

– Si el modo de la *tupla* es un modo estructura **variable sin marcadores** o un modo estructura **parametrizada sin marcador(es)**:

8) No se apermite ninguna tupla de estructura no etiquetada.

En el caso de una tupla de estructura etiquetada:

– Si el modo de la *tupla* no es ni un modo estructura **variable** ni un modo estructura **parametrizada**:

  9) Cada nombre de **campo** de la lista de nombres de **campo** del modo de la *tupla* debe mencionarse una vez, y solo una en la *tupla*.

  10) La clase de cada *valor* debe ser **compatible** con el modo de cada nombre de **campo** especificado en la *lista de nombres de campo* que etiqueta ese *valor*. Los modos de todos los nombres de **campo** en *la lista de nombre de campo* deben ser **equivalentes**.

– Si el modo de la *tupla* es un modo estructura **variable con marcadores** o un modo estructura **parametrizada con marcadores**:

  11) Cada *valor* especificado para un campo **marcador** debe ser una *expresión literal discreta*.

  12) Cada nombre de **campo** que denote un campo fijo o un campo indicado como existente por el valor o valores entregados por las ocurrencias de *expresión literal discreta* especificadas para los campos **marcadores** deben mencionarse una vez y sólo una vez en la *tupla*.

  13) La clase de cada *valor* debe ser **compatible** con el modo de cualquier nombre de **campo** especificado en la *lista de nombres de campo* que etiquetan ese *valor*.

– Si el modo de la *tupla* es un modo estructura **variable sin marcadores** o un modo estructura **parametrizada sin marcadores**:

  14) Cada nombre de **campo** debe mencionarse como máximo una vez en la tupla. Todos los nombres de **campos fijos** deben ser mencionados. Los nombres de **campo** mencionados en la tupla, que están definidos en el mismo *campo alternativo*, deben definirse todos en la misma alternativa variable, o después de **ELSE**. Todos los nombres de **campo** de un campo alternativo en cada alternativa variable o todos los nombres de **campo** definidos después de **ELSE** deben mencionarse.

  15) La clase de cada *valor* debe ser **compatible** con el modo de cualquier nombre de **campo** especificado en la *lista de nombres de campo* que etiqueta ese *valor*.

  16) Si el modo de una *tupla* es un modo estructura **parametrizada con marcadores**, la lista de valores entregados por las ocurrencias de *expresión literal discreta* especificadas para los campos **marcadores** debe ser la misma que la lista de valores del modo de la *tupla*.

  17) Para una *tupla* de estructura **constante**, cada **valor** especificado para un campo con un modo discreto debe entregar un valor definido por el modo **campo**, a menos que se trate de un valor **indefinido**.

  18) Al menos una ocurrencia de *valor* debe ser una *expresión*.

Una *tupla* no puede tener dos ocurrencias de *valor*, de manera que una sea **extrarregional** y la otra **intrarregional** (véase 11.2.2).

condiciones dinámicas: En caso de *tupla conjuntista*, *tupla de matriz* o *tupla de estructura* se aplican las condiciones de asignación de cualquier valor con respecto al modo **miembro**, modo **elemento** o modo **campo** asociado, respectivamente (véase 6.2) (véanse las condiciones a2, b2, c4, c7, c10, c13 y c15).

Si la *tupla* tiene un modo matriz dinámico, se produce la excepción *RANGEFAIL* si no se cumple una cualquiera de las condiciones b3 o b5.

Si la *tupla* tiene un modo estructura **parametrizada** dinámico, se produce la excepción *TAGFAIL* si no se cumple una cualquiera de las condiciones c14 o c16.

El valor entregado por una *tupla* no debe ser **indefinido**.

ejemplos:

9.6 *number_list [ ]^* *(1.1)*

9.7 *[2:max]* *(2.1)*

8.26 *[('A'):3,('B','K','Z'):1,(**ELSE**):0]* *(6.1)*

17.5 *[(*):' ']* *(6.1)*

12.35 *(:NULL,NULL,536:)* *(7.1)*

11.18 *[.status:occupied,.p:[white,rook]]* *(9.1)*

5.2.6 Valores elemento de cadena

sintaxis:

<valor elemento de cadena> ::= *(1)*
<valor primitivo de cadena> (<elemento de arranque>) *(1.1)*

NOTA – Si el *valor primitivo de cadena* es una *localización cadena*, la construcción sintáctica es ambigua y se interpretará como un *elemento de cadena* (véase 4.2.6).

semántica: Un valor elemento de cadena entrega un valor que es el elemento del valor de cadena especificado indicado por *elemento de arranque*.

propiedades estáticas: La clase del *valor elemento de cadena* es la clase M-valuada, siendo M el modo **elemento** del modo del *valor primitivo de cadena*.

Un *valor elemento de cadena* es **constante** si y sólo si el *valor primitivo de cadena* y el *elemento de arranque* son **constantes**.

condiciones dinámicas: El valor entregado por un *valor elemento de cadena* no debe ser **indefinido**.

Se produce la excepción *RANGEFAIL* si no se cumple la relación siguiente:

$$0 \leq NUM(\textit{elemento de arranque}) \leq L - 1$$

donde L es la **longitud efectiva** del *valor primitivo de cadena*.

5.2.7 Valores segmento de cadena

sintaxis:

<valor segmento de cadena> ::= *(1)*
<valor primitivo de cadena> (<elemento de la izquierda> : <elemento de la derecha>) *(1.1)*
| *<valor primitivo de cadena> (<elemento de arranque>* **UP** *<tamaño de segmento>)* *(1.2)*

NOTA – Si el *valor primitivo de cadena* es una *localización cadena*, la construcción sintáctica es ambigua y se interpretará como un *segmento de cadena* (véase 4.2.7).

semántica: Un valor segmento de cadena entrega un valor cadena (posiblemente dinámico) que es la parte del valor de cadena especificado indicado por *elemento de la izquierda* y *elemento de la derecha,* o *elemento de arranque* y *tamaño de segmento*. La longitud (posiblemente dinámica) del segmento de cadena se determina a partir de las expresiones especificadas.

Un *segmento de cadena* en el que el *elemento de la derecha* entrega un valor inferior al que entrega el *elemento de la izquierda*, o en el que *tamaño de segmento* entrega un valor no positivo, denota una cadena vacía.

propiedades estáticas: La clase (posiblemente dinámica) de un *valor segmento de cadena* es la clase M-valuada si el *valor primitivo de cadena* es **fuerte**, y en otro caso la clase M-derivada, donde M es un modo cadena **parametrizado** construido como sigue:

&nombre (*longitud de cadena*)

donde *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo **raíz** (posiblemente dinámico) del valor primitivo *de cadena* si es un modo cadena **fijo**, y otro caso del modo **componente**, y en el cual *el tamaño de cadena* es

$$NUM(\textit{elemento de la derecha}) - NUM(\textit{elemento de la izquierda}) + 1$$

o

$$NUM(\textit{tamaño de segmento}).$$

Sin embargo, si se denota una cadena vacía, *tamaño de segmento* es 0. La clase de un *valor segmento de cadena* es estática si *tamaño de cadena* es **literal**, es decir, *elemento de la izquierda* y *elemento de la derecha* son **literales** o *tamaño de segmento* es **literal**; en otro caso, la clase es dinámica.

Un *valor segmento de cadena* es **constante** si y sólo si el *valor primitivo de cadena* y el *tamaño de cadena* son **constantes**.

condiciones estáticas: Deben cumplirse las siguientes relaciones:

$0 \leq NUM$ (*elemento de la izquierda*) $\leq L - 1$

$0 \leq NUM$ (*elemento de la derecha*) $\leq L - 1$

$0 \leq NUM$ (*elemento de arranque*) $\leq L - 1$

NUM (*elemento de arranque*) + NUM (*tamaño de segmento*) $\leq L$

donde *L* es la **longitud efectiva** del *valor primitivo de cadena*. Si *L* y el valor de todas las *expresiones enteras* son conocidas estáticamente, las relaciones pueden comprobarse estáticamente.

condiciones dinámicas: El valor entregado por un *valor segmento de cadena* no debe ser **indefinido**.

Se produce la excepción *RANGEFAIL* si falla una parte dinámica de la verificación de las relaciones mencionadas.

5.2.8 Valores elemento de matriz

sintaxis:

<valor elemento de matriz> ::= *(1)*
<valor primitivo de matriz> (*<lista de expresiones>*) *(1.1)*

NOTA – Si el *valor primitivo de matriz* es una *localización matriz*, la construcción sintáctica es ambigua y se interpretará como un *elemento de matriz* (véase 4.2.8).

sintaxis derivada: Véase 4.2.8.

semántica: Un valor elemento de matriz entrega un valor que es el elemento del valor de matriz especificado indicado por *expresión*.

propiedades estáticas: La clase del *valor elemento de matriz* es la clase M-valuada, donde M es el modo **elemento** del modo del *valor primitivo de matriz*.

Un *valor elemento de matriz* es **constante** si y sólo si *valor primitivo de matriz* y *expresión* son **constantes**.

condiciones estáticas: La clase de la *expresión* debe ser **compatible** con el modo **índice** del modo del *valor primitivo de matriz*.

condiciones dinámicas: El valor entregado por un *valor elemento de matriz* no debe ser **indefinido**.

Se produce la excepción *RANGEFAIL* si no se cumple la siguiente relación:

$L \leq$ *expresión* $\leq U$

donde *L* y *U* son respectivamente el **límite inferior** y el **límite superior** (posiblemente dinámico) del modo del *valor primitivo de matriz*.

5.2.9 Valores segmento de matriz

sintaxis:

<valor segmento de matriz> ::= *(1)*
<valor primitivo de matriz> (*<elemento inferior>* : *<elemento superior>*) *(1.1)*
| *<valor primitivo de matriz>* (*<primer elemento>* **UP** *<tamaño de segmento>*) *(1.2)*

NOTA – Si el *valor primitivo de matriz* es una *localización matriz*, la construcción sintáctica es ambigua y se interpretará como un *segmento de matriz* (véase 4.2.9).

semántica: Un valor segmento de matriz entrega un valor matriz (posiblemente dinámico), que es la parte del valor matriz especificado indicado por *elemento inferior* y *elemento superior*, o *primer elemento* y *tamaño de segmento*. El **límite inferior** del valor segmento de matriz es igual al **límite inferior** del valor matriz especificado; el **límite superior** (posiblemente dinámico) se determina a partir de las expresiones especificadas.

propiedades estáticas: La clase (posiblemente dinámica) de un *valor segmento de matriz* es la clase M-valuada, donde M es un modo matriz **parametrizado** construido como sigue:

&nombre (*índice superior*)

donde *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo (posiblemente dinámico) del valor *primitivo de matriz* e *índice superior* es una expresión cuya clase es **compatible** con las clases de *elemento inferior* y *elemento superior*, y entrega un valor tal que:

NUM (*índice superior*) = *NUM* (*L*) + *NUM* (*elemento superior*) – *NUM* (*elemento inferior*)

o es una expresión cuya clase es **compatible** con la clase de *primer elemento* y entrega un valor tal que:

NUM (*índice superior*) = *NUM* (*L*) + *NUM* (*tamaño de segmento*) – 1

donde *L* es el **límite inferior** del modo del *valor primitivo de matriz*.

La clase de un *valor segmento de matriz* es estática si *índice superior* es **literal**: es decir, *elemento inferior* y *elemento superior* son ambos **literales** o *tamaño de segmento* es **literal**; en otro caso, la clase es dinámica.

condiciones estáticas: Las clases de *elemento superior* y *elemento inferior* o la clase de *primer elemento* deben ser **compatibles** con el modo **índice** del *valor primitivo de matriz*.

Deben cumplirse las siguientes relaciones:

$$L \leq NUM(\textit{elemento inferior}) \leq NUM(\textit{elemento superior}) \leq U$$

$$1 \leq NUM(\textit{tamaño de segmento}) \leq NUM(U) - NUM(L) + 1$$

$$NUM(L) \leq NUM(\textit{primer elemento}) \leq NUM(\textit{primer elemento}) + NUM(\textit{tamaño de segmento}) - 1 \leq NUM(U)$$

donde *L* y *U* son respectivamente el **límite inferior** y el **límite superior** del modo del *valor primitivo de matriz*. Si *U* y el valor de todas las *expresiones* son conocidos estáticamente, las relaciones pueden comprobarse estáticamente.

Un *valor segmento de matriz* es **constante** si y sólo si *valor primitivo de matriz* e *índice superior* son **constantes**.

condiciones dinámicas: El valor entregado por un *valor segmento de matriz* no debe ser **indefinido**.

Se produce la excepción *RANGEFAIL* si falla una parte dinámica de la comprobación de las relaciones mencionadas.

5.2.10 Valores campo de estructura

sintaxis:

<valor campo de estructura> ::= *(1)*
<valor primitivo de estructura> . <nombre de campo> *(1.1)*

NOTA – Si el *valor primitivo de estructura* es una *localización estructura*, la construcción sintáctica es ambigua y se interpretará como un *campo de estructura* (véase 4.2.10).

semántica: Un valor campo de estructura entrega un valor, que es el campo del valor de estructura especificado indicado por *nombre de campo*. Si el *valor primitivo de estructura* tiene un modo estructura **variable sin marcadores** y el *nombre de campo* es un nombre de **campo variable**, la semántica se define en la implementación.

propiedades estáticas: La clase del *valor campo de estructura* es la clase M-valuada, donde M es el modo del *nombre de campo*.

Un *valor campo de estructura* es **constante** si y sólo si *valor primitivo de estructura* es **constante**.

condiciones estáticas: El *nombre de campo* debe ser un nombre perteneciente al conjunto de nombres de **campo** del modo del *valor primitivo de estructura*.

condiciones dinámicas: El valor entregado por un *valor campo de estructura* no debe ser **indefinido**.

Un valor no debe denotar:

- un modo estructura **variable con marcadores**, en el que el valor o valores de campo **marcador** indican que el campo denotado no existe;
- un modo estructura **parametrizada** dinámico, en el que la lista de valores asociada indica que el campo no existe.

Estas condiciones se denominan las condiciones de acceso de campo variable para el valor (obsérvese que la condición no incluye la ocurrencia de una excepción). Se produce la excepción *TAGFAIL* si las condiciones mencionadas no se cumplen para el *valor primitivo de estructura*.

ejemplo:

11.140 b (lin)(col).status *(1.1)*

5.2.11 Conversión de expresión

sintaxis:

<conversión de expresión> ::= *(1)*
<nombre de modo> # (<expresión>) *(1.1)*

NOTA – Si la *expresión* es una *localización modo estático*, la construcción sintáctica es ambigua y se interpretará como una *conversión de localización* (véase 4.2.13).

semántica: Una conversión de expresión prevalece sobre la verificación de modos y las reglas de compatibilidad de CHILL. Esta conversión asocia explícitamente un modo a la expresión sin cambiar la representación interna.

propiedades estáticas: La clase de la *conversión de expresión* es la clase M-valuada, donde M es el *nombre de modo*. Una *conversión de expresión* es **constante** si y sólo si lo la *expresión* es *constante*.

condiciones estáticas: El *nombre de modo* no puede tener la **propiedad de no-valor**. El tamaño del modo **raíz** de la *expresión* y el tamaño del *nombre de modo* deben ser iguales.

5.2.12 Conversión de representación

sintaxis:

<representación de conversión>::= *(1)*
<nombre de modo> (<expresión>) *(1.1)*

semántica: Una conversión de representación prevalece sobre las reglas de verificación de modo y de compatibilidad de CHILL. Asocia un modo a la expresión y puede cambiar la representación interna del valor entregado por la propia expresión. Si el modo del *nombre de modo* es un modo discreto y la clase del valor entregado por la expresión es discreta, el valor entregado por la conversión de representación es tal que:

NUM (*nombre de modo* (*expresión*)) = *NUM* (*expresión*)

Una conversión de representación en la que *nombre de modo* y el modo **raíz** de la clase de la expresión son respectivamente:

- un modo entero y un modo coma flotante;
- un modo coma flotante y un modo entero;
- un modo coma flotante y otro modo coma flotante con modos **raíz** diferentes,

puede implicar una aproximación. Si el valor entregado por la expresión es exactamente representable en el conjunto de valores de *nombre de modo*, el resultado de la conversión de representación es el valor de la propia expresión, y en otro caso es uno de los dos valores pertenecientes al conjunto de valores de *nombre de modo* que delimitan el intervalo más pequeño en que está contenido el valor entregado por la expresión. Una conversión de representación en la que *nombre de modo* es un modo entero y el modo **raíz** de la clase de la expresión es un modo duración, entrega un valor entero que representa en milisegundos el valor entregado por la expresión.

Una conversión de representación en la que el *nombre de modo* o el modo **raíz** de la clase de la expresión es un modo estructura, y el otro es un modo estructura **parametrizada** cuyo modo estructura de **origen** es **similar** a ésta, entrega un valor de estructura en el cual los valores de los campos son iguales a los correspondientes de la expresión, si están presentes. En otro caso, el resultado se define en la implementación. Obsérvese que para valores de estructura **variable sin marcadores** y para valores de estructura **variable con marcadores** en la que la lista de valores de marcadores es diferente de la del modo estructura **parametrizada**, el resultado de la conversión de representación se define en la implementación.

En una conversión de representación en la que el modo M del *nombre de modo* es un modo referencia y la clase de la expresión es la clase **nulo**, y el resultado de la conversión de representación es **nulo** si M es **compatible** con la clase de –> ((*expresión*) –>), entonces el resultado es igual al mismo; de lo contrario, el resultado está definido por la implementación.

En otro caso, el valor entregado por la conversión de representación está definido por la implementación y puede depender de la representación interna de los valores.

propiedades estáticas: La clase de la *conversión de representación* es la clase M-valuada, donde M es el *nombre de modo*. Una *conversión de representación* es **constante** si y sólo si la *expresión* es **constante**.

condiciones estáticas: El *nombre de modo* no debe tener la **propiedad de no-valor**. Una implementación puede imponer condiciones estáticas adicionales.

condiciones dinámicas: En caso de una *expresión* que no sea **constante**:

- se produce una excepción *RANGEFAIL* si *nombre de modo* es un modo duración y el modo **raíz** de la clase de la *expresión* es un modo entero (o viceversa) y el valor entregado por *conversión de representación* no pertenece al conjunto de valores definido por *nombre de modo*;
- se produce una excepción *OVERFLOW* si:
  - la clase del valor entregado por *expresión* es discreto y el modo de *nombre de modo* es un modo discreto que no define un valor con una representación interna igual a *NUM* (*expresión*);
  - el modo de *nombre de modo* y el modo **raíz** de la clase de la *expresión* son, independientemente, un modo entero o un modo coma flotante, y la *expresión* entrega un valor que no está comprendido entre los límites del modo **raíz** de *nombre de modo*;
- se produce una excepción *UNDERFLOW* si el *nombre de modo* y el modo **raíz** de la clase de la *expresión* son modos coma flotante, y el valor entregado por la *expresión* es mayor que el **límite inferior negativo** y menor que el **límite inferior positivo** del *nombre de modo*, y es diferente de cero.

Una implementación puede imponer condiciones dinámicas adicionales que, si son incumplidas, producen una excepción definida por la implementación.

5.2.13 Llamada a procedimiento que entrega un valor

sintaxis:

<llamada a procedimiento que entrega un valor> ::= *(1)*
<llamada a procedimiento que entrega un valor> *(1.1)*

semántica: Una llamada a procedimiento que entrega un valor entrega el valor retornado por un procedimiento.

propiedades estáticas: La clase de la *llamada a procedimiento que entrega un valor* es la clase M-valuada, donde M es el modo de la **espec de resultado** de la *llamada a procedimiento que entrega un valor*.

condiciones dinámicas: La *llamada a procedimiento que entrega un valor* no debe entregar un valor **indefinido** (véanse 5.3.1 y 6.8).

ejemplos:

6.50 *julian_day_number([10,dec,1979])* *(1.1)*

11.63 *ok_bishop(b,m)* *(1.1)*

5.2.14 Llamada a una rutina incorporada que entrega un valor

sintaxis:

<llamada a rutina incorporada que entrega un valor>::= *(1)*
<llamada a una rutina incorporada que entrega un valor> *(1.1)*

semántica: Una llamada a rutina incorporada entrega el valor retornado por la rutina incorporada.

propiedades estáticas: La clase asociada a la *llamada a rutina incorporada que entrega un valor* es la clase de la *llamada a rutina incorporada que entrega un valor*.

condiciones dinámicas: La *llamada a rutina incorporada que entrega un valor* no debe entregar un valor **indefinido** (véanse 5.3.1 y 6.8).

5.2.15 Expresiones arrancar

sintaxis:

<expresión arrancar> ::= *(1)*
START *<nombre de proceso>* ([*<lista de parámetros efectivos>*]) *(1.1)*

semántica: La evaluación de la expresión arrancar crea y activa un nuevo proceso cuya definición se indica mediante el *nombre de proceso* (véase la cláusula 11). La expresión arrancar entrega el valor de ejemplar que identifica el proceso creado. La transferencia de parámetros es similar a la transferencia de parámetros de procedimiento; sin embargo, pueden darse parámetros efectivos adicionales con un significado definido por la implementación.

propiedades estáticas: La clase de una *expresión arrancar* es la clase *INSTANCE*-derivada.

condiciones estáticas: El número de ocurrencias de *parámetro efectivo* en la *lista de parámetros efectivos* no debe ser inferior al número de ocurrencias de *parámetro formal* en la *lista de parámetros formales* de la definición de proceso del

nombre de proceso. Si el número de parámetros efectivos es *m* y el de parámetros formales es *n* ($m \geq n$), los requisitos de compatibilidad y de **regionalidad** para los *n* primeros parámetros efectivos son los mismos que para la transferencia de parámetros de procedimiento (véase 6.7). Las condiciones estáticas para el resto de los parámetros efectivos son definidos por la implementación.

condiciones dinámicas: Para la transferencia de parámetros se aplican las condiciones de asignación de cualquier valor efectivo con relación al modo de su parámetro formal asociado (véase 6.7).

La expresión *arrancar* provoca la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplo:

15.35 **START** *counter()* *(1.1)*

5.2.16 Operador cero-ádico

sintaxis:

<operador cero-ádico> ::= *(1)*
THIS *(1.1)*

semántica: El operador cero-ádico entrega el valor de ejemplar único que identifica al proceso que lo ejecuta. Si es ejecutado por una localización de tarea se produce una excepción *THIS_FAIL*.

propiedades estáticas: La clase del *operador cero-ádico* es la clase *INSTANCE*-derivada.

condiciones estáticas: El *operador cero-ádico* **THIS** no debe aparecer dentro de una definición de *modo tarea*.

5.2.17 Expresión parentizada

sintaxis:

<expresión parentizada> ::= *(1)*
(*<expresión>*) *(1.1)*

semántica: Una expresión parentizada entrega el valor entregado por la evaluación de la expresión.

propiedades estáticas: La clase de la *expresión parentizada* es la clase de la *expresión*.

Una *expresión parentizada* es **constante** (**literal**) si y sólo si la *expresión* es **constante** (**literal**).

ejemplo:

5.10 *(a1* ***OR*** *b1)* *(1.1)*

5.3 Valores y expresiones

5.3.1 Generalidades

sintaxis:

<valor> ::= *(1)*
<expresión> *(1.1)*
| *<valor indefinido>* *(1.2)*

<valor indefinido> ::= *(2)*
* *(2.1)*
| *<nombre de sinónimo indefinido>* *(2.2)*

semántica: Un valor es un valor **indefinido** o un valor (definido en CHILL) entregado como resultado de la evaluación de una expresión.

Salvo cuando se indique explícitamente lo contrario, el orden de evaluación de los constituyentes de una expresión y de sus subconstituyentes, etc., es indefinido, y puede considerarse que se evalúan en orden mixto. Sólo necesitan ser evaluados hasta el punto en que el valor que ha de entregarse se determine unívocamente. Si el contexto requiere una expresión **constante** o **literal**, se supone que la evaluación se efectúa antes de la ejecución y no puede provocar una excepción. Una implementación definirá gamas de valores autorizados para expresiones **literales** y **constantes**, y puede rechazar un programa si esa evaluación anterior a la ejecución entrega un valor fuera de los límites definidos por la implementación.

propiedades estáticas: La clase de un *valor* es la clase de la *expresión* o *valor indefinido*, respectivamente.

La clase del *valor indefinido* es la clase **general** si el *valor indefinido* es un ∗; en otro caso, esta clase es la del *nombre de sinónimo indefinido*.

Un *valor* es **constante** si y sólo si es un *valor indefinido* o una *expresión* que es **constante**. Un *valor* es **literal** si y sólo si es una *expresión* que es **literal**.

propiedades dinámicas: Se dice que un valor es **indefinido** si está denotado por el *valor indefinido* o si se indica explícitamente en esta Recomendación. Un valor compuesto es **indefinido** si y sólo si todos sus subcomponentes (es decir valores subcadena, valores elemento y valores campo), son **indefinidos**.

ejemplo:

6.40 *(146_097∗c)/4+(1_461∗y)/4*
*+(153*m+2)/5+day+1_721_119* *(1.1)*

5.3.2 Expresiones

sintaxis:

<expresión> ::= *(1)*
    <operando–0> *(1.1)*
  | *<expresión condicional>* *(1.2)*

<expresión condicional> ::= *(2)*
  | **IF** *<expresión booleana>* *<alternativa entonces>*
    <alternativa si no> **FI** *(2.1)*
  | **CASE** *<lista de selectores de caso>* **OF** { *<alternativa de elegir valor>* }$^{+}$
    [**ELSE** *<subexpresión>*] **ESAC** *(2.2)*

<alternativa entonces> ::= *(3)*
    THEN *<subexpresión>* *(3.1)*

<alternativa si no> ::= *(4)*
    ELSE *<subexpresión>* *(4.1)*
  | **ELSIF** *<expresión booleana>* *<alternativa entonces>* *<alternativa si no>* *(4.2)*

<subexpresión> ::= *(5)*
    <expresión> *(5.1)*

<alternativa de elegir valor> ::= *(6)*
    <especificación de etiqueta de caso> : *<subexpresión>* ; *(6.1)*

semántica: Si **IF** está especificado, la *expresión booleana* es evaluada, y si da *TRUE*, el resultado es el valor entregado por la *subexpresión* en la *alternativa entonces*; en otro caso es el valor entregado por la *alternativa si no*.

El valor entregado por una *alternativa si no* es el valor de la *subexpresión* si **ELSE** está especificado; en otro caso la *expresión booleana* es evaluada, y si da *TRUE*, es el valor entregado por la *subexpresión* en la *alternativa entonces*, y en otro caso el entregado por la *alternativa si no*.

Si se especifica **CASE** se evalúan las *subexpresiones* en la *lista de selectores de caso*, y si concuerda una *especificación etiqueta de caso*, el resultado es el valor entregado por la *subexpresión* correspondiente, y en otro caso el entregado por la *subexpresión* que sigue a **ELSE** (que estará presente).

Las *subexpresiones* no empleadas en una *expresión condicional* no se evalúan.

propiedades estáticas: Si una *expresión* es un *operando–0*, la clase de la *expresión* es la clase del *operando–0*. Si es una *expresión condicional*, la clase de la *expresión* es la clase M-valuada, donde M es el modo que depende del contexto en el que ocurre la *expresión condicional*, de acuerdo con las mismas reglas que definen el modo de la clase de una tupla sin un *nombre de modo* (véase 5.2.5).

Una expresión es **constante** (**literal**) si y sólo si es un *operando–0* que es **constante** (**literal**), o una *expresión condicional* en la que todas las *expresiones booleanas* o *listas de selectores de caso* contenidas en ella son **constantes** (**literales**), y en la que todas las *subexpresiones* son **constantes** (**literales**).

condiciones estáticas: Si una *expresión* es una *expresión condicional*, se aplican las condiciones siguientes:

- una *expresión condicional* puede ocurrir únicamente en los contextos en que puede ocurrir una tupla sin un *nombre de modo* delante de ella;

- cada *subexpresión* debe ser **compatible** con el modo que se ha derivado del contexto con las mismas reglas empleadas para las tuplas. Sin embargo, la parte dinámica de la relación de compatibilidad se aplica únicamente a la *subexpresión* seleccionada;
- si se especifica **CASE**, deben cumplirse las condiciones de selección de caso (véase 12.3), y los mismos requisitos de completud, consistencia y compatibilidad que para la acción de caso (acción de elegir) (véase 6.4);
- ninguna *expresión condicional* puede contener dos ocurrencias de *subexpresión*, tales que una sea **extrarregional** y la otra **intrarregional** (véase 11.2.2).

condiciones dinámicas: En el caso de una *expresión condicional*, se aplican las condiciones de asignación del valor entregado por la *subexpresión* seleccionada con respecto al modo M derivado del contexto.

5.3.3 Operando–0

sintaxis:

<operando–0> ::= *(1)*
<operando–1> *(1.1)*
| *<suboperando–0>* { **OR** | **ORIF** | **XOR** } *<operando–1>* *(1.2)*

<suboperando–0> ::= *(2)*
<operando–0> *(2.1)*

semántica: Si se especifica **OR**, **ORIF** o **XOR**, el *suboperando–0* y el *operando–1* entregan:

- valores booleanos, en cuyo caso **OR** y **XOR** denotan los operadores lógicos "disyunción inclusiva" y "disyunción exclusiva", respectivamente, que entregan un valor booleano. Si se especifica **ORIF** y el *operando–0* entrega el valor booleano *TRUE*, el resultado es este valor; en otro caso, el resultado es el *operando–1*;
- valores de cadena de bits, en cuyo caso **OR** y **XOR** denotan las operaciones lógicas sobre cada elemento de las cadenas de bits, y entregan un valor de cadena de bits;
- valores conjuntistas, en cuyo caso **OR** denota la unión de varios valores conjuntistas y **XOR** denota el valor conjuntista compuesto por los valores miembros que están solamente en uno de los valores conjuntistas especificados (por ejemplo, *A* **XOR** *B* = *A* – *B* **OR** *B* – *A*).

propiedades estáticas: Si un *operando–0* es un *operando–1*, la clase del *operando–0* es la clase del *operando–1*. Si se especifica **OR**, **ORIF** o **XOR**, la clase del *operando–0* es la **clase resultante** de las clases del *suboperando–0* y el *operando–1*.

Un *operando–0* es **constante** (**literal**) si y sólo si es un *operando–1* que es **constante** (**literal**), o está formado por un *operando–0* y un *operando–1* que son ambos **constantes** (**literales**).

condiciones estáticas: Si se especifica **OR**, **ORIF** o **XOR**, la clase del *suboperando–0* debe ser **compatible** con la clase del *operando–1*. Si se especifica **ORIF**, ambas clases deben tener un modo **raíz** booleano; en otro caso, ambas clases deben tener un modo **raíz** booleano, conjuntista o de cadena de bits, en cuyo caso la **longitud efectiva** del *sub-operando-0* y del *operando–1* debe ser la misma. Esta verificación es dinámica si uno o ambos modos son dinámicos o de cadena **variable**.

condiciones dinámicas: En el caso de **OR** o **XOR**, se produce una excepción *RANGEFAIL* si uno o ambos operandos tienen una clase dinámica y falla la parte dinámica de la verificación de compatibilidad mencionada anteriormente.

ejemplos:

10.31 *i<min* *(1.1)*

10.31 *i<min* **OR** *i>max* *(1.2)*

5.3.4 Operando–1

sintaxis:

<operando–1> ::= *(1)*
<operando–2> *(1.1)*
| *<suboperando–1>* { **AND** | **ANDIF** } *<operando–2>* *(1.2)*

<suboperando–1> ::= *(2)*
<operando–1> *(2.1)*

semántica: Si se especifica **AND** o **ANDIF**, el *suboperando–1* y el *operando–2* entregan:

- valores booleanos, en cuyo caso **AND** denota la operación "conjunción" lógica, que entrega un valor booleano. Si se especifica **ANDIF** y el *suboperando–1* entrega el valor booleano *FALSE*, este es el resultado; en otro caso, el resultado es el *operando–2*;
- valores cadena de bits, en cuyo caso **AND** denota la operación lógica sobre cada elemento de las cadenas de bits, y entrega un valor cadena de bits;
- valores conjuntistas, en cuyo caso **AND** denota la operación "intersección" de valores conjuntistas, que entrega como resultado un valor conjuntista.

propiedades estáticas: Si un *operando–1* es un *operando–2*, la clase del *operando–1* es la clase del *operando–2*.

Si se especifica **AND** o **ANDIF**, la clase del *operando–1* es la **clase resultante** de las clases del *suboperando–1* y el *operando–2*.

Un *operando–1* es **constante** (**literal**) si y sólo si es un *operando–2* que es **constante** (**literal**), o está formado por un *operando–1* y un *operando–2* que son ambos **constantes** (**literales**).

condiciones estáticas: Si se especifica **AND** o **ANDIF**, la clase del *suboperando–1* debe ser **compatible** con la clase del *operando–2*. Si se especifica **ANDIF**, ambas clases deben tener un modo **raíz** booleano; en otro caso, ambas clases deben tener un modo **raíz** booleano, conjuntista o de cadena de bits, en cuyo caso la **longitud efectiva** del *sub-operando-1* y del *operando–2* debe ser la misma. Esta verificación es dinámica si uno o ambos modos son modos dinámicos o de cadena **variable**.

condiciones dinámicas: En el caso de **AND**, se produce una excepción *RANGEFAIL* si uno o ambos operandos tienen una clase dinámica y falla la parte dinámica de la verificación de compatibilidad mencionada anteriormente.

ejemplos:

5.10 (*a1* **OR** *b1*) *(1.1)*

5.10 **NOT** *k2* **AND** (*a1* **OR** *b1*) *(1.2)*

5.3.5 Operando–2

sintaxis:

<operando–2> ::= *(1)*
    <operando–3> *(1.1)*
  | *<suboperando–2> <operador–3> <operando–3>* *(1.2)*

<suboperando–2> ::= *(2)*
    <operando–2> *(2.1)*

<operador–3> ::= *(3)*
    <operador relacional> *(3.1)*
  | *<operador de pertenencia>* *(3.2)*
  | *<operador de inclusión conjuntista>* *(3.3)*

<operador relacional> ::= *(4)*
    = | / = | > | > = | < | < = *(4.1)*

<operador de pertenencia> ::= *(5)*
    IN *(5.1)*

<operador de inclusión conjuntista> ::= *(6)*
    < = | > = | < | > *(6.1)*

semántica: Los operadores de igualdad (=) y desigualdad (/=) se definen entre todos los valores de un modo dado. Los restantes operadores relacionales (menor que: <, menor o igual que: <=, mayor que: >, mayor o igual que: >=) se definen entre valores de un determinado modo discreto, temporización o cadena. Todos los operadores relacionales entregan como resultado un valor booleano.

El operador de pertenencia se define entre un valor es miembro y un valor conjuntista. El operador entrega *TRUE* si el valor miembro pertenece al valor conjuntista especificado; en otro caso, entrega *FALSE*.

Los operadores de inclusión conjuntista se definen entre valores conjuntistas y comprueban si un valor conjuntista está o no contenido en : <=, está contenido correctamente en: <, contiene: >=, o contiene correctamente > el otro valor conjuntista. Un operador de inclusión conjuntista proporciona como resultado un valor booleano.

propiedades estáticas: Si un *operando–2* es un *operando–3*, la clase del *operando–2* es la clase del *operando–3*. Si se especifica un *operador–3*, la clase del *operando–2* es la clase *BOOL*-derivada.

Un *operando–2* es **constante** (**literal**) si y sólo si es un *operando–3* que es **constante** (**literal**), o está formado por un *suboperando–2* y un *operando–3* que son ambos **constantes** (**literales**).

condiciones estáticas: Si se especifica un *operador–3*, deben cumplirse los siguientes requisitos de compatibilidad entre la clase del *suboperando–2* y la clase del *operando–3*:

- si el *operador–3* es = o /=, ambas clases deben ser **compatibles**;
- si el *operador–3* es un *operador relacional* diferente de = o /=, ambas clases deben ser **compatibles** y tener un modo **raíz** discreto, temporización o cadena;
- si el *operador–3* es un *operador de pertenencia*, la clase del *operando–3* debe tener un modo **raíz** conjuntista y la clase del *suboperando–2* debe ser **compatible** con el modo **miembro** del citado modo **raíz**;
- si el *operador–3* es un *operador de inclusión conjuntista*, ambas clases deben ser **compatibles** y poseer un modo **raíz** conjuntista.

condiciones dinámicas: En el caso de un *operador relacional*, se produce una excepción *RANGEFAIL* o *TAGFAIL* si uno o ambos operandos tienen una clase dinámica y falla la parte dinámica de la verificación de compatibilidad mencionada anteriormente. La excepción *TAGFAIL* se produce si y sólo si una clase dinámica se basa en un modo estructura **parametrizada** dinámico.

ejemplos:

10.50 NULL *(1.1)*

10.50 last=NULL *(1.2)*

5.3.6 Operando–3

sintaxis:

<operando–3> ::= *(1)*
    <operando–4> *(1.1)*
  | *<suboperando–3> <operador–4> <operando–4>* *(1.2)*

<suboperando–3> ::= *(2)*
    <operando–3> *(2.1)*

<operador–4> ::= *(3)*
    <operador aritmético aditivo> *(3.1)*
  | *<operador de concatenación de cadena>* *(3.2)*
  | *<operador de diferencia conjuntista>* *(3.3)*

<operador aritmético aditivo> ::= *(4)*
    + | – *(4.1)*

<operador de concatenación de cadena> ::= *(5)*
    // *(5.1)*

<operador de diferencia conjuntista> ::= *(6)*
    – *(6.1)*

semántica: Si el *operador–4* es un operador aritmético aditivo, ambos operandos entregan valores enteros o valores de coma flotante, y el valor entero resultante o valor de coma flotante, respectivamente, es la suma (+) o diferencia (–) de los dos valores.

Si el *operador–4* es un operador de concatenación de cadena, ambos operandos entregan valores de cadena de bits o de cadena de caracteres; el valor resultante es la concatenación de estos valores. Se permiten también valores booleanos (carácter); éstos se consideran valores de cadena de bits (caracteres) de longitud 1.

Si el *operador–4* es un operador de diferencia conjuntista, ambos operandos entregan valores conjuntistas, y el valor resultante es el valor conjuntista constituido por los valores miembros que están en el valor entregado por el *suboperando–3* y que no están en el valor entregado por el *operando–4*.

Si la clase de *operando–3* tiene un modo **raíz** coma flotante, el resultado es el valor coma flotante que aproxima, utilizando el mismo criterio utilizado para conversión de representación, el resultado de la operación matemática exacta.

propiedades estáticas: Si un *operando–3* es un *operando–4*, la clase del *operando–3* es la del *operando–4*. Si se especifica un *operador–4*, a partir de él se determina la clase del *operando–3* como sigue:

- Si el *operador–4* es un *operador de concatenación de cadena*, la clase del *operando–3* depende de las clases del *operando–4* y del *suboperando–3*, en las que un operando que es un valor booleano o de carácter se considera como un valor cuya clase es una clase **BOOLS** *(1)*-derivada, o una clase **CHARS** *(1)*-derivada, respectivamente:
  - si ninguna de ellas es **fuerte**, la clase es la clase **BOOLS** *(n)*-derivada o la clase **CHARS** *(n)*-derivada, según que ambos operandos sean cadenas de bits o de caracteres, donde *n* es la suma de las **longitudes de cadena** de los modos **raíz** de ambas clases;
  - en otro caso la clase es la clase *&nombre(n)*-valuada, donde *&nombre* es un nombre **de sínmodo** virtual, **sinónimo** del modo **raíz** de la **clase resultante** de las clases de los operandos, y *n* es la suma de las **longitudes de cadena** de los modos **raíz** de ambas clases;

    (esta clase es dinámica si uno o ambos operandos tienen una clase dinámica);
- si el *operador–4* es un *operador aritmético aditivo* o un *operador de diferencia conjuntista*, la clase del *operando–3* es la **clase resultante** de las clases del *operando–4* y del *suboperando–3*.

Un *operando–3* es **constante** (**literal**) si y sólo si es un *operando–4* que es **constante** (**literal**), o está formado por un *operando–3* y un *operando–4* que son ambos **constantes** (**literales**), y el *operador–4* es el *operador aritmético aditivo* o el *operador de diferencia conjuntista*.

Si el *operador–4* es el *operador de concatenación de cadena*, un *operando–3* es **constante** si está formado por un *operando–3* y un *operando–4* que son ambos **constantes**.

condiciones estáticas: Si se especifica un *operador–4*, deben cumplirse los siguientes requisitos de compatibilidad:

- si el *operador–4* es un *operador aritmético aditivo*, las clases de ambos operandos deben ser **compatibles** y poseer ambas un modo **raíz** entero o coma flotante; Además, si *operando–3* no es **constante**, el modo **raíz** de la clase de *operando–3* debe ser un modo entero **predefinido** o un modo coma flotante **predefinido**.
- si el *operador–4* es un *operador de concatenación de cadena*, entonces:
  - las clases de ambos operandos deben ser **compatibles** y poseer ambas un modo **raíz** cadena **de bits** o un modo **raíz** cadena **de caracteres**, o
  - las clases de ambos operandos deben ser **compatibles** con el modo *BOOL* o con el modo *CHAR*, o
  - la clase de un operando debe tener un modo **raíz** cadena de **bits** (**caracteres**), y la del otro debe ser **compatible** con el modo *BOOL* (*CHAR*);
- si el *operador–4* es un *operador de diferencia conjuntista*, las clases de ambos operandos deben ser **compatibles** y poseer ambas un modo **raíz** conjuntista.

condiciones dinámicas: En el caso de un *operando–3* que no es **constante**, si el *operador–4* es un *operador aritmético aditivo*, se produce una excepción *OVERFLOW* si una adición (+) o una sustracción (–) da lugar a un valor que no es uno de los valores definidos por el modo **raíz** de la clase del *operando–3*, o uno o ambos operandos no pertenecen al conjunto de valores del modo **raíz** del *operando–3*.

En el caso de un *operando–3* que no es **constante**, se produce una excepción *UNDERFLOW* si la clase del *operando–3* tiene un modo **raíz** coma flotante y la adición (+) o sustracción (–) matemáticas exactas dan lugar a un valor que es mayor que el **límite superior negativo** y menor que el **límite inferior positivo** del modo **raíz** del *operando–3*, y es diferente de cero.

ejemplos:

1.6 j *(1.1)*

1.6 i+j *(1.2)*

5.3.7 Operando–4

sintaxis:

<operando–4> ::= *(1)*
<operando–5> *(1.1)*
| *<suboperando–4> <operador aritmético multiplicativo> <operando–5>* *(1.2)*

<suboperando–4> ::= *(2)*
<operando–4> *(2.1)*

<operador aritmético multiplicativo> ::= *(3)*
∗ | / | **MOD** | **REM** *(3.1)*

semántica: Si el operador aritmético multiplicativo es el operador producto (∗) o el operador cociente (/), el *suboperando–4* y el *operando–5* entregan valores enteros o valores coma flotante y el valor entero o valor coma flotante, respectivamente, resultante es el producto o cociente de ambos valores.

Si el operador aritmético multiplicativo es el operador módulo (**MOD**) o el operador resto de división (**REM**), el *suboperando–4* y el *operando–5* entregan valores enteros y el valor entero resultante es el módulo o resto de división de ambos valores.

La operación módulo se define de forma que *i* **MOD** *j* entrega el valor entero único *k,* $0 \leq k < j$, tal que existe un valor entero *n* para el cual $i = n * j + k$; *j* debe ser mayor que 0.

La operación cociente se define como una operación tal que las tres relaciones:

$ABS(x/y) = ABS(x) / ABS(y)$ y

signo (x/y) = signo (x) / signo (y) y

$ABS(x) - (ABS(x) / ABS(y)) * ABS(y) = ABS(x)$ **MOD** $ABS(y)$

dan *TRUE* para todos los valores enteros de *x* e *y*, siendo signo $(x) = -1$ si $x < 0$, y en otro caso signo $(x) = 1$.

La operación resto se define como una operación tal que *x* **REM** $y = x - (x/y) * y$ da *TRUE* para todos los valores enteros de *x* e *y*.

Si la clase de *operando–4* tiene un modo **raíz** coma flotante, el resultado es el valor coma flotante que aproxima, utilizando el mismo criterio utilizado para conversión de representación, el resultado de la operación matemática exacta.

propiedades estáticas: Si el *operando–4* es un *operando–5*, la clase del *operando–4* es la clase del *operando–5*; en otro caso, la clase del *operando–4* es la **clase resultante** de las clases del *suboperando–4* y del *operando–5*.

Un *operando–4* es **constante** (**literal**) si y sólo si es un *operando–5* que es **constante** (**literal**), o está formado a partir de un *operando–4* y un *operando–5* que son ambos **constantes** (**literales**).

condiciones estáticas: Si se especifica un *operador aritmético multiplicativo* entre operandos entero y coma flotante, las clases del *operando–5* y del *suboperando–4* deben ser **compatibles**, y poseer ambas un modo **raíz** entero, o un modo **raíz** coma flotante, respectivamente. Además, si el *operando–4* no es **constante**, el modo **raíz** de la clase del *operando-4* debe ser un modo entero **predefinido** o un modo coma flotante **predefinido**.

condiciones dinámicas: En el caso de un *operando–4* que no es **constante**, si se especifica un *operador aritmético multiplicativo* se produce una excepción *OVERFLOW* si una operación multiplicación (∗), división (/), módulo (**MOD**) o resto (**REM**) da lugar a un valor que no es ninguno de los valores definidos por el modo **raíz** de la clase del *operando–4*, o si dicha operación se efectúa sobre valores operandos para los que el operador no está definido matemáticamente, por ejemplo, una división o resto con un *operando–5* que entregue 0 o una operación módulo con un *operando–5* que entregue un valor entero no positivo, uno o ambos operandos no pertenezcan al conjunto de valores del modo **raíz** del *operando–4*.

En el caso de un *operando–4* que no es **constante**, se produce una excepción *UNDERFLOW* si la clase del *operando–4* tiene un modo **raíz** coma flotante y la multiplicación (*) o división (/) matemáticas exactas dan lugar a un valor que es mayor que el **límite superior negativo** y menor que el **límite inferior positivo** del modo **raíz** del *operando–4*, y es diferente de cero.

ejemplos:

6.15 1_461 *(1.1)*

6.15 (4 ∗ d + 3) / 1_461 *(1.2)*

5.3.8 Operando–5

sintaxis:

<operando–5> ::= *(1)*
<operando–6> *(1.1)*
| *<suboperando–5> <operador exponenciación> <operando–6>* *(1.2)*

<suboperando–5> ::= *(2)*
<operando–5> *(2.1)*

<operador exponenciación> ::= *(3)*
** *(3.1)*

semántica: Si se especifica el *operador exponenciación*, el *suboperando–5* y el *operando–6* entregan un valor coma flotante o un valor entero. El valor resultante es el que se obtiene elevando el valor entregado por el *suboperando–5* a la potencia del entregado por el *operando–6*.

Si la clase del *operando–5* tiene un modo **raíz** coma flotante, el resultado es el valor coma flotante que aproxima, utilizando el mismo criterio utilizado para conversión de representación, el resultado de la operación matemática exacta.

propiedades estáticas: Si el *operando–5* es un *operando–6*, la clase del *operando–5* es la clase del *operando–6*.

Si se especifica el *operador exponenciación*, la clase del *operando–5* es la del *suboperando–5*.

Un *operando–5* es **constante** (**literal**) si y sólo si es un *operando–6 que es* **constante** (**literal**), o se ha formado a partir de un *operando–5* y un *operando–6* que son, ambos, **constantes** (**literales**)

condiciones estáticas: Si se especifica un *operador exponenciación*:

- si la clase del *suboperando–5* tiene un modo **raíz** coma flotante, la clase del *operando–6* debe tener un modo **raíz** entero o un modo **raíz** coma flotante.
- en otro caso, la clase del *suboperando–5* debe tener un modo **raíz** entero y la clase del *operando–6* debe tener un modo **raíz** entero.

condiciones dinámicas: En el caso de un *operando–5* que no es **constante**, se produce una excepción *OVERFLOW* si una operación de exponenciación da lugar a un valor fuera del intervalo del modo **raíz** de la clase del *operando–5*.

En el caso de un *operando–5* que no es **constante**, se produce una excepción *UNDERFLOW* si la clase del *operando–5* tiene un modo **raíz** y la exponenciación matemática exacta da lugar a un valor que es menor que el **límite inferior positivo** del modo **raíz** del *operando–5*.

Si se especifica un *operador exponenciación* y la clase del *operando–5* tiene un modo **raíz** entero, entonces, si el *operando–6* no es **constante**, su valor deber ser superior o igual a cero.

ejemplo:

*r ** 4* *(1.2)*

5.3.9 Operando–6

sintaxis:

<operando–6> ::= *(1)*
[*<operador monádico>*] *<operando–7>* *(1.1)*
| *<literal entero con signo>* *(1.2)*
| *<literal coma flotante con signo>* *(1.3)*

<operador monádico> ::= *(2)*
– | **NOT** *(2.1)*
| *<operador de repetición de cadena>* *(2.2)*

<operador de repetición de cadena> ::= *(3)*
(*<expresión literal entera>*) *(3.1)*

NOTA – Si el operador *monádico* es el operador cambio de signo (–), el *operando–7* es un *literal entero sin signo* o un *literal coma flotante sin signo*, la construcción sintáctica es ambigua y se interpretará como un *literal entero con signo* o un *literal coma flotante con signo*, respectivamente.

semántica: Si el operador monádico es el operador cambio de signo (–), el *operando–7* entrega un valor entero o un valor coma flotante, y el entero o coma flotante resultante es el valor entero o coma flotante anterior con su signo cambiado.

Si el operador monádico es **NOT**, el *operando–7* entrega un valor booleano, un valor cadena de bits o un valor conjuntista. En los dos primeros casos, el resultado es la negación lógica del valor booleano o de los elementos del valor cadena de bits; en el último, es el valor complemento de conjunto, es decir, el conjunto de los valores miembros que no pertenecen al valor conjuntista del operando.

Si el operador monádico es un operador de repetición de cadena, el *operando–7* es un *literal de cadena de caracteres* o un *literal de cadena de bits*. Si la *expresión literal entera* entrega 0, el resultado es el valor cadena vacía; en otro caso, el

resultado es el valor cadena formado concatenando la cadena consigo misma tantas veces como especifique el valor proporcionado por la *expresión literal entera* menos 1.

propiedades estáticas: Si el *operando–6* es un *operando–7*, la clase del *operando–6* es la clase del *operando–7*.

Si se especifica un *operador monádico*, la clase del *operando–6* es:

- si el *operador monádico* es – o **NOT**, la **clase resultante** es la clase del *operando–7*;
- si el *operador monádico* es el *operador de repetición de cadena*, la clase **CHARS** (*n*)-derivada o **BOOLS** (*n*)-derivada (según que el literal sea un *literal de cadena de caracteres* o un *literal de cadena de bits*), donde $n = r * l$, siendo r el valor proporcionado por la *expresión literal entera* y *l* la **longitud de cadena** del literal de cadena.

Un *operando–6* es **constante** si y sólo si lo es el *operando–7*. Un *operando–6* es **literal** si y sólo si el *operando–7* es **literal** y el *operador monádico* es – o **NOT**.

condiciones estáticas: Si el *operador monádico* es –, la clase del *operando–7* debe tener un modo **raíz** entero o un modo **raíz** coma flotante. Además, si el *operando–6* no es **constante**, el modo **raíz** de la clase del *operando–6* debe ser un modo entero **predefinido** o un modo coma flotante **predefinido**.

Si el *operador monádico* es **NOT**, la clase del *operando–7* debe tener un modo **raíz** booleano, cadena de **bits** o conjuntista.

Si el *operador monádico* es el *operador de repetición de cadena*, el *operando–7* debe ser un *literal de cadena de caracteres* o un *literal de cadena de bits*. La *expresión literal entera* debe entregar un valor entero no negativo.

condiciones dinámicas: Si el *operando–6* no es **constante**, se produce una expresión *OVERFLOW* si una operación cambio de signo (–) origina un valor que no es ninguno de los valores definidos por el modo **raíz** de la clase del *operando–6*.

En el caso de un *operando–6* que no es **constante** , se produce una excepción *UNDERFLOW* si la clase del *operando–6* tiene un modo **raíz** coma flotante y la operación matemática exacta de cambio de signo (–) da lugar a un valor que es mayor que el **límite superior negativo** y menor que el **límite inferior positivo** del modo **raíz** del *operando–6*, y es diferente de cero.

ejemplos:

5.10 **NOT** *k2* *(1.1)*

7.54 *(6)" "* *(1.1)*

7.54 *(6)* *(2.2)*

5.3.10 Operando–7

sintaxis:

<operando–7> ::= *(1)*
<localización referenciada> *(1.1)*
| *<valor primitivo>* *(1.2)*

<localización referenciada> ::= *(2)*
–> <localización> *(2.1)*

semántica: Una localización referenciada entrega una referencia a la localización referenciada.

propiedades estáticas: La clase de un *operando–7* es la clase de la *localización referenciada* o *valor primitivo*, respectivamente. La clase de la *localización referenciada* es la clase M-referencia, siendo M el modo de la *localización*.

Un *operando–7* es **constante** si y sólo si el *valor primitivo* es **constante** o la *localización referenciada* es **constante**. Una *localización referenciada* es **constante** si y sólo si la *localización* es **estática**. Un *operando–7* es **literal** si y sólo si el *valor primitivo* es **literal**.

condiciones estáticas: La *localización* debe ser **referenciable**.

ejemplo:

8.25 *–> c* *(2.1)*

6 Acciones

6.1 Generalidades

sintaxis:

<sentencia de acción> ::= *(1)*
[*<ocurrencia de definición>* :] *<acción>* [*<manejador>*]
[*<cadena de nombre simple>*]; *(1.1)*
| *<módulo>* *(1.2)*
| *<módulo de espec>* *(1.3)*
| *<módulo de contexto>* *(1.4)*

<acción> ::= *(2)*
<acción encorchetada> *(2.1)*
| *<acción de asignación>* *(2.2)*
| *<acción llamar>* *(2.3)*
| *<acción salir>* *(2.4)*
| *<acción retornar>* *(2.5)*
| *<acción resultar>* *(2.6)*
| *<acción ir a>* *(2.7)*
| *<acción afirmar>* *(2.8)*
| *<acción vacía>* *(2.9)*
| *<acción arrancar>* *(2.10)*
| *<acción parar>* *(2.11)*
| *<acción demorar>* *(2.12)*
| *<acción continuar>* *(2.13)*
| *<acción enviar>* *(2.14)*
| *<acción causar>* *(2.15)*

<acción encorchetada> ::= *(3)*
<acción condicional> *(3.1)*
| *<acción de caso>* *(3.2)*
| *<acción hacer>* *(3.3)*
| *<bloque principio-fin>* *(3.4)*
| *<acción demorar y elegir>* *(3.5)*
| *<acción recibir y elegir>* *(3.6)*
| *<acción de temporización>* *(3.7)*

semántica: Las sentencias de acción constituyen la parte algorítmica de un programa CHILL. Toda sentencia de acción puede etiquetarse. Las acciones que nunca pueden causar una excepción no pueden tener agregado un manejador.

propiedades estáticas: Una *ocurrencia de definición* en una *sentencia de acción* define un nombre de **etiqueta**.

condiciones estáticas: La *cadena de nombre simple* sólo puede darse después de una *acción* que sea una *acción encorchetada* o si se especifica un *manejador*, y únicamente si se especifica una *ocurrencia de definición*. La *cadena de nombre simple* debe ser la misma cadena de nombre que la *ocurrencia de definición*.

6.2 Acción de asignación

sintaxis:

<acción de asignación> ::= *(1)*
<acción de asignación simple> *(1.1)*
| *<acción de asignación múltiple>* *(1.2)*

<acción de asignación simple> ::= *(2)*
<localización> <símbolo de asignación> <valor> *(2.1)*
| *<localización> <operador de asignación> <expresión>* *(2.2)*

<acción de asignación múltiple> ::= *(3)*
<localización> { , *<localización>* }$^{+}$ *<símbolo de asignación> <valor>* *(3.1)*

<operador de asignación> ::= *(4)*
<operador diádico cerrado> <símbolo de asignación> *(4.1)*

<operador diádico cerrado> ::= *(5)*
OR | XOR | AND *(5.1)*
| *<operador de diferencia conjuntista>* *(5.2)*
| *<operador aritmético aditivo>* *(5.3)*
| *<operador aritmético multiplicativo>* *(5.4)*
| *<operador de concatenación de cadena>* *(5.5)*

<símbolo de asignación> ::= *(6)*
:= *(6.1)*

semántica: Una acción de asignación almacena un valor en una o más localizaciones.

Si se utiliza un símbolo de asignación, el valor proporcionado por el lado derecho se almacena en la(s) localización(es) en el lado izquierdo.

Si se emplea un operador de asignación, el valor contenido en la localización se combina con el valor del lado derecho (en ese orden), de acuerdo con la semántica del operador diádico cerrado especificado, y el resultado se vuelve a almacenar en la misma localización.

La evaluación de las localizaciones del lado izquierdo, del valor del lado derecho, y de las propias asignaciones se efectúa en cualquier orden. Toda asignación puede efectuarse tan pronto como se hayan evaluado el valor y la localización.

Si la localización (o cualquiera de las localizaciones) es el campo **marcador** de una estructura variable, la semántica para los campos variables que dependen de ella estará definida por la implementación.

condiciones estáticas: Los modos de todas las ocurrencias de *localización* deben ser **equivalentes** y no deben tener ni la **propiedad de lectura estamente**, ni la **propiedad de no-valor**. Cada modo debe ser **compatible** con la clase del *valor*. Las verificaciones son dinámicas cuando afecten a localizaciones de modo dinámico y/o a valores con clase dinámica.

El *valor* tiene que ser **regionalmente seguro** para cada *localización* (véase 11.2.2).

Si cualquier *localización* tiene un modo cadena **fijo**, la **longitud de cadena** del modo y la **longitud efectiva** del valor deben ser las mismas; en otro caso, si tiene un modo cadena **variable**, la **longitud de cadena** del modo no debe ser inferior a la **longitud efectiva** del valor. Esta comprobación es dinámica si uno o ambos modos son modos dinámicos o modos cadena **variable**. Esta condición se llama condición de asignación de cadena.

Si una de las asignaciones es de la forma "pvl–> := pvr–>;" donde pvl y pvr tienen el modo "REF ML" y "REF MR" respectivamente, y ML y MR son nombres de modo moreta, ML y MR deben estar en el mismo camino.

Si una de las asignaciones es de la forma "pvl–> := mr;" donde pvl tiene el modo "REF ML", y ML y el nombre de modo de mr son nombres de modo módulo, entonces debe cumplirse que mr succ ML.

Si una de las asignaciones es de la forma "ml–> := pvr–>;" donde pvr tiene el modo "REF MR", y MR y el nombre de modo de ml son nombres de modo módulo, entonces debe cumplirse que MR succ ml.

Si el modo de cualquiera de las localizaciones del lado izquierdo es un modo módulo, los nombres de modo de todos esos modos tienen que ser sinónimos por pares.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* o *TAGFAIL* si el modo de la localización y/o del valor son modos dinámicos y falla la parte dinámica de las verificaciones de compatibilidad mencionadas anteriormente.

Se produce la excepción *RANGEFAIL* si el modo de la localización y/o del valor son modos cadena **variable** y falla la parte dinámica de las verificaciones de compatibilidad mencionadas anteriormente.

Se produce la excepción *RANGEFAIL* si cualquier *localización* tiene un modo intervalo discreto (modo intervalo coma flotante) y el valor entregado por la evaluación de *valor* no es ni uno de los valores definidos por el modo intervalo discreto (modo intervalo coma flotante) ni el valor **indefinido**.

Si el modo de cualquier localización L es del tipo REF MM, donde MM es un modo moreta, debe cumplirse lo siguiente: el modo del valor vigente del rhs debe ser un sucesor del modo de L; en otro caso se produce la excepción *RANGEFAIL*.

Las condiciones dinámicas mencionadas, junto con la condición de asignación de cadena, se denominan condiciones de asignación de un valor con respecto a un modo.

En el caso de un *operador de asignación*, se producen las mismas excepciones que si se evaluase la expresión:

<localización> <operador diádico cerrado> (*<expresión>*)

y el valor proporcionado se almacenase en la localización especificada (obsérvese que la localización se evalúa una vez solamente).

Si el modo de cualquier localización L es del tipo "REF MM", donde MM es un modo moreta, el modo del valor vigente del rhs debe ser un sucesor del modo de L; en otro caso se produce la excepción *RANGEFAIL*.

Si cualquiera de las asignaciones es de la forma "pvl–> := pvr–>;", "pvl–> := mr;" or "ml := pvr–>;", donde pvl y pvr son los modos "REF ML" y "REF MR" respectivamente y ML, MR, ml y mr son modos módulo, entonces los modos vigentes del lhs y el rhs deben cumplir las reglas para la asignación de modos módulo.

ejemplos:

4.12 *a := b+c* *(1.1)*

10.25 *stackindex- := 1* *(2.1)*

19.19 *x–>.prev, x–>.next := NULL* *(3.1)*

10.25 *– :=* *(4.1)*

6.3 Acción condicional

sintaxis:

<acción condicional> ::= *(1)*
IF *<expresión booleana>* *<cláusula entonces>* [*<cláusula si no>*] **FI** *(1.1)*

<cláusula entonces> ::= *(2)*
THEN *<lista de sentencias de acción>* *(2.1)*

<cláusula si no> ::= *(3)*
ELSE *<lista de sentencias de acción>* *(3.1)*
| **ELSIF** *<expresión booleana>* *<cláusula entonces>* [*<cláusula si no>*] *(3.2)*

sintaxis derivada: La notación:

ELSIF *<expresión booleana>* *<cláusula entonces>* [*<cláusula si no>*]

es sintaxis derivada de:

ELSE IF *<expresión booleana>* *<cláusula entonces>* [*<cláusula si no>*] **FI**;

semántica: Una acción condicional es una bifurcación condicional. Si la *expresión booleana* produce *TRUE*, se entra en la lista de sentencias de acción que sigue a **THEN**; en otro caso, se entra en la lista de sentencias de acción que sigue a **ELSE**, si existe.

condiciones dinámicas: Se produce la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplos:

7.22
```
IF n >= 50 THEN rn(r) := 'L';
                n– := 50;
                r+ := 1;
FI
```
(1.1)

10.50
```
IF last = NULL
                THEN first,last := p;
                ELSE last–>.succ := p;
                p–>.pred := last;
                last := p;
FI
```
(1.1)

6.4 Acción de caso

sintaxis:

<acción de caso> ::= *(1)*
CASE *<lista de selectores de caso>* **OF** [*<lista de intervalos>*;]
{ *<alternativa de caso>* }$^{+}$
[**ELSE** *<lista de sentencias de acción>*] **ESAC** *(1.1)*

<lista de selectores de caso> ::= *(2)*
*<expresión discreta> { , <expresión discreta> }** *(2.1)*

<lista de intervalos> ::= *(3)*
*<nombre de modo discreto> { , <nombre de modo discreto> }** *(3.1)*

<alternativa de caso> ::= *(4)*
<especificación de etiqueta de caso> : <lista de sentencias de acción> *(4.1)*

semántica: La acción de caso es una ramificación múltiple. Consta de la especificación de una o más expresiones discretas (la lista de selectores de caso) y cierto número de listas de sentencias de acción etiquetadas (alternativas de caso). Cada lista de sentencias de acción se etiqueta con una especificación de etiqueta de caso compuesta de una lista de especificaciones de etiqueta de caso (una para cada selector de caso). Cada etiqueta de caso define un conjunto de valores. La utilización de una lista de expresiones discretas en la lista de selectores de caso permite seleccionar una alternativa basada en múltiples condiciones.

La acción de caso produce la entrada en la lista de sentencias de caso para la cual los valores dados en la especificación de etiqueta de caso concuerdan con los valores de la lista de selectores de caso; si no concuerda ningún valor, se pasa a la *lista de sentencias de acción* que sigue a **ELSE**.

Las expresiones de la lista de selectores de caso se evalúan en un orden indefinido y posiblemente mixto. Necesitan evaluarse solamente hasta el punto en que se determina una alternativa de caso.

condiciones estáticas: Para la lista de ocurrencias de *especificación de etiqueta de caso*, se aplican las condiciones de selección de caso (véase 12.3).

El número de ocurrencias en de *expresión discreta* en la *lista de selectores de caso*, debe ser igual al número de clases de la **lista resultante de clases** de la lista de ocurrencias de *lista de etiquetas de caso* y, si existe, al número de ocurrencias de *nombre de modo discreto* de la *lista de intervalos*.

La clase de cualquier *expresión discreta* en la *lista de selectores de caso* debe ser **compatible** con la clase correspondiente (en posición) de la **lista resultante de clases** de las ocurrencias de *lista de etiquetas de caso* y, si está presente, **compatible** con el *nombre de modo discreto* (en posición) de la *lista de intervalos*. Dicho modo debe ser asimismo **compatible** con la clase correspondiente de la **lista resultante de clases**.

Todo valor proporcionado por una *expresión literal discreta*, o definido por un *intervalo de literal* o un *nombre de modo discreto* en una *etiqueta de caso* (véase 12.3), debe estar en el intervalo del *nombre de modo discreto* correspondiente de la *lista de intervalos*, si existe, y asimismo en el intervalo definido por el modo de la *expresión discreta* correspondiente de la *lista de selectores de caso*, si se trata de una *expresión discreta* **fuerte**. En este último caso, los valores definidos por el *nombre de modo discreto* correspondiente de la *lista de intervalos*, si existe, deben estar también incluidos en ese intervalo.

La parte opcional **ELSE** conforme a la sintaxis sólo puede omitirse si la lista de ocurrencias de *lista de etiquetas de caso* es **completa** (véase el 12.3).

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si se especifica una *lista de intervalos*, y el valor proporcionado por una *expresión discreta* de la *lista de selectores de caso* no está comprendido entre los límites especificados por el *nombre de modo discreto* correspondiente de la *lista de intervalos*.

Se produce la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplos:

4.11
```
CASE order OF
        (1): a := b+c;
             RETURN;
        (2): d := 0;
        (ELSE): d := 1;
ESAC
```
(1.1)

11.43 *starting.p.kind, starting.p.color* *(2.1)*

11.58
```
(rook),(*):
        IF NOT ok_rook(b,m)
            THEN
                CAUSE illegal;
        FI;
```
(4.1)

6.5 Acción hacer

6.5.1 Generalidades

sintaxis:

<acción hacer> ::= *(1)*
DO [*<parte de control>*;] *<lista de sentencias de acción>* **OD** *(1.1)*

<parte de control> ::= *(2)*
<control para> [*<control mientras>*] *(2.1)*
| *<control mientras>* *(2.2)*
| *<parte con>* *(2.3)*

semántica: Una acción hacer tiene una de tres formas diferentes: las versiones hacer-para y hacer mientras, ambas para formar bucles, y la versión hacer-con que es una forma abreviada conveniente para acceder a campos de estructura de un modo eficaz. Si no se especifica una parte de control, se entra una vez en la lista de sentencias de acción cada vez que se entra en la acción hacer.

Cuando se combinan las versiones hacer-para y hacer-mientras, el control mientras se evalúa después del control para, y solamente en el caso en que la acción hacer no sea terminada por el control para.

Si la parte control especificada es de control para y/o control mientras, entonces en tanto que el control permanezca dentro del dominio de la acción hacer, se entra en la lista de sentencias de acción de acuerdo con la parte control, pero no se vuelve a entrar en el dominio hacer para cada ejecución de la lista de sentencias de acción.

condiciones dinámicas: Se produce la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplos:

4.17
```
DO FOR i := 1 TO c;
        op(a,b,d,order–1);
        d := a;
OD                                                    (1.1)
```

15.58
```
DO WITH each;
        IF this_counter = counter
            THEN
                status := idle;
                EXIT find_counter;
        FI;
OD                                                    (1.1)
```

6.5.2 Control para (o control de iteración)

sintaxis:

<control para> ::= *(1)*
FOR { *<iteración>* {, *<iteración>*}* | **EVER** } *(1.1)*

<iteración> ::= *(2)*
<enumeración de valor> *(2.1)*
| *<enumeración de localización>* *(2.2)*

<enumeración de valor> ::= *(3)*
<enumeración por paso> *(3.1)*
| *<enumeración por intervalo>* *(3.2)*
| *<enumeración conjuntista>* *(3.3)*

<enumeración por paso> ::= *(4)*
<contador de bucle> <símbolo de asignación>
<valor inicial> [*<valor de paso>*] [**DOWN**] *<valor final>* *(4.1)*

<contador de bucle> ::= *(5)*
<ocurrencia de definición> *(5.1)*

<valor inicial> ::= *(6)*
<expresión discreta> *(6.1)*

<valor de paso> ::= *(7)*
BY *<expresión entera>* *(7.1)*

<valor final> ::= *(8)*
TO *<expresión discreta>* *(8.1)*

<enumeración por intervalo> ::= *(9)*
<contador de bucle> [**DOWN**] **IN** *<nombre de modo discreto>* *(9.1)*

<enumeración conjuntista> ::= *(10)*
<contador de bucle> [**DOWN**] **IN** *<expresión conjuntista>* *(10.1)*

<enumeración de localización> ::= *(11)*
<contador de bucle> [**DOWN**] **IN** *<objeto compuesto>* *(11.1)*

<objeto compuesto> ::= *(12)*
<localización matriz> *(12.1)*
| *<expresión matriz>* *(12.2)*
| *<localización cadena>* *(12.3)*
| *<expresión cadena>* *(12.4)*

NOTA – Si el *objeto compuesto* es una *localización* (*cadena o matriz*), se resuelve la ambigüedad sintáctica interpretando *objeto compuesto* como una *localización* y no como una *expresión*.

semántica: El control para puede mencionar varios contadores de bucle. Los contadores de bucle se evalúan cada vez en un orden no especificado, antes de entrar en la lista de sentencias de acción; esta evaluación es necesaria solamente hasta el punto en que pueda decidirse la terminación de la acción hacer. La acción hacer termina si al menos uno de los contadores de bucle indica terminación.

1) **hacer permanente**:

Se repite indefinidamente la lista de acciones. La acción hacer sólo puede terminar por una transferencia del control fuera de ella.

2) **enumeración de valor**:

Se entra repetidamente en la lista de sentencias de acción para el conjunto de valores especificados de los contadores de bucle. El conjunto de valores se especifica mediante un *nombre de modo discreto* (enumeración por intervalo), o mediante un valor conjuntista (enumeración conjuntista), o mediante un valor inicial, un valor de paso y un valor final (enumeración por paso).

El contador de bucle define implícitamente un nombre que denota su valor o localización dentro de la lista de sentencias de acción.

enumeración por intervalo:

En el caso de enumeración por intervalo, sin (con) especificación **DOWN**, el valor inicial del contador de bucle es el valor mínimo (máximo) del conjunto de valores definido por el *nombre de modo discreto*. Para las ejecuciones ulteriores de la lista de sentencias de acción, el *valor siguiente* se evaluará como:

SUCC(valor anterior) (PRED(valor anterior)).

La terminación se produce si la lista de sentencias de acción se ha ejecutado para el valor máximo (mínimo) definido por el *nombre de modo discreto*.

enumeración conjuntista:

En el caso de una enumeración conjuntista sin (con) la especificación **DOWN**, el valor inicial del contador de bucle es el valor miembro mínimo (máximo) del valor conjuntista denotado. Si el valor conjuntista es vacío, no se ejecutará la lista de sentencias de acción. Para las ejecuciones ulteriores de la lista de sentencias de acción, el valor siguiente será el siguiente mayor (menor) valor miembro del valor conjuntista. Se termina la acción hacer cuando se ha ejecutado la lista de sentencias de acción para el valor máximo (mínimo). Cuando se ejecuta la acción hacer, la expresión **conjuntista** se evalúa solamente una vez.

enumeración por paso:

En el caso de una enumeración por paso sin (con) la especificación **DOWN**, el conjunto de valores del contador de bucle se determina mediante un valor inicial, un valor final y posiblemente un valor de paso. Cuando se ejecuta la acción hacer, estas expresiones se evalúan una sola vez en un orden no especificado, posiblemente mixto. El valor de paso es siempre positivo. Antes de ejecutar la lista de sentencias de acción, se comprueba la terminación. Primeramente se verifica si el valor inicial del contador de bucle es superior (inferior) al valor final. Para las ejecuciones ulteriores, el *valor siguiente* se evalúa como:

valor anterior + valor de paso (valor anterior – valor de paso)

en caso de especificación *de valor de paso*; en otro caso como:

SUCC(valor anterior) (PRED(valor anterior)).

La terminación se produce si la evaluación proporciona un valor mayor (menor) que el valor final o provocaría una excepción *OVERFLOW*.

3) **enumeración de localización**:

En el caso de una enumeración de localización sin (con) la especificación **DOWN**, se entra repetidamente en la lista de sentencias de acción para un conjunto de localizaciones, que son los elementos de la localización matriz designada por *localización matriz* o los componentes de la localización cadena denotada por *localización cadena*. Si se especifica una *expresión matriz* o *cadena* que no sea una localización, se creará implícitamente una localización que contendrá el valor especificado. El tiempo de vida de la localización creada es la acción hacer. El modo de la localización creada es dinámico si el valor tiene una clase dinámica. La semántica es equivalente a la que existiría si se encontrara inicialmente la declaración de identidad-loc:

DCL <*contador de bucle*> <*modo*> **LOC** := <*objeto compuesto*> *(*<*índice*>*)*;

donde *modo* es el modo elemento de la localización matriz, o *&nombre(1)* tal que *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo de la localización cadena, si es un modo cadena **fijo**, y en otro caso del modo **componente**, y donde *índice* está puesto inicialmente al **límite inferior** (**límite superior**) del modo de localización e *índice* antes de cada ejecución subsiguiente de la lista de sentencias de acción se pone a *SUCC* (*índice*) (*PRED* (*índice*)). La lista de sentencias de acción no se ejecutará si la **longitud efectiva** de la *longitud cadena* es igual a 0. La acción hacer se termina (terminación normal) si *índice* justamente después de una ejecución de la lista de sentencias de acción es igual al **límite superior** (**límite inferior**) del modo de localización. Cuando se ejecuta la acción hacer, *el objeto compuesto* se evalúa una sola vez.

propiedades estáticas: Un contador de bucle tiene asociada una cadena de nombre que es la cadena de nombre de su *ocurrencia de definición*.

enumeración de valor: El nombre definido por el *contador de bucle* es un nombre de **enumeración de valor**.

enumeración por paso: La clase del nombre definido por un *contador de bucle* es la **clase resultante** de las clases del *valor inicial*, del *valor de paso*, si existe, y del *valor final*.

enumeración por intervalo: La clase del nombre definido por el *contador de bucle* es la clase M-valuada, donde M es el *nombre de modo discreto*.

enumeración conjuntista: La clase del nombre definido por el *contador de bucle* es la clase M-valuada, donde M es el modo miembro del modo de la *expresión conjuntista* (**fuerte**).

enumeración de localización: El nombre definido por el *contador de bucle* es un nombre de **enumeración de localización**. Su modo es el modo **elemento** del modo de la *localización matriz* o *expresión matriz*, o el modo cadena *&nombre(1)*, donde *&nombre* es un nombre de **sínmodo** virtual, **sinónimo** del modo de la *localización cadena* o el modo **raíz** de *la expresión cadena*.

Un nombre de **enumeración de localización** es **referenciable** si la organización de elemento del modo de la *localización matriz* es **NOPACK**.

condiciones estáticas: Las clases del *valor inicial*, del *valor final* y del *valor de paso*, si existe, deben ser **compatibles** por pares.

El modo **raíz** de la clase de un *contador de bucle* en una *enumeración de valor* no debe ser un modo conjunto **numerado**.

Si el modo **raíz** de la clase de un *contador de bucle* es un modo entero, debe existir un modo entero **predefinido** que contiene todos los valores derivados por *valor inicial*, *valor final* y *valor de paso*, si están presentes.

condiciones dinámicas: Se produce una excepción *RANGEFAIL* si el valor proporcionado por el *valor de paso* no es mayor que 0. Esta excepción se produce fuera del bloque de la acción hacer.

ejemplos:

4.17 **FOR** *i := 1* **TO** *c* *(1.1)*

15.37 **FOR EVER** *(1.1)*

4.17 *i := 1* **TO** *c* *(3.1)*

9.12 *j := MIN (sieve)* **BY** *MIN (sieve)* **TO** *max* *(3.1)*

14.28 *i* **IN** *INT (1:100)* *(3.2)*

6.5.3 Control mientras

sintaxis:

<control mientras> ::= (1)
WHILE *<expresión booleana>* (*1.1*)

semántica: Se evalúa la expresión booleana inmediatamente antes de entrar en la lista de sentencias de acción (tras la evaluación del control para, si existe). Si el resultado es *TRUE*, se entra en la lista de sentencias de acción; en otro caso, se termina la acción hacer.

ejemplo:

7.35 **WHILE** *n >= 1* *(1.1)*

6.5.4 Parte con

sintaxis:

<parte con> ::= *(1)*
WITH *<control con>* {, *<control con>* }* *(1.1)*

<control con> ::= *(2)*
<localización estructura> *(2.1)*
| *<valor primitivo de estructura>* *(2.2)*

NOTA – Si el *control con* es una *localización estructura*, la ambigüedad sintáctica se resuelve interpretando *control con* como una *localización* y no como un *valor primitivo*.

semántica: Los nombres de campo (**visibles**) del modo de las localizaciones estructura o valores de estructura especificados en cada *control con* se hacen disponibles como accesos directos a los campos.

Las reglas de visibilidad son las mismas que si se introdujera una ocurrencia de definición de nombre de campo para cada nombre **de campo** asociado al modo de la localización o valor primitivo y con la misma cadena de nombre que el nombre de campo.

Si se especifica una *localización estructura*, se declaran implícitamente nombres de acceso con la misma cadena de nombre que los nombres de campo del modo de la *localización estructura*, los cuales designan las sublocalizaciones de la localización estructura.

Si se especifica un *valor primitivo de estructura*, se definen implícitamente nombres de valor con la misma cadena de nombre que los nombres de campo del modo del *valor primitivo de estructura* (**fuerte**), los cuales designan los subvalores del valor estructura.

Cuando se entra en la acción hacer, las localizaciones estructura y/o los valores estructura especificados se evalúan una sola vez, al entrar la acción hacer, en cualquier orden.

propiedades estáticas: La ocurrencia de definición (virtual) introducida para un nombre de **campo** tiene la misma cadena de nombre que la *ocurrencia de definición de nombre de campo* de ese nombre de **campo**.

Si se especifica un *valor primitivo de estructura*, una ocurrencia de definición (virtual) en una *parte con* define un nombre de **valor hacer-con**. Su clase es la clase M-valuada, siendo M el modo de ese nombre de **campo** del modo estructura del *valor primitivo de estructura*, que se pone a disposición como un nombre de **valor hacer-con**.

Si se especifica una *localización estructura*, una ocurrencia de definición (virtual) en una *parte con* define un nombre de **localización hacer-con**. Su modo es el modo del nombre de **campo** del modo de la *localización estructura*, que se pone a disposición como un nombre de **localización hacer-con**. Un nombre de **localización hacer-con** es **referenciable** si la organización de campo del nombre de **campo** asociado es **NOPACK**.

ejemplo:

15.58 **WITH** *each* *(1.1)*

6.6 Acción salir

sintaxis:

<acción salir> ::= *(1)*
EXIT *<nombre de etiqueta>* *(1.1)*

semántica: Una acción de salir se utiliza para abandonar una sentencia de acción encorchetada o un módulo. La ejecución se reanuda inmediatamente después de la sentencia de acción encorchetada circundante más próxima o del módulo etiquetado con el *nombre de etiqueta*.

condiciones estáticas: La *acción salir* debe estar dentro de la sentencia de acción encorchetada o el módulo para el cual la *ocurrencia de definición* precedente tiene la misma cadena de nombre que el *nombre de etiqueta*.

Si la *acción salir* se coloca dentro de una definición de procedimiento o definición de proceso, la sentencia de acción encorchetada o módulo de que se sale tiene que estar dentro de la definición del mismo procedimiento o proceso (es decir, la acción salir no puede utilizarse para abandonar procedimientos o procesos).

No puede agregarse ningún *manejador* a una *acción salir*.

ejemplo:

15.62 **EXIT** *find_counter* *(1.1)*

6.7 Acción llamar

sintaxis:

<acción llamar> ::= *(1)*
<llamada a procedimiento> *(1.1)*
| *<llamada a rutina incorporada>* *(1.2)*
| *<llamada a procedimiento componente moreta>* *(1.3)*

<llamada a procedimiento> ::= *(2)*
{ *<nombre de procedimiento>* | *<valor primitivo de procedimiento>* }
([*<lista de parámetros efectivos>*]) *(2.1)*

<lista de parámetros efectivos> ::= *(3)*
<parámetro efectivo> { , *<parámetro efectivo>* }* *(3.1)*

<parámetro efectivo> ::= *(4)*
<valor> *(4.1)*
| *<localización>* *(4.2)*

<llamada a rutina incorporada> ::= *(5)*
<nombre de rutina incorporada> ([*<lista de parámetros de rutina incorporada>*]) *(5.1)*

<lista de parámetros de rutina incorporada> ::= *(6)*
<parámetro de rutina incorporada> { , *<parámetro de rutina incorporada>* }* *(6.1)*

<parámetro de rutina incorporada> ::= *(7)*
<valor> *(7.1)*
| *<localización>* *(7.2)*
| *<nombre no reservado>* [(*<lista de parámetros de rutina incorporada>*)] *(7.3)*

<llamada a procedimiento componente moreta> ::= *(8)*
<localización moreta> . *<llamada a procedimiento componente moreta>*
[*<prioridad>*] *(8.1)*
| *<valor primitivo localización moreta de referencia ligada >* –> **.**
| *<llamada a procedimiento componente moreta>* [*<prioridad>*] *(8.2)*
| *<llamada a procedimiento componente moreta>* [*<prioridad>*] *(8.3)*

NOTA – Si el *parámetro efectivo* o el *parámetro de rutina incorporada* es una *localización*, la ambigüedad sintáctica se suprime interpretándolo como una *localización* y no como un *valor*.

sintaxis derivada: Una llamada a procedimiento **P(...)** de un *procedimiento componente moreta* **P** es una sintaxis derivada para **SELF.P(...)**.

semántica: Una acción llamar provoca la llamada a un procedimiento o a una rutina incorporada, o a un procedimiento componente moreta. Una llamada a procedimiento provoca una llamada del procedimiento **general** indicado por el valor proporcionado por el *valor primitivo de procedimiento* o el procedimiento indicado por el *nombre de procedimiento*. Una *llamada a procedimiento componente moreta* **L.name(...)** provoca la llamada de ese procedimiento componente moreta que se identifica por nombre en el modo de **L**. **L** se pasa como un parámetro localización inicial al procedimiento. Los valores efectivos y las localizaciones especificadas en la lista de parámetros efectivos se transfieren al procedimiento.

Una *llamada a rutina incorporada* es una *llamada a rutina incorporada CHILL* o una *llamada a rutina incorporada* en la implementación (véanse 6.20 y 13.1 respectivamente).

Un valor, una localización o cualquier nombre definido en un programa que no sea una cadena de nombre simple **reservada**, puede transferirse como *parámetro de rutina incorporada*. La llamada a rutina incorporada puede devolver un valor o una localización.

Una rutina incorporada puede ser genérica, es decir, su clase (si es una llamada a rutina incorporada que entrega un **valor**) o su modo (si es una llamada a rutina incorporada que entrega una **localización**) puede depender no solamente del *nombre de rutina incorporada*, sino también de las propiedades estáticas de los parámetros efectivos transmitidos y del contexto estático de la llamada.

Una llamada a procedimiento componente moreta tiene siempre la estructura "localización . llamada a procedimiento". Se caracteriza por la expresión "la llamada a procedimiento se aplica a la localización".

Para una llamada a procedimiento componente moreta se efectúan los siguientes pasos:

a) *El procedimiento llamado se aplica a una localización de modo módulo:*
   1) evaluación de los parámetros efectivos
   2) verificación de la precondición
   3) verificación de la invariable completa
   4) ejecución del cuerpo del procedimiento
   5) verificación de la invariable completa
   6) verificación de la poscondición completa
   7) retorno al punto llamante.

b) *El procedimiento llamado se aplica a una localización de modo región RL:*
   1) evaluación de los parámetros efectivos
   2) esperar hasta que RL esté libre y bloquear RL
   3) verificación de la precondición
   4) verificación de la invariable completa
   5) ejecución del cuerpo del procedimiento
   6) verificación de la invariable completa
   7) verificación de la poscondición completa
   8) liberación de RL
   9) retorno al punto llamante.

c) *El procedimiento llamado se aplica a una localización de modo tarea TL:*

   el llamante efectúa los siguientes pasos:
   1) evaluación de los parámetros efectivos
   2) enviar identificación de procedimiento, parámetros efectivos y prioridad a TL
   3) continuar con la acción siguiente

   TL efectúa los siguientes pasos:
   1) recibir identificación de procedimiento y parámetros efectivos de acuerdo con la prioridad
   2) verificación de la precondición
   3) verificación de la invariable completa
   4) ejecución del cuerpo del procedimiento
   5) verificación de la invariable completa
   6) verificación de la poscondición completa.

propiedades estáticas: Una *llamada a procedimiento* tiene asociadas las siguientes propiedades: una lista de **especs de parámetro**, posiblemente una **espec de resultado**, un conjunto posiblemente vacío de nombres de excepción, una **generalidad**, una **recursividad**, y posiblemente sea **intrarregional** (esto último sólo es posible con un *nombre de procedimiento*, véase 11.2.2). Estas propiedades se heredan del *nombre de procedimiento* o de cualquier modo **compatible** con la clase del *valor primitivo de procedimiento* (en este último caso, la generalidad es siempre **general**).

Un *llamada a procedimiento*, con una **espec de resultado** es una *llamada a procedimiento que entrega una localización* si y sólo si se especifica **LOC** en la **espec de resultado**; en otro caso, es una *llamada a procedimiento que entrega un valor*.

Un *nombre de rutina incorporada* es un nombre CHILL o definido por la implementación, que se considera definido en el dominio de la definición de proceso imaginario más exterior o en cualquier contexto (véase 10.8).

Una *llamada a rutina incorporada* es una *llamada a rutina incorporada* que entrega una **localización** si entrega una localización; es una *llamada a rutina incorporada* que entrega **un valor** si entrega un valor.

condiciones estáticas: Una prioridad sólo puede aplicarse en una llamada de un procedimiento aplicado a una localización de tarea.

El número de ocurrencias de *parámetro efectivo* en la *llamada a procedimiento* debe ser el mismo que el de sus especs de parámetro. Los requisitos de compatibilidad para el *parámetro efectivo* y la correspondiente espec de parámetro (en posición) de *la llamada a procedimiento* son:

- Si la espec de parámetro tiene el atributo **IN** (por defecto), el *parámetro efectivo* debe ser un *valor* cuya clase sea **compatible** con el modo de la correspondiente espec de parámetro. Este último modo no debe tener la **propiedad de no-valor**. El *parámetro efectivo* es un *valor* que debe ser **regionalmente seguro** para *la llamada a procedimiento*.
- Si la espec de parámetro tiene el atributo **INOUT** o **OUT**, el *parámetro efectivo* debe ser una *localización* cuyo modo debe ser **compatible** con la clase M-valuada, siendo M el modo de la correspondiente espec de parámetro. El modo de la *localización* (efectiva) debe ser estático, y no debe tener la **propiedad de lectura solamente** ni la **propiedad de no-valor**. El *parámetro efectivo* es una *localización*. Puede percibirse como un *valor* que debe ser **regionalmente seguro** para la *llamada a procedimiento*.
- Si la espec de parámetro tiene el atributo **INOUT**, el modo de la espec de parámetro debe ser **compatible** con la clase M-valuada, donde M es el modo de la *localización*.
- Si la espec de parámetro tiene el atributo **LOC** especificado sin **DYNAMIC**, el *parámetro efectivo* debe ser una *localización* que sea **referenciable** y para la cual el modo de la espec de parámetro sea de **lectura compatible** con el modo de la *localización* (efectiva), o el *parámetro efectivo* debe ser un *valor* que no sea una *localización*, pero cuya clase sea **compatible** con el modo de la espec de parámetro. Si el modo del parámetro formal es un modo moreta, el nombre de modo del parámetro formal y el nombre de modo del parámetro efectivo deben ser sinónimos. Si el modo del parámetro formal es de la forma "REF MM", donde MM es un modo moreta, el modo del parámetro formal y el modo del parámetro efectivo deben ser similares.
- Si la espec de parámetro tiene especificado el atributo **LOC** con **DYNAMIC**, el *parámetro efectivo* debe ser una *localización* que sea **referenciable** y para la cual el modo de la espec de parámetro sea de **lectura dinámica compatible** con el modo de la *localización* (efectiva), o el *parámetro efectivo* debe ser un *valor* que no sea una *localización*, pero cuya clase sea **compatible** con una versión parametrizada de este modo.
- Si la espec de parámetro tiene el atributo **LOC**, entonces:
  - si el *parámetro efectivo* es una *localización* debe tener la misma **regionalidad** que la *llamada a procedimiento*;
  - si el *parámetro efectivo* es un *valor* debe ser **regionalmente seguro** para la *llamada a procedimiento*.

condiciones dinámicas: Una *acción llamar* puede originar cualquier excepción del conjunto de nombres de excepción asociado. Una *llamada a procedimiento* produce la excepción *EMPTY* si el *valor primitivo procedimiento* entrega *NULL*. Una *acción llamar* causa la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento. Si la **recursividad** del procedimiento es **no recursiva**, el procedimiento no debe entonces llamarse a sí mismo directa ni indirectamente.

La transferencia de parámetro puede originar las siguientes excepciones:

- Si la espec de parámetro tiene el atributo **IN** o **INOUT**, se aplican en el punto de llamada las condiciones de asignación del valor (efectivo) con respecto al modo de la espec de parámetro, produciéndose las posibles excepciones antes de llamar al procedimiento (véase 6.2).
- Si la espec de parámetro tiene el atributo **INOUT** u **OUT**, se aplican en el punto de retorno, las condiciones de asignación del valor local del parámetro formal con respecto al modo de la localización (efectiva), produciéndose las posibles excepciones después del retorno del procedimiento (véase 6.2).
- Si la espec de parámetro tiene el atributo **LOC** y el *parámetro efectivo* es un *valor* distinto de una *localización*, se aplican en el punto de llamada las condiciones de asignación del *valor* (efectivo) con respecto al modo de la espec de parámetro, produciéndose las posibles excepciones antes de llamar al procedimiento (véase 6.2).

La verificación de afirmación puede originar las siguientes excepciones:

- Si la precondición evalúa a *FALSE* se origina la excepción *PREFAIL*. La búsqueda de un manejador apropiado comienza al final del cuerpo del procedimiento y continúa de acuerdo con 8.3.
- Si la poscondición evalúa a *FALSE* se origina la excepción *POSFAIL*. La búsqueda de un manejador apropiado comienza al final del cuerpo del procedimiento y continúa de acuerdo con 8.3.
- Si la invariable evalúa a *FALSE* se origina la excepción *INVFAIL*. La búsqueda de un manejador apropiado comienza al final del cuerpo del correspondiente modo moreta y continúa de acuerdo con 8.3.

El *valor primitivo de procedimiento* no debe proporcionar un procedimiento definido dentro de una definición de proceso cuya activación no sea la misma que la activación del proceso que ejecuta la llamada a procedimiento (distinto del proceso imaginario más externo), y el tiempo de vida del procedimiento designado no debe haber concluido.

Si una llamada se aplica a una localización de tarea TL, TL no debe ser terminada.

ejemplo:

4.18 op(a,b,d,order–1) *(1.1)*

6.8 Acción resultar y acción retornar

sintaxis:

<acción retornar> ::= *(1)*
RETURN [*<resultado>*] *(1.1)*

<acción resultar> ::= *(2)*
RESULT *<resultado>* *(2.1)*

<resultado> ::= *(3)*
<valor> *(3.1)*
| *<localización>* *(3.2)*

sintaxis derivada: La *acción retornar* con *resultado* se deriva de **DO RESULT** *<resultado>*; **RETURN**; **OD**.

semántica: Una acción resultar sirve para establecer el resultado que ha de ser entregado por una llamada a procedimiento. Este resultado puede ser una localización o un valor. Una acción retornar produce el retorno desde la invocación del procedimiento en cuya definición se ha colocado. Si el procedimiento retorna un resultado, éste viene determinado por la última acción resultar ejecutada. Si no se ha ejecutado ninguna acción resultar, la llamada a procedimiento proporciona, respectivamente, una localización **indefinida** o un valor **indefinido**.

propiedades estáticas: Una *acción resultar* y una *acción retornar* tienen asociado un nombre de **procedimiento**, que es el nombre de la definición de procedimiento circundante más próxima.

condiciones estáticas: Una *acción retornar* y una *acción resultar* deben estar circundadas textualmente por una definición de procedimiento. Una *acción resultar* sólo puede especificarse si su nombre de **procedimiento** tiene una **spec de resultado**.

A una *acción retornar* (sin *resultado*) no debe agregársele un *manejador*.

Si se especifica **LOC** (**LOC DYNAMIC**) en la **spec de resultado** del nombre de **procedimiento** de la *acción resultar*, el *resultado* debe ser una *localización* tal que el modo en la **spec de resultado** sea de **lectura compatible** (de **lectura dinámica compatible**) con el modo de la *localización*. La *localización* debe ser **referenciable** si no se especifica **NONREF** en la **spec de resultado**. El *resultado* es una *localización* que debe tener la misma **regionalidad** que el nombre de **procedimiento** asociado a la *acción resultar*.

Si no se especifica **LOC** en la **spec de resultado** del nombre de **procedimiento** de la *acción resultar*, el *resultado* debe ser un valor cuya clase sea **compatible** con el modo de la **spec de resultado**. El *resultado* es un *valor* que debe ser **regionalmente seguro** para el nombre de **procedimiento** asociado a la *acción resultar*.

condiciones dinámicas: Si no se especifica **LOC** en la **spec de resultado** del nombre de **procedimiento**, se aplican las condiciones de asignación del *valor* en la *acción resultar* con respecto al modo de la **spec de resultado** de su nombre de **procedimiento**.

ejemplos:

4.21 **RETURN** *(1.1)*

1.6 **RESULT** *i+j* *(2.1)*

5.19 c *(3.1)*

6.9 Acción ir a

sintaxis:

<acción ir a> ::= *(1)*
GOTO *<nombre de etiqueta>* *(1.1)*

semántica: Una acción ir a produce una transferencia de control. La ejecución se reanuda con la sentencia de acción etiquetada con el *nombre de etiqueta*.

condiciones estáticas: Si una *acción ir a* está situada dentro de una definición de procedimiento o definición de proceso, la etiqueta indicada por el *nombre de etiqueta* debe estar también definida dentro de la definición (es decir, no es posible saltar fuera de una invocación de procedimiento o de proceso).

A una *acción ir a* no debe agregársele un *manejador*.

6.10 Acción afirmar

sintaxis:

<acción afirmar> ::= *(1)*
ASSERT *<expresión booleana>* *(1.1)*

semántica: Una acción afirmar proporciona un medio de verificar una condición.

condiciones dinámicas: Se produce la excepción *ASSERTFAIL* si la *expresión booleana* entrega *FALSE*.

ejemplo:

4.7 **ASSERT** *b>0* **AND** *c>0* **AND** *order>0* *(1.1)*

6.11 Acción vacía

sintaxis:

<acción vacía> ::= *(1)*
<vacía> *(1.1)*

<vacía> ::= *(2)*

semántica: Una acción vacía no causa ninguna acción.

condiciones estáticas: A una *acción vacía* no debe agregársele un *manejador*.

6.12 Acción causar

sintaxis:

<acción causar> ::= *(1)*
CAUSE *<nombre de excepción>* *(1.1)*

semántica: Una acción causar produce la excepción cuyo nombre viene indicado por *nombre de excepción*.

condiciones estáticas: A una *acción causar* no debe agregársele un *manejador*.

ejemplo:

4.9 **CAUSE** *wrong_input* *(1.1)*

6.13 Acción arrancar

sintaxis:

<acción arrancar> ::= *(1)*
<expresión arrancar> *(1.1)*

semántica: Una acción arrancar evalúa la expresión arrancar (véase 5.2.15) sin utilizar el valor de ejemplar resultante.

ejemplo:

14.45 **START** *call_distributor ( )* *(1.1)*

6.14 Acción parar

sintaxis:

<acción parar> ::= *(1)*
STOP *(1.1)*

semántica: Una acción parar termina el proceso que la ejecuta (véase 11.1).

condiciones estáticas: A una *acción parar* no debe agregársele un *manejador*.

6.15 Acción continuar

sintaxis:

<acción continuar> ::= *(1)*
CONTINUE *<localización suceso>* *(1.1)*

semántica: Una acción continuar evalúa la *localización suceso*.

Si la localización suceso tiene asociado un conjunto no vacío de procesos demorados, se reactivará de entre esos procesos el de mayor prioridad. Si hay varios procesos así, se seleccionará uno de una manera definida por la implementación. Si no hay tales procesos, la acción continuar no produce ningún otro efecto.

Si un proceso es reactivado, se le retira de todos los conjuntos de procesos demorados a los que pertenezca.

ejemplo:

13.25 **CONTINUE** *resource_freed* *(1.1)*

6.16 Acción demorar

sintaxis:

<acción demorar> ::= *(1)*
DELAY *<localización evento>* [*<prioridad>*] *(1.1)*

<prioridad> ::= *(2)*
PRIORITY *<expresión literal entera>* *(2.1)*

semántica: Una acción demorar evalúa la *localización evento*.

Se produce entonces una excepción *DELAYFAIL* (véase más adelante) o el proceso ejecutante es demorado.

Si el proceso ejecutante es demorado, pasa a ser miembro prioritario del conjunto de procesos demorados, asociado a la localización evento especificada. La prioridad es la especificada, si existe; en otro caso es 0 (la más baja).

propiedades dinámicas: Un proceso que ejecuta una acción demorar se vuelve **temporizable** cuando alcanza el punto de ejecución en que puede demorarse. Deja de ser **temporizable** cuando sale de ese punto.

condiciones estáticas: La *expresión literal entera* no debe entregar un valor negativo.

condiciones dinámicas: Se produce la excepción *DELAYFAIL* si la *localización evento* tiene asociado un modo con una **longitud suceso** igual al número de procesos ya demorados sobre la localización evento.

El tiempo de vida de la *localización evento* no debe terminar mientras el proceso ejecutante está demorado sobre ella.

ejemplo:

13.18 **DELAY** *resource_freed* *(1.1)*

6.17 Acción demorar y elegir

sintaxis:

<acción demorar y elegir> ::= *(1)*
DELAY CASE [**SET** *<localización ejemplar>* [*<prioridad>*] ; |
<prioridad>;] { *<alternativa de demora>* }$^{+}$
ESAC *(1.1)*

<alternativa de demora> ::= *(2)*
(*<lista de eventos>*) : *<lista de sentencias de acción>* *(2.1)*

<lista de eventos> ::= *(3)*
<localización evento> { , *<localización evento>* }* *(3.1)*

semántica: Una acción demorar y elegir evalúa, en un orden no especificado y posiblemente mixto, la *localización ejemplar*, si está presente, y todas las *localizaciones evento* especificadas en una *alternativa de demora*.

Se produce entonces una excepción *DELAYFAIL* (véase más adelante) o el proceso ejecutante es demorado.

Si el proceso ejecutante es demorado, pasa a ser miembro prioritario del conjunto de procesos demorados, asociado a cada una de las localizaciones suceso especificadas. La prioridad es la especificada, si existe; en otro caso es 0 (la más baja).

Si el proceso demorado es reactivado por otro proceso ejecutando una acción continuar sobre una localización evento, se entra en la correspondiente *lista de sentencias de acción*. Si varias *alternativas de demora* especifican la misma localización evento, la elección entre ellas no se especifica. Antes de la entrada, si se especifica una *localización ejemplar*, se almacena en la misma el valor de ejemplar que identifica el proceso que ha ejecutado la acción continuar.

propiedades dinámicas: Un proceso que ejecuta una acción demorar y elegir se vuelve **temporizable** cuando alcanza el punto de ejecución en que puede demorarse. Deja de ser **temporizable** cuando sale de ese punto.

condiciones estáticas: El modo de la *localización ejemplar*, no debe tener la **propiedad de lectura solamente**. La *expresión literal entera* en *prioridad* no debe entregar un valor negativo.

condiciones dinámicas: Se produce la excepción *DELAYFAIL* si cualquier *localización evento* tiene un modo con una **longitud de evento** asociada que es igual al número de procesos ya demorados sobre esa localización de evento.

No debe terminar el tiempo de vida de ninguna de las *localizaciones evento* mientras el proceso ejecutante está demorado sobre ellas.

Se produce la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplo:

```
14.26   DELAY CASE
            (operator_is_ready): /* some actions */
            (switch_is_closed):          DO FOR i IN INT (1:100);
                    CONTINUE operator_is_ready;
                    /* empty the queue */
            OD;
        ESAC                                                    (1.1)
```

6.18 Acción enviar

6.18.1 Generalidades

sintaxis:

```
<acción enviar> ::=                                             (1)
        <acción enviar señal>                                   (1.1)
    |   <acción enviar tampón>                                  (1.2)
```

semántica: Una acción enviar inicia la transferencia de información de sincronización desde un proceso remitente. La semántica detallada depende de que el objeto de sincronización sea una señal o un tampón.

6.18.2 Acción enviar señal

sintaxis:

```
<acción enviar señal> ::=                                       (1)
        SEND <nombre de señal> [ (<valor> { , <valor> } *) ]
        TO <valor primitivo de ejemplar> [ <prioridad> ]        (1.1)
```

semántica: Una acción enviar señal evalúa, en cualquier orden, la lista de *valores*, si está presente, y el *valor primitivo ejemplar*.

La señal especificada por *nombre de señal* se forma, para la transmisión, por los valores especificados y una prioridad. La prioridad es la especificada, si existe; en otro caso es 0 (la más baja).

Si el nombre de **señal** tiene asociado un nombre de **proceso**, sólo los procesos de ese nombre pueden recibir la señal; si se especifica un *valor primitivo ejemplar*, sólo ese proceso puede recibir la señal.

Si la señal tiene asociado un conjunto no vacío de procesos demorados, uno o más de los cuales pueden recibir la señal, se reactivará uno de ellos. Si hay varios procesos así, se seleccionará uno de una manera definida por la implementación. Si no hay tales procesos, la señal quedará pendiente.

Si un proceso es reactivado se retira de todos los conjuntos de procesos demorados a los que pertenezca.

condiciones estáticas: El número de ocurrencias de *valor* debe ser igual al número de modos del *nombre de señal*. La clase de cada *valor* debe ser **compatible** con el modo correspondiente del *nombre de señal*. Ninguna ocurrencia de *valor* puede ser **intrarregional** (véase 11.2.2). La *expresión literal entera* en *prioridad* no debe entregar un valor negativo.

condiciones dinámicas: Se aplican las condiciones de asignación de cada *valor* con respecto a su modo correspondiente del *nombre de señal*.

Se produce la excepción *EMPTY* si el *valor primitivo ejemplar* entrega *NULL*.

El tiempo de vida del proceso indicado por el valor proporcionado por el *valor primitivo ejemplar* no debe haber terminado en el momento de la ejecución de la acción enviar señal.

Se produce la excepción *SENDFAIL* si el *nombre de señal* tiene asociado un nombre de **proceso** que no es el indicado nombre del proceso por el valor proporcionado por el *valor primitivo de ejemplar*.

ejemplos:

15.78 **SEND** *ready* **TO** *received_user* *(1.1)*

15.86 **SEND** *readout(count)* **TO** *user* *(1.1)*

6.18.3 Acción enviar tampón

sintaxis:

<acción enviar tampón> ::= *(1)*
SEND *<localización tampón>* (*<valor>*) [*<prioridad>*] *(1.1)*

semántica: Una acción enviar tampón evalúa la *localización tampón* y el *valor* en cualquier orden.

Si la localización tampón tiene asociado un conjunto no vacío de procesos demorados, se reactivará uno de ellos. Si hay varios procesos así, se seleccionará uno de ellos de una manera definida por la implementación. Si no hay tales procesos y se ha rebasado la capacidad de la localización tampón, el proceso ejecutante es demorado, con una prioridad. En otro caso, el valor es almacenado con una prioridad. La prioridad es la especificada, si existe, en otro caso es 0 (la más baja). La capacidad del tampón es rebasada si la *localización tampón* tiene asociado un modo con una **longitud de tampón** igual al número de valores ya almacenados en la localización tampón.

Si el proceso ejecutante es demorado, pasa a ser miembro del conjunto de procesos remitentes demorados, asociado a la localización tampón. Si un proceso es reactivado, se retira de todos los conjuntos de procesos demorados a los que pertenezca.

propiedades dinámicas: Un proceso que ejecuta una acción enviar tampón se vuelve **temporizable** cuando alcanza el punto de ejecución en que puede ser demorado. Deja de ser **temporizable** cuando sale de ese punto.

condiciones estáticas: La clase del *valor* debe ser **compatible** con el modo **elemento tampón** del modo de la *localización tampón*. El *valor* no debe ser **intrarregional** (véase 11.2.2). La *expresión literal entera* en *prioridad* no debe entregar un valor negativo.

condiciones dinámicas: Se aplican las condiciones de asignación de *valor* con respecto al modo **elemento tampón** del modo de la *localización tampón*; las posibles excepciones se producen antes de que pueda demorarse el proceso.

El tiempo de vida de la *localización tampón* no debe terminar mientras el proceso ejecutante esté demorado sobre ella.

ejemplo:

16.123 **SEND** *user–>* ([*ready*, *–>counter_buffer*]) ; *(1.1)*

6.19 Acción recibir y elegir

6.19.1 Generalidades

sintaxis:

<acción recibir y elegir> ::= *(1)*
<acción recibir señal y elegir> *(1.1)*
| *<acción recibir tampón y elegir>* *(1.2)*

semántica: Una acción recibir y elegir recibe información de sincronización transmitida por la acción enviar. La semántica detallada depende del objeto de sincronización utilizado, que es una señal o un tampón. La entrada de una acción recibir y elegir no produce necesariamente una demora del hilo ejecutante (véase la cláusula 11 para más detalle).

6.19.2 Acción recibir señal y elegir

sintaxis:

<acción recibir señal y elegir> ::= *(1)*
RECEIVE CASE [**SET** *<localización ejemplar>* ;]
{ *<alternativa recibir señal>* }$^+$
[**ELSE** *<lista de sentencias de acción>*] **ESAC** *(1.1)*
| **RECEIVE** [**SET** *<localización ejemplar>*]
(*<nombre de señal>* [**IN** *<lista de localizaciones>*]) *(1.2)*

<lista de localizaciones> ::= *(2)*
<localización> { , *<localización>*}* *(2.1)*

<alternativa recibir señal> ::= *(3)*
(*<nombre de señal>* [**IN** *<lista de ocurrencias de definición>*]) *:*
<lista de sentencias de acción> *(3.1)*

sintaxis derivada: La notación (1.2) es sintaxis derivada para
RECEIVE CASE [SET <*localización ejemplar***>** ;]
(*<nombre de señal>* [**IN** *<&nombre>*$_1$ ..., *<&nombre>*$_n$]):

<localización>$_1$:= *<&nombre>*$_1$ ..., *<localización>*$_n$:= *<&nombre>*$_n$; **ESAC**,

donde *<&nombre>*$_1$ ..., *<&nombre>*$_n$ son nombres de **valor a recibir** introducidos virtualmente, y

<localización>$_1$ ..., *<localización>*$_n$ son las *localizaciones* en la *lista de localizaciones*.

semántica: Una acción recibir señal y elegir evalúa la *localización ejemplar* , si está presente.

A continuación, el proceso ejecutante: recibe (inmediatamente) una señal o, si se especifica **ELSE**, entra en la correspondiente *lista de sentencias de acción*; en otro caso es demorado. El proceso ejecutante recibe inmediatamente una señal si hay un *nombre de señal* especificado en una *alternativa recibir señal* que está pendiente y corresponde a una señal que puede ser recibida por el proceso. Si puede recibirse más de una señal, se seleccionará la de mayor prioridad, de una manera definida por la implementación.

Si el proceso ejecutante es demorado, pasa a ser miembro del conjunto de los procesos demorados, asociado a cada una de las señales especificadas. Si el proceso demorado es reactivado por otro proceso que ejecuta una acción enviar señal, recibe una señal.

Si el proceso ejecutante recibe una señal, se entra en la correspondiente *lista de sentencias de acción*. Antes de la entrada, si se especifica una *localización ejemplar*, se almacena en la misma el valor de ejemplar que identifica el proceso que ha enviado la señal recibida. Si el nombre de **señal** de la señal recibida tiene asociada una lista de modos, se especifica una lista de nombres de **valor a recibir**; la señal transporta una lista de valores, y los nombres de **valor a recibir** denotan su valor correspondiente en la *lista de sentencias de acción*.

propiedades estáticas: Una *ocurrencia de definición* en la *lista de ocurrencias de definición* de una *alternativa recibir señal* define un nombre de **valor a recibir**. Su clase es la clase M-valuada, donde M es el modo correspondiente de la lista de modos ligados al *nombre de señal* que le precede.

propiedades dinámicas: Un proceso que ejecuta una acción recibir señal y elegir se vuelve **temporizable** cuando alcanza el punto de ejecución en que puede ser demorado. Deja de ser **temporizable** cuando sale de ese punto.

condiciones estáticas: El modo de la *localización ejemplar* no debe tener la **propiedad de lectura solamente**.

Todas las ocurrencias *de nombre de señal* deben ser diferentes.

La **IN** facultativa y la *lista de ocurrencias de definición* de la *alternativa recibir señal* deben especificarse si y sólo si el *nombre de señal* tiene un conjunto de modos no vacío. El número de nombres de la *lista de ocurrencias de definición* debe ser igual al número de modos del *nombre de señal*.

Se aplican las condiciones de asignación de los valores entregados por *&nombre*$_1$, ..., *&nombre*$_n$ con respecto a los modos de *localización*$_1$, ..., *localización*$_n$.

condiciones dinámicas: Se produce la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplo:

15.83 **RECEIVE CASE**
(advance): count + := 1;
(terminate):
SEND *readout(count)* **TO** *user;*
EXIT *work_loop;*
ESAC *(1.1)*

6.19.3 Acción recibir tampón y elegir

sintaxis:

<acción recibir tampón y elegir> ::= *(1)*
RECEIVE CASE [**SET** *<localización ejemplar>* ;]
{ *<alternativa recibir tampón>* }$^{+}$
[**ELSE** *<lista de sentencias de acción>*]
ESAC *(1.1)*
| **RECEIVE CASE** [**SET** *<localización ejemplar>*]
(*<localización tampón>* **IN** *<localización>*) *(1.2)*

<alternativa recibir tampón> ::= *(2)*
(*<localización tampón>* **IN** *<ocurrencia de definición>*) :
<lista de sentencias de acción> *(2.1)*

sintaxis derivada: La notación (1.2) es sintaxis derivada para
RECEIVE CASE [**SET** *<localización ejemplar>*;]
(*<localización tampón>* **IN** *<&nombre>*) : *<localización>* = *<&nombre>*;
donde *<&nombre>* es un nombre de **valor a recibir** introducido virtualmente.

semántica: Una acción recibir tampón y elegir evalúa, en cualquier orden la *localización ejemplar*, si está presente, y todas las *localizaciones tampón* especificadas en una *alternativa recibir tampón*.

A continuación, el proceso ejecutante: recibe (inmediatamente) un valor o, si se especifica **ELSE**, entra en la correspondiente *lista de sentencias de acción*; en otro caso es demorado. El proceso ejecutante recibe inmediatamente un valor si hay uno almacenado en una de las localizaciones tampón especificadas, o si hay un proceso remitente demorado sobre ella. Si puede recibirse más de un valor, se seleccionará uno de los de mayor prioridad, de una manera definida por la implementación.

Si el proceso ejecutante es demorado, pasa a ser miembro del conjunto de los procesos demorados, asociado a cada una de las localizaciones tampón especificadas. Si el proceso demorado es reactivado por otro proceso que ejecuta una acción enviar tampón, recibe un valor.

Si el proceso ejecutante recibe un valor, se entra en la correspondiente *lista de sentencias de acción*. Si varias *alternativas recibir tampón* especifican la misma localización tampón, la elección entre las mismas no se especifica. Antes de la entrada, si se especifica una *localización ejemplar*, se almacena en la misma el valor de ejemplar que identifica el proceso que ha enviado el valor recibido. El nombre de **valor a recibir** especificado denota el valor recibido en la *lista de sentencias de acción*.

Otro proceso es reactivado si el proceso ejecutante recibe un valor de una localización tampón cuyo conjunto asociado de procesos remitentes demorados no está vacío. El proceso reactivado es uno que tiene asociada la prioridad más elevada, si el valor recibido fue almacenado en la localización tampón; en otro caso, será reactivado el que envió el valor recibido. En el primer caso, el valor a enviar por el proceso reactivado se almacena en la localización tampón (cuya capacidad sigue siendo rebasada), y si puede reactivarse más de un proceso, se elegirá uno de ellos de una manera definida por la implementación. El proceso reactivado se retira del conjunto de procesos remitentes demorados, asociado a la localización tampón.

propiedades estáticas: Una *ocurrencia de definición* en una *alternativa recibir tampón* define un nombre de **valor a recibir**. Su clase es la clase M-valuada, donde M es el modo **elemento tampón** del modo de la *localización tampón* que etiqueta la *alternativa recibir tampón*.

propiedades dinámicas: Un proceso que ejecuta una acción recibir tampón y elegir se vuelve **temporizable** cuando alcanza el punto de ejecución en que puede ser demorado. Deja de ser **temporizable** cuando sale de ese punto.

condiciones estáticas: El modo de la *localización ejemplar* no debe tener la **propiedad de lectura solamente**.

Son aplicables las condiciones de asignación del valor denotado por *&name* con respecto al modo de la *localización*.

condiciones dinámicas: Se produce una excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

No debe terminar el tiempo de vida de ninguna de las *localizaciones tampón* mientras el proceso ejecutante está demorado sobre ellas.

6.20 Llamadas a rutina incorporada CHILL

sintaxis:

<llamada a rutina incorporada CHILL> ::= *(1)*
<llamada a rutina incorporada CHILL simple> *(1.1)*
| *<llamada a rutina incorporada CHILL que entrega una localización>* *(1.2)*
| *<llamada a rutina incorporada CHILL que entrega un valor>* *(1.3)*

nombres predefinidos: Los nombres de rutina incorporada CHILL están predefinidos como nombres de **rutina incorporada** (véase 6.7).

semántica: Una *llamada a rutina incorporada CHILL* es una *llamada a rutina incorporada CHILL simple*, que no entrega resultados (véase 6.20.1), una *llamada a rutina incorporada CHILL que entrega una localización* (véase 6.20.2), o una *llamada a rutina incorporada CHILL que entrega un valor* (véase 6.20.3).

propiedades estáticas: Una *llamada a rutina incorporada CHILL* es una *llamada a rutina incorporada* que entrega una **localización** si es una *llamada a rutina incorporada CHILL que entrega una localización*; es una *llamada a rutina incorporada* que entrega un **valor** si es una *llamada a rutina incorporada CHILL que entrega un valor*.

6.20.1 Llamadas a rutina incorporada CHILL simple

sintaxis:

<llamada a rutina incorporada CHILL simple> ::= *(1)*
<llamada a rutina incorporada terminar> *(1.1)*
| *<llamada a rutina incorporada simple de e/s>* *(1.2)*
| *<llamada a rutina incorporada simple que entrega una temporización>* *(1.3)*

semántica: Una *llamada a rutina incorporada CHILL simple* es una *llamada a rutina incorporada* que no entrega ni un valor ni una localización. Las rutinas incorporadas simples para entrada/salida se definen en la cláusula 7. Las rutinas incorporadas simples para temporización se describen en la cláusula 9.

6.20.2 Llamadas a rutina incorporada CHILL que entregan una localización

sintaxis:

<llamada a rutina incorporada CHILL que entrega una localización> ::= *(1)*
<llamada a rutina incorporada de e/s que entrega una localización> *(1.1)*

semántica: Una *llamada a rutina incorporada CHILL que entrega una localización* es una *llamada a rutina incorporada* que entrega una localización. Las rutinas incorporadas de entrada/salida que entregan una localización se definen en la cláusula 7.

6.20.3 Llamadas a rutina incorporada CHILL que entrega un valor

sintaxis:

<llamada a una rutina incorporada CHILL que entrega un valor> ::= *(1)*
NUM (*<expresión discreta>*) *(1.1)*
| *PRED* (*<expresión discreta>*) *(1.2)*
| *SUCC* (*<expresión discreta>*) *(1.3)*
| *ABS* (*<expresión entera>*) *(1.4)*
| *CARD* (*<expresión conjuntista>*) *(1.5)*
| *MAX* (*<expresión conjuntista>*) *(1.6)*
| *MIN* (*<expresión conjuntista>*) *(1.7)*
| *SIZE* ({*<localización>* | *<argumento de modo>*}) *(1.8)*
| *UPPER* (*<argumento de upper lower>*) *(1.9)*
| *LOWER* (*<argumento de upper lower>*) *(1.10)*
| *LENGTH* (*<argumento de longitud>*) *(1.11)*
| *<llamada a rutina incorporada atribuir>* *(1.12)*
| *<llamada a rutina incorporada de e/s que entrega un valor>* *(1.13)*
| *<llamada a rutina incorporada que entrega un valor de tiempo>* *(1.14)*
| *SIN* (*<expresión coma flotante>*) *(1.15)*
| *COS* (*<expresión coma flotante>*) *(1.16)*
| *TAN* (*<expresión coma flotante>*) *(1.17)*

| *ARCSIN* (<*expresión coma flotante*>) *(1.18)*
| *ARCCOS* (<*expresión coma flotante*>) *(1.19)*
| *ARCTAN* (<*expresión coma flotante*>) *(1.20)*
| *EXP* (<*expresión coma flotante*>) *(1.21)*
| *LN* (<*expresión coma flotante*>) *(1.22)*
| *LOG* (<*expresión coma flotante*>) *(1.23)*
| *SQRT* (<*expresión coma flotante*>) *(1.24)*

<*expresión numérica*> ::= *(2)*
<*expresión entera*>) *(2.1)*
| <*expresión coma flotante*> *(2.2)*

<*argumento de modo*> ::= *(3)*
<*nombre de modo*> *(3.1)*
| <*nombre de modo matriz*> (<*expresión*>) *(3.2)*
| <*nombre de modo cadena*> (<*expresión entera*>) *(3.3)*
| <*nombre de modo estructura variable*> (<*lista de expresiones*>) *(3.4)*

<*argumento de upper lower*> ::= *(4)*
<*localización matriz*> *(4.1)*
| <*expresión matriz*> *(4.2)*
| <*nombre de modo matriz*> *(4.3)*
| <*localización cadena*> *(4.4)*
| <*expresión cadena*> *(4.5)*
| <*nombre de modo cadena*> *(4.6)*
| <*localización discreta*> *(4.7)*
| <*expresión discreta*> *(4.8)*
| <*nombre de modo discreto*> *(4.9)*
| <*localización coma flotante* > *(4.10)*
| <*expresión coma flotante* > *(4.11)*
| <*nombre de modo coma flotante* > *(4.12)*
| <*localización acceso* > *(4.13)*
| <*nombre de modo acceso* > *(4.14)*
| <*localización texto* > *(4.15)*
| <*nombre de modo texto* > *(4.16)*

<*argumento de longitud*> ::= *(5)*
<*localización cadena*> *(5.1)*
| <*expresión cadena*> *(5.2)*
| <*nombre de modo cadena*> *(5.3)*
| <*localización evento*> *(5.4)*
| <*nombre de modo evento*> *(5.5)*
| <*localización tampón*> *(5.6)*
| <*nombre de modo tampón*> *(5.7)*
| <*localización texto*> *(5.8)*
| <*nombre de modo texto*> *(5.9)*

NOTA – Si el argumento de *upper lower* es una *localización matriz*, una *localización cadena*, una *localización discreta* o una *localización coma flotante*, la ambigüedad sintáctica se resuelve interpretando *argumento de upper lower* como una *localización*, y no como una *expresión* o *valor primitivo*. Si el *argumento de longitud* es una *localización cadena*, la ambigüedad sintáctica se resuelve interpretando *argumento de longitud* como una *localización*, y no como una *expresión*.

semántica: Una *llamada a rutina incorporada CHILL que entrega un valor* es una *llamada a rutina incorporada* que entrega un valor.

NUM entrega un valor entero con la misma representación interna que el valor entregado por su argumento.

PRED y *SUCC* entregan respectivamente el valor discreto más bajo y más alto siguientes de su argumento.

ABS se define sobre valores numéricos, esto es valores enteros y valores coma flotante, y entrega el valor absoluto correspondiente.

CARD, *MAX* y *MIN* se definen sobre valores conjuntistas. *CARD* entrega el número de valores de elemento de su argumento.

MAX y *MIN* entregan respectivamente los valores de elemento mayor y menor de su argumento.

SIZE se define para las localizaciones y los modos (posiblemente dinámicos) **referenciables**. En el primer caso, proporcionan el número de unidades de memoria direccionables ocupadas por esa localización; en el segundo caso, el número de unidades de memoria direccionables que ocupará una localización **referenciable** de ese modo. El modo es estático si el *argumento de modo* es un *nombre de modo*; en otro caso, es una versión dinámicamente parametrizada del mismo, con los parámetros especificados en el *argumento de modo*. En el primer caso, la *localización* no se evaluará en la ejecución.

UPPER y *LOWER* se definen (en forma posiblemente dinámica) por:

- localizaciones matriz, cadena y discretas, que entregan el **límite superior** y el **límite inferior** del modo de la localización,
- expresiones matriz y cadena, que entregan el **límite superior** y el **límite inferior** del modo de la clase del valor,
- expresiones discretas y coma flotante **fuertes**, que entregan el **límite superior** y el **límite inferior** del modo de la clase del valor,
- nombres de **modo** matriz, cadena, discreto, como flotante y texto que entregan el **límite superior** y el **límite inferior** del modo.

LENGHT se define sobre lo siguiente (posiblemente en forma dinámica):

- localizaciones de cadena y texto y expresiones de cadena que entregan el valor real de las mismas;
- localizaciones de evento que entregan la **longitud de evento** del modo de las localizaciones;
- localizaciones de tampón que entregan la **longitud de tampón** del modo de las localizaciones;
- nombres de **modo** cadena que entregan la **longitud de cadena** del modo;
- nombres de **modo** texto que entregan la **longitud de texto** del modo;
- nombres de **modo** tampón que entregan la **longitud de tampón** del modo.
- nombres de **modo** evento que entregan la **longitud de evento** del modo.

SIN entrega el seno de su argumento (interpretado en radianes).

COS entrega el coseno de su argumento (interpretado en radianes).

TAN entrega la tangente de su argumento (interpretada en radianes).

ARCSIN entrega la función sen^{-1} de su argumento en la gama $-\pi/2 : \pi/2$.

ARCCOS entrega la función $\cos^{-1}$ de su argumento en la gama $0 : \pi$.

ARCTAN entrega la función $\tan^{-1}$ de su argumento en la gama $-\pi/2 : \pi/2$.

EXP entrega la función e^x siendo x su argumento.

LN entrega el logaritmo natural de su argumento.

LOG entrega el logaritmo de base 10 de su argumento.

SQRT entrega la raíz cuadrada de su argumento.

Para la evaluación del resultado de *llamada a rutina incorporada* con argumentos **constantes** se aplican las mismas reglas que para *expresión* **constante** (véase 5.3.1).

propiedades estáticas: La clase de una llamada a rutina incorporada *NUM* es la clase *&INT*-derivada. La llamada a rutina incorporada es **constante** (**literal**) si y sólo si el argumento es **constante** (**literal**).

La clase de una llamada a rutina incorporada *PRED* o *SUCC* es la **clase resultante** del argumento. La llamada a rutina incorporada es **constante (literal)** si y sólo si el argumento es **contante** (**literal**).

La clase de una llamada a rutina incorporada *ABS* es la **clase resultante** de su argumento. La llamada a rutina incorporada es **constante (literal)** si y sólo si el argumento es **contante** (**literal**).

La clase de una llamada a rutina incorporada *CARD* es la clase *&INT*-derivada. La llamada a esta rutina incorporada es **constante** si y sólo si el argumento es **contante**.

La clase de una llamada a rutina incorporada *MAX* o *MIN* es la clase M-valuada, donde M es el modo **miembro** del modo de la *expresión conjuntista*. La llamada a rutina incorporada es **constante** si y sólo si el argumento es **contante**.

La clase de una llamada a rutina incorporada *SIZE* es la clase *&INT*-derivada. La llamada a rutina incorporada es **constante** si el modo del argumento es estático.

La clase de una llamada a rutina incorporada *UPPER* y *LOWER* es:

- la clase M-valuada, si *argumento de upper lower* es una *localización matriz*, *expresión matriz* o *nombre de modo matriz*, donde M es el modo **índice** de *localización matriz*, *expresión matriz* o *nombre de modo matriz*, respectivamente;
- la clase *&INT*-derivada, si *argumento de upper lower* es una *localización cadena*, *expresión cadena* o *nombre de modo cadena*;
- la clase M-valuada, si *argumento de upper lower* es una *localización discreta*, *expresión discreta* o *nombre de modo discreto*, donde M es el modo de *localización discreta* o el *modo de la expresión discreta* o el *nombre de modo discreto*, respectivamente.
- la clase M-valuada, si *argumento de upper lower* es una *localización coma flotante*, *expresión coma flotante* o *nombre de modo coma flotante*, donde M es el modo de *localización coma flotante*, *expresión coma flotante* o *nombre de modo coma flotante*, respectivamente;
- la clase M-valuada, si *argumento de upper lower* es una *localización acceso* o *nombre de modo acceso*, donde M es el modo **índice** del modo de la *localización acceso* o del *nombre de modo acceso*, respectivamente;
- la clase M-valuada, si *argumento de upper lower* es una *localización texto* o *nombre de modo texto*, donde M es el modo índice de la *localización texto* o el *nombre de modo texto*, respectivamente.

Una llamada a rutina incorporada *UPPER* o *LOWER* es **literal** si *argumento de upper lower* es un *nombre de modo matriz*, un *nombre de modo cadena*, un *nombre de modo discreto*, un *nombre de modo coma flotante*, un *nombre de modo acceso*, o un *nombre de modo texto*, si el modo de la *localización matriz* o *cadena* es estático, si la *expresión matriz* o *cadena* tiene una clase estática, o si *argumento de upper lower* es una *localización discreta* o una *expresión discreta*, una *localización coma flotante*, una *expresión coma flotante*, una *localización acceso*, o una *localización texto*.

La clase de una llamada a rutina incorporada *LENGTH* es la clase *&INT*-derivada. La llamada a rutina incorporada es **literal** si el *argumento longitud* es una *localización cadena* con un modo estático, una *expresión cadena* con una clase estática, una *localización evento*, una *localización tampón*, o si es un *nombre de modo cadena*, un *nombre de modo evento*, un *nombre de modo tampón*, o un *nombre de modo texto*.

La clase de una llamada a rutina incorporada *TAN, EXP, LN, LOG* o *SQRT* es la **clase resultante** de su argumento.

La clase de *SIN, COS, ARCSIN, ARCCOS, ARCTAN* es la 1. clase N-derivada, 2. clase N-valuada si la clase del argumento es 1. una clase N-derivada, 2. una clase N- valuada, donde N es un modo construido como sigue:

- para *SIN*: *&* **RANGE** (–1,0 : 1,0, *S*)
- para *COS*: *&* **RANGE** (–1,0 : 1,0, *S*)
- para *ARCSIN*: *&* **RANGE** (–π/2 : π/2, *S*)
- para *ARCCOS*: *&* **RANGE** (0 : π, *S*)
- para *ARCTAN*: *&* **RANGE** (–π/2 : π/2, *S*)

donde S es la **precisión** de N, y la **novedad** es la de N.

Una llamada a rutina incorporada *SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, EXP, LN, LOG* o *SQRT* es **constante** (**literal**) si y sólo si el argumento es **constante** (**literal**).

condiciones estáticas: Si el argumento de una llamada a rutina incorporada *PRED* o *SUCC* es **constante**, no debe entregar respectivamente los valores discretos más bajo o más alto definidos por el modo **raíz** de la clase del argumento. El modo **raíz** del argumento de *expresión discreta* de *PRED* y *SUCC* no debe ser un modo conjuntista **numerado**.

Si el argumento de una llamada a rutina incorporada *MAX* o *MIN* es **constante**, no debe entregar el valor conjuntista vacío.

El argumento de *localización* de *SIZE* debe ser **referenciable**.

La *expresión discreta* como argumentos de *UPPER* y *LOWER* deben ser **fuertes**.

Si el *argumento upper lower* es un *nombre de modo acceso* o una *localización acceso*, el modo acceso correspondiente debe tener un modo **índice**.

Si el *argumento upper lower* es un *nombre de modo texto* o una *localización texto*, el modo acceso correspondiente debe tener un modo **índice**.

Los siguientes requisitos de compatibilidad se cumplen para un *argumento de modo* que no es un *nombre de modo* simple.

- La clase de la *expresión* debe ser **compatible** con el modo **índice** del *nombre de modo matriz*.
- El *nombre de modo estructura variable* debe ser **parametrizable**, y debe haber tantas expresiones en la *lista de expresiones* como clases haya en su lista de clases; además, la clase de cada expresión debe ser **compatible** con la clase correspondiente en la lista de clases.

condiciones dinámicas: *PRED* y *SUCC* que no son **constantes** causan la excepción *OVERFLOW* si se aplican al valor discreto más bajo o más alto definido por el modo **raíz** de la clase del argumento.

NUM y *CARD* que no son **constantes** causan la excepción *OVERFLOW* si el valor resultante está fuera del conjunto de valores definidos por *&INT*.

MAX y *MIN* causan la excepción *EMPTY* si se aplican a valores conjuntistas vacíos.

ABS que no es **constante** causa la excepción *OVERFLOW* si el valor resultante está fuera de los límites definidos por el modo **raíz** de la clase del argumento.

Se produce la excepción *RANGEFAIL* si en el *argumento de modo*:

- la *expresión* entrega un valor que no pertenece al conjunto de valores definido por el modo **índice** del *nombre de modo matriz*;
- la *expresión entera* entrega un valor negativo o un valor igual o mayor que la **longitud de cadena** del *nombre de modo cadena*;
- cualquier expresión de la *lista de expresiones* para la que la clase correspondiente de la lista de clases del *nombre de modo estructura variable* es una clase M-valuada (es decir, es **fuerte**) entrega un valor que no pertenece al conjunto de valores definido por M.

ARCSIN y *ARCCOS* que no son **constantes** causan la excepción *OVERFLOW* si el argumento no está comprendido en el intervalo *–1,0* : *1,0*.

LN y *LOG* que no son **constantes** causan la excepción *OVERFLOW* si el argumento no es mayor que cero.

SQRT que no es **constante** causa la excepción *OVERFLOW* si el argumento no es superior o igual a cero.

SIN, COS, TAN, ARCSIN, ARCTAN, LN y *LOG* que no son **constantes** causan la excepción *OVERFLOW* si el valor resultante es mayor que el **límite superior** o menor que el **límite inferior** del modo **raíz** de la clase del argumento. En el caso de un valor resultante matemático exacto que es mayor que el **límite superior negativo** y menor que el **límite inferior positivo** del modo **raíz** del argumento, y es diferente de cero, se produce la excepción *UNDERFLOW*.

ARCCOS, EXP y *SQRT* que no son **constantes** causan la excepción *OVERFLOW* si el valor resultante es mayor que el **límite superior** o menor que el **límite inferior** del modo **raíz** de la clase del argumento. En el caso de un valor resultante matemático exacto que es mayor que cero y menor que el **límite inferior positivo** del modo **raíz** del argumento, se produce una excepción *UNDERFLOW*.

ejemplos:

9.12 *MIN (sieve)* *(1.7)*

11.47 *PRED (col_1)* *(1.2)*

11.47 *SUCC (col_1)* *(1.3)*

6.20.4 Rutinas incorporadas que manejan almacenamiento dinámico

sintaxis:

<llamada a rutina incorporada atribuir> ::= *(1)*
GETSTACK (*<argumento de modo>* [, *<valor>* |
([*<lista de parámetros efectivos constructor>*])]) *(1.1)*
| *ALLOCATE* (*<argumento de modo>* [, *<valor>*] |
([*<lista de parámetros efectivos constructor>*])]) *(1.2)*

<llamada a rutina incorporada terminar> ::= *(2)*
TERMINATE (*<valor primitivo de referencia>*) *(2.1)*

semántica: *GETSTACK* y *ALLOCATE* crean una localización del modo especificado y entregan un valor de referencia para la localización creada. *GETSTACK* crea esta localización en la pila (véase 10.9). Se crea una localización cuyo

modo es el del *argumento de modo*, y se entrega un valor referente al mismo. La localización creada se inicializa con el valor de *valor*, si está presente, y si no, con el valor **indefinido** (véase 4.1.2) si el argumento de modo no es un *modo moreta*.

Si el *argumento de modo* es un *modo moreta*, primeramente todas las inicializaciones en los componentes se efectúan en orden textual. Si se especifica una lista de parámetros (posiblemente vacía), el correspondiente **constructor** del *argumento de modo* se aplica a la localización últimamente creada. Si el *argumento de modo* es un *modo tarea*, se empieza la tarea perteneciente a la localización últimamente creada.

TERMINATE termina el tiempo de vida de la localización a la que se refiere el valor entregado por *valor primitivo de referencia*. En consecuencia, una implementación podría liberar el almacenamiento ocupado por esa localización y, si el *valor primitivo de referencia* es una localización que no es de **lectura solamente**, asignar el valor **indefinido** a la localización.

Si el *valor primitivo de referencia* se refiere a una localización de **región** o de **tarea L**, se efectúan secuencialmente los pasos siguientes:

a) Se cierra **L**. Si se cierra una localización, no se acepta más ninguna otra llamada externa de los procedimientos componentes **públicos** en **L**.

b) El hilo que ejecuta *TERMINATE* espera hasta que **L** esté vacía.

c) Si el modo de **L** contiene un **destructor**, ese **destructor** se aplica a L.

propiedades estáticas: La clase de una llamada a rutina incorporada *GETSTACK* o *ALLOCATE* es la clase M-referencia, donde M es el modo del *argumento de modo*. M es o bien el *nombre de modo* o un modo **parametrizado** construido como sigue:

& <nombre de modo matriz> (<expresión>) o

& <nombre de modo cadena> (<expresión entera>) o

& <nombre de modo de estructura variable> (<lista de expresiones>),

respectivamente.

Una llamada a rutina incorporada *GETSTACK* o *ALLOCATE* es **intrarregional** si está rodeada por una región; en otro caso, es **extrarregional**.

condiciones estáticas: La clase del *valor*, si existe, en la llamada a rutina incorporada *GETSTACK* y *ALLOCATE* debe ser **compatible** con el modo de *argumento de modo*; esta verificación es dinámica cuando el modo de *argumento de modo* es un modo dinámico.

Si el modo del *argumento de modo* tiene la **propiedad de lectura solamente**, debe estar presente el segundo argumento.

El *valor*, si existe, en las llamadas a rutina incorporadas *GETSTACK* y *ALLOCATE*, debe ser **regionalmente seguro** para la localización creada.

propiedades dinámicas: Un valor de referencia es un valor de referencia **atribuido** si y sólo si es retornado por una llamada a rutina incorporada *ALLOCATE*.

condiciones dinámicas: *GETSTACK* causa la excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ALLOCATE causa la excepción *ALLOCATEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

Para *GETSTACK* y *ALLOCATE* son aplicables las condiciones de asignación del valor entregado por *valor* con respecto al modo de *argumento de modo*.

TERMINATE causa la excepción *EMPTY* si el *valor primitivo de referencia* entrega el valor *NULL*.

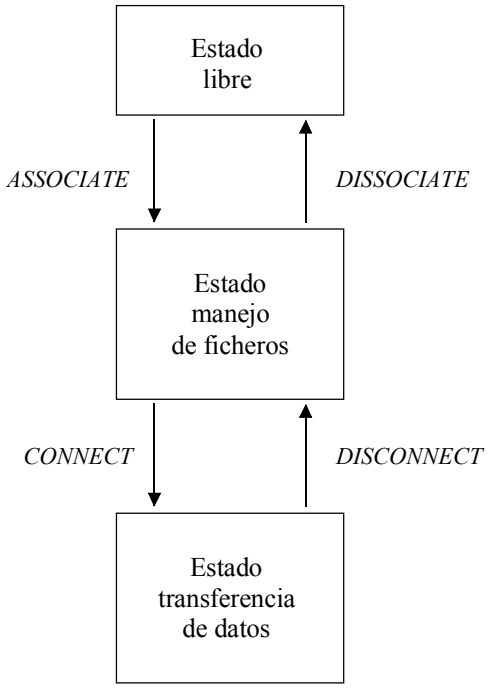
El *valor primitivo de referencia* debe entregar un valor de referencia **atribuido**. El tiempo de vida de la localización referenciada no debe haber terminado.

7 Entrada y salida

7.1 Modelo de referencia E/S

Se utiliza un modelo para la descripción de los medios de entrada/salida de una manera independiente de la implementación; para una localización de asociación dada se distinguen tres estados: un estado libre, un estado manejo de ficheros y un estado transferencia de datos.

El diagrama muestra los tres estados y las posibles transiciones entre los mismos.



La localización asociación no contiene ningún valor. No hay relación con un objeto del mundo exterior.

La localización asociación contiene una asociación. Operaciones como crear y suprimir un fichero, o cambiar sus propriedades.

Una localización exceso está conectada a la localización asociación. Transferencia de datos hacia/desde un fichero: operaciones de lectura y escritura.

Este modelo presupone que en el mundo exterior, es decir, en el entorno (externo) de un programa CHILL, existen objetos, en implementaciones que suelen conocerse por conjuntos de datos, ficheros o dispositivos. En el modelo, tal objeto del mundo exterior se denomina un fichero. Un fichero puede ser un dispositivo físico, una línea de comunicación o un fichero propiamente dicho en un sistema de gestión de ficheros; en general, un fichero es un objeto que puede producir y/o consumir datos.

Para manipular un fichero en CHILL se necesita una asociación; una asociación se crea por la operación asociar, e identifica un fichero. Una asociación tiene atributos; estos atributos describen las propiedades de un fichero que está o podría estar vinculado a la asociación.

En el estado libre, no hay interacción o relación entre el programa CHILL y objetos del mundo exterior. La operación asociar cambia el estado del modelo haciéndolo pasar del estado libre al estado manejo de ficheros. Esta operación utiliza como un argumento una localización asociación y una denotación definida por la implementación para un objeto del mundo exterior con respecto al cual hay que crear una asociación; se pueden utilizar argumentos adicionales para indicar la clase de asociación para el objeto, y los valores iniciales para los atributos de la asociación. Una determinada asociación implica también un conjunto (dependiente de la implementación) de operaciones que pueden efectuarse sobre el fichero que está vinculado a esa asociación.

En el estado manejo de ficheros es posible manipular un fichero y sus propiedades mediante una asociación, a condición que la asociación permita la operación en cuestión; para operaciones que cambian las propiedades de un fichero será necesario, en general, una asociación exclusiva para ese fichero.

El modelo supone que las asociaciones en general son exclusivas: es decir, para un objeto dado del mundo exterior existe una sola asociación en un momento cualquiera. Sin embargo, ciertas implementaciones pueden permitir la creación de más asociaciones para el mismo objeto, siempre que el objeto pueda compartirse entre diferentes usuarios (programas) y/o entre diferentes asociaciones dentro del mismo programa. Todas las operaciones en el estado manejo de ficheros utilizan una asociación como argumento.

La operación disociar se emplea para terminar una asociación respecto a un objeto del mundo exterior; esta operación produce una transición del estado manejo de ficheros al estado libre.

La transferencia de datos hacia un fichero o desde un fichero sólo es posible desde el estado transferencia de datos; para operaciones de transferencia es necesario que una localización de acceso se conecte a una asociación para ese fichero. La operación conectar conecta una localización de acceso a una asociación y hace que el modelo pase al estado transferencia de datos. Dicha operación utiliza como argumento una localización asociación y una localización acceso; la localización

asociación contiene una asociación para el fichero hacia el cual, o a partir del cual, pueden transferirse datos vía de localización acceso. Argumentos adicionales de la operación conectar denotan para qué tipo de operaciones de transferencia hay que conectar la localización acceso, y a qué registro habrá que posicionar el fichero. En un instante cualquiera, nunca podrá haber más de una localización acceso conectada a una localización asociación.

La operación desconectar utiliza como argumento una localización acceso y la desconecta de la asociación a la cual aquella está conectada; esta operación hace que el estado del modelo retorne al estado manejo de ficheros.

En el estado transferencia de datos hay que utilizar una localización acceso como argumento de una operación de transferencia; se proporcionan dos operaciones de transferencia, a saber, una operación de lectura para transferir datos de un fichero al programa y una operación de escritura para transferir datos del programa a un fichero. Las operaciones de transferencia utilizan el modo registro de la localización acceso para transformar valores CHILL en registros del fichero, y viceversa.

En el modelo, un fichero se percibe con una matriz de valores; cada elemento de esta matriz está relacionado con un registro del fichero. La operación conectar determina que el modo elemento de esta matriz sea el modo registro de la localización acceso que se está conectando. Se asigna un valor índice a cada registro del fichero. Este valor identifica unívocamente cada registro del fichero. En la descripción de las operaciones conectar y transferir se utilizarán tres valores de índices especiales, a saber, un índice de **base**, un índice **vigente** y un índice de **transferencia**. El índice de **base** se establece por la operación conectar y se mantiene sin modificación hasta la siguiente operación conectar; se utiliza para calcular el índice de **transferencia** en operaciones de transferencia y el índice **vigente** en una operación conectar. El índice de **transferencia** indica la posición del fichero en la cual se producirá una transferencia; el índice **vigente** indica el registro al cual está posicionado el fichero en un momento dado.

7.2 Valores de asociación

7.2.1 Generalidades

Un valor de asociación refleja las propiedades que un fichero podría tener o a las cuales podría estar vinculado. Un valor de asociación determinado implica también un conjunto (dependiente de la implementación) de operaciones sobre el fichero al que posiblemente está vinculado.

Los valores de asociación no tienen una denotación pero están contenidos en localizaciones de modo asociación; no existe una expresión que denote un valor de modo asociación. Los valores de asociación sólo pueden manipularse por rutinas incorporadas que utilizan como parámetro una localización de asociación.

7.2.2 Atributos de valores de asociación

Un valor de asociación tiene atributos; estos atributos describen las propiedades de la asociación y el fichero que está o puede estar vinculado a la misma.

Los siguientes atributos están definidos por el lenguaje:

- **existente**: indica que un fichero (posiblemente vacío) está vinculado a la asociación;
- **legible**: indica que son posibles operaciones de lectura en el fichero cuando éste está vinculado a la asociación;
- **escribible**: indica que son posibles operaciones de escritura en el fichero cuando éste está vinculado a la asociación;
- **indexable**: indica que el fichero, cuando está vinculado a la asociación, permite el acceso arbitrario (dícese también acceso aleatorio) a sus registros;
- **secuenciable**: indica que el fichero, cuando está vinculado a la asociación, permite el acceso secuencial a sus registros;
- **variable**: indica que el **tamaño** de los registros del fichero, cuando éste está vinculado a la asociación, puede variar dentro del fichero.

Estos atributos tienen un valor booleano; los atributos se inicializan cuando se crea la asociación y pueden actualizarse como consecuencia de operaciones particulares sobre la asociación. Esta lista comprende solamente los atributos definidos por el lenguaje; las implementaciones pueden añadir atributos según sus propias necesidades.

7.3 Valores de acceso

7.3.1 Generalidades

Los valores de acceso están contenidos en localizaciones de modo acceso. Una localización acceso es necesaria para transferir datos desde un fichero o hacia un fichero del mundo exterior.

Los valores de acceso no tienen denotación pero están contenidos en localizaciones de modo acceso; no existe una expresión que denote un valor de modo acceso. Los valores de acceso sólo pueden ser manipulados por rutinas incorporadas que utilizan una localización acceso como parámetro.

7.3.2 Atributos de valores de acceso

Los valores de acceso tienen atributos que describen sus propiedades dinámicas, la semántica de las operaciones de transferencia, y las condiciones en las que pueden producirse excepciones.

CHILL define los siguientes atributos:

- **utilización**: indica para qué operación u operaciones de transferencia la localización acceso está conectada a una asociación; el atributo se fija por la operación conectada;
- **fuera del fichero**: indica si el índice de transferencia calculado por la última operación de lectura está o no está en el fichero; el atributo *FALSE* por la operación conectar y tomar un valor por cada operación de lectura.

7.4 Rutinas incorporadas para entrada/salida

7.4.1 Generalidades

Existen rutinas incorporadas definidas por lenguaje para operaciones sobre localizaciones asociación y localizaciones acceso, y para inspeccionar y cambiar los atributos de sus valores.

Las rutinas incorporadas se describen en los puntos siguientes:

sintaxis:

<llamada a rutina incorporada de e/s que entrega un valor> ::= *(1)*
<llamada a rutina incorporada que entrega un atributo de asociación> *(1.1)*
| *<llamada a rutina e/s asociado>* *(1.2)*
| *<llamada a rutina incorporada que entrega un atributo de acceso>* *(1.3)*
| *<llamada a rutina incorporada leer registro>* *(1.4)*
| *<llamada a rutina incorporada obtener texto>* *(1.5)*

<llamada a rutina incorporada simple de e/s> ::= *(2)*
<llamada a rutina incorporada disociar> *(2.1)*
| *<llamada a rutina incorporada modificación>* *(2.2)*
| *<llamada a rutina incorporada conectar>* *(2.3)*
| *<llamada a rutina incorporada desconectar>* *(2.4)*
| *<llamada a rutina incorporada escribir registro>* *(2.5)*
| *<llamada a rutina incorporada texto>* *(2.6)*
| *<llamada a rutina incorporada establecer texto>* *(2.7)*

<llamada a rutina incorporada de e/s que entrega una localización> ::= *(3)*
<llamada a rutina incorporada asociar> *(3.1)*

condiciones estáticas: Un *parámetro de rutina incorporada* en una rutina incorporada de e/s que es una *localización asociación*, *acceso* o *texto* debe ser **referenciable**.

7.4.2 Asociar un objeto del mundo exterior

sintaxis:

<llamada a rutina incorporada asociar> ::= *(1)*
ASSOCIATE (*<localización asociación>* [, *<lista de parámetros asociar>*]) *(1.1)*

<llamada a rutina incorporada e/s asociado> ::= *(2)*
ISASSOCIATED (*<localización asociación>*) *(2.1)*

<lista de parámetros asociar> ::= *(3)*
<parámetro asociar> { , *<parámetro asociar>* }* *(3.1)*

<parámetro asociar> ::= *(4)*
<localización> *(4.1)*
| *<valor>* *(4.2)*

semántica: *ASSOCIATE* crea una asociación con un objeto del mundo exterior. Inicializa la *localización asociación* con la asociación creada. Inicializa los atributos de la asociación creada. La localización asociación se retorna también como resultado de la llamada. La asociación particular que se crea está determinada por las localizaciones y/o valores que aparecen en la *lista de parámetros asociar*; los modos (clases) y la semántica de estas localizaciones (valores) están definidos por la implementación.

ISASSOCIATED retorna *TRUE* si la *localización asociación* contiene una asociación y *FALSE* en otro caso.

propiedades estáticas: La clase de una llamada a rutina incorporada *ISASSOCIATED* es la clase *BOOL*-derivada. El modo de una llamada a rutina incorporada *ASSOCIATE* es el modo de la *localización asociación*.

La **regionalidad** de una llamada a rutina incorporada *ASSOCIATION* es la de la *localización asociación*.

condiciones estáticas: El modo y la clase de cada *parámetro asociar* está definido por la implementación.

condiciones dinámicas: *ASSOCIATE* causa la excepción *ASSOCIATEFAIL* si la *localización asociación* contiene ya una asociación o si la asociación no puede crearse por razones definidas por la implementación.

ejemplo:

20.21 ASSOCIATE (file_association,"DSK:RECORDS.DAT"); (1.1)

7.4.3 Disociar un objeto del mundo exterior

sintaxis:

<llamada a rutina incorporada disociar> ::= *(1)*
DISSOCIATE (*<localización asociación>*) *(1.1)*

semántica: *DISSOCIATE* termina una asociación con un objeto del mundo exterior. Una localización acceso que esté aún conectada a la asociación contenida en una localización asociación se desconecta antes de terminarse la asociación.

condiciones dinámicas: *DISSOCIATE* causa la excepción *NOTASSOCIATED* si la *localización asociación* no contiene una asociación.

ejemplo:

22.38 DISSOCIATE (association); (1.1)

7.4.4 Acceder a atributos de asociación

sintaxis:

<llamada a rutina incorporada que entrega un atributo de asociación> ::= *(1)*
EXISTING (*<localización asociación>*) *(1.1)*
| *READABLE* (*<localización asociación>*) *(1.2)*
| *WRITEABLE* (*<localización asociación>*) *(1.3)*
| *INDEXABLE* (*<localización asociación>*) *(1.4)*
| *SEQUENCIBLE* (*<localización asociación>*) *(1.5)*
| *VARIABLE* (*<localización asociación>*) *(1.6)*

semántica: *EXISTING*, *READABLE*, *WRITEABLE*, *INDEXABLE*, *SEQUENCIBLE* y *VARIABLE* entregan respectivamente el valor del atributo **existente**, **legible**, **escribible**, **indexable**, **secuenciable** y **variable** de la asociación contenida en la *localización asociación*.

propiedades estáticas: La clase de una *llamada a rutina incorporada que entrega un valor atributo de asociación* es la clase *BOOL*-derivada.

condiciones dinámicas: La *llamada a rutina incorporada que entrega un atributo de asociación* causa la excepción *NOTASSOCIATED* si la *localización asociación* no contiene una asociación.

7.4.5 Modificar atributos de asociación

sintaxis:

<llamada a rutina incorporada modificación> ::= *(1)*
CREATE (*<localización asociación>*) *(1.1)*
| *DELETE* (*<localización asociación>*) *(1.2)*
| *MODIFY* (*<localización asociación>* [, *<lista de parámetros modificar>*]) *(1.3)*

<lista de parámetros modificar> ::= *(2)*
<parámetro modificar> { , *<parámetro modificar>* }* *(2.1)*

<parámetro modificar> ::= *(3)*
<valor> *(3.1)*
| *<localización>* *(3.2)*

semántica: *CREATE* crea un fichero vacío y lo vincula a la asociación denotada por la *localización asociación*. El atributo **existente** de la asociación indicada se pone a *TRUE* si la operación tiene éxito.

DELETE desvincula un fichero de la asociación denotada por *localización asociación* y lo suprime. El atributo **existente** de la asociación indicada se pone a *FALSE* si la operación tiene éxito.

MODIFY proporciona el medio de cambiar propiedades de un objeto del mundo exterior con el cual existe una asociación y que está designado por *localización asociación*; las localizaciones y/o valores que se producen en la *lista de parámetros modificar* describen cómo deben modificarse las propiedades. Los modos (clases) y la semántica de estas localizaciones (valores) están definidos por la implementación.

condiciones dinámicas: *CREATE*, *DELETE* y *MODIFY* causan la excepción *NOTASSOCIATED* si la *localización asociación* no contiene una asociación.

CREATE causa la excepción *CREATEFAIL* si se cumple una de las siguientes condiciones:

- el atributo **existente** de la asociación es *TRUE*;
- fracasa la creación del fichero (definido por la implementación).

DELETE causa la excepción *DELETEFAIL* si se cumple una de las siguientes condiciones:

- el atributo **existente** de la asociación es *FALSE*;
- fracasa la supresión del fichero (definido por la implementación).

MODIFY causa la excepción *MODIFYFAIL* si no se pueden – o no se permite – modificar las propiedades, definidas por la *lista de parámetros modificar*; las condiciones en las que se produce esta excepción se definen en la implementación.

ejemplos:

21.39 CREATE (outassoc); *(1.1)*

21.69 DELETE (curassoc); *(1.2)*

7.4.6 Conectar una localización acceso

sintaxis:

<llamada a rutina incorporada conectar> ::= *(1)*
CONNECT (*<localización transferencia>*, *<localización asociación>*,
<expresión utilización> [, *<expresión dónde>* [, *<expresión índice>*]]) *(1.1)*

<localización transferencia> ::= *(2)*
<localización acesso> *(2.1)*
| *<localización texto>* *(2.2)*

<expresión utilización> ::= *(3)*
<expresión> *(3.1)*

<expresión dónde> ::= *(4)*
<expresión> *(4.1)*

<expresión índice> ::= *(5)*
<expresión> *(5.1)*

nombres predefinidos: Para controlar la operación conectar, realizada por la rutina incorporada *CONNECT*, hay dos nombres de **sínmodo** predefinidos en el lenguaje, a saber, *USAGE* y *WHERE*; sus modos **definidores** son **SET** (*READONLY*, *WRITEONLY*, *READWRITE*) y **SET** (*FIRST*, *SAME*, *LAST*), respectivamente.

Los valores del modo *USAGE* indican para qué tipo de operaciones de transferencia debe conectarse la localización de acceso a una asociación, mientras que los valores del modo *WHERE* indican cómo debe el fichero que está vinculado a una asociación ser posicionado por la operación conectar.

semántica: *CONNECT* conecta la localización acceso denotada por *localización transferencia* a la asociación que está contenida en la *localización asociación*; debe haber un fichero vinculado a la asociación designada; es decir, el atributo **existente** de la asociación tiene que ser *TRUE*.

La localización de acceso denotada por *localización transferencia* es la localización propiamente dicha si se trata de una *localización acceso*; en otro caso, será la sublocalización **acceso** de la *localización texto*.

El valor entregado por *expresión utilización* indica para qué tipo de operaciones de transferencia debe la localización acceso estar conectada al fichero. Si la expresión entrega *READONLY*, la conexión está preparada para operaciones de lectura solamente; si entrega *WRITEONLY*, la conexión está establecida para la operación de escritura solamente; si entrega *READWRITE*, la conexión está preparada para las operaciones de lectura y escritura.

El atributo **indexable** de la asociación designada tiene que ser *TRUE* si la localización acceso tiene un modo **índice**, mientras que el atributo **secuenciable** debe ser *TRUE* si la localización no tiene modo **índice**.

CONNECT (re)posiciona el fichero que está vinculado a la asociación designada; es decir, establece un índice de **base** (nuevo) y un índice **vigente** en el fichero. El índice de **base** (nuevo) depende del valor proporcionado por la *expresión dónde*:

- si la *expresión dónde* entrega *FIRST* o no está especificada, el índice de **base** se pone a 0, es decir, el fichero se posiciona antes del primer registro;
- si la *expresión dónde* entrega *SAME*, el índice de **base** se pone al índice **vigente** en el fichero; es decir, la posición del fichero no cambia;
- si la *expresión dónde* entrega *LAST*, el índice de **base** se pone a N, siendo N el número de registros en el fichero; es decir, el fichero se posiciona después del último registro.

Una vez fijado un índice de **base**, se establecerá, mediante *CONNECT*, un índice **vigente**. Este índice **vigente** depende de la especificación facultativa de una *expresión índice*:

- si no se especifica una *expresión índice*, el índice **vigente** se pone al índice de **base** (nuevo);
- si se especifica una *expresión índice*, el índice **vigente** se pone a

  índice de **base** + *NUM* (*v*) – *NUM* (*l*)

  donde *l* denota el **límite inferior** del modo **índice** de la localización acceso y *v* denota el valor proporcionado por la *expresión índice*.

Si la localización acceso se está conectando para operaciones secuenciales de escritura (es decir, la localización acceso no tiene el modo **índice** y la *expresión utilización* entrega *WRITEONLY*), los registros en el fichero que tienen un índice mayor que el índice **vigente** (nuevo) se suprimirán del fichero; es decir, el fichero puede ser truncado o vaciado por *CONNECT*.

Una localización acceso que no tiene modo índice no puede conectarse a una asociación para operaciones de lectura y escritura al mismo tiempo.

Toda localización acceso a la cual pueda conectarse la asociación designada, se desconectará implícitamente antes de que se conecte a la localización designada por *localización transferencia*.

CONNECT inicializa el atributo **fuera de fichero** de la localización acceso a *FALSE* y pone el atributo **utilización** según el valor proporcionado por *expresión utilización*.

propiedades estáticas: El modo ligado a una *localización transferencia* es el modo de la *localización acceso* o el modo **acceso** de la *localización texto*, respectivamente.

condiciones estáticas: El modo de la *localización tranferencia* debe tener un modo **índice** si se especifica una *expresión índice*; la clase del valor entregado por *expresión índice* debe ser **compatible** con ese modo **índice**. La *localización transferencia* debe tener la misma **regionalidad** que la *localización asociación*.

La clase del valor entregado por *expresión utilización* debe ser **compatible** con la clase *USAGE*-derivada.

La clase del valor proporcionado por *expresión dónde* debe ser **compatible** con la clase *WHERE*-derivada.

condiciones dinámicas: *CONNECT* causa la excepción *NOTASSOCIATED* si la *localización asociación* no contiene una asociación.

CONNECT causa la excepción *CONNECTFAIL* si se cumple una de las siguientes condiciones:

- el atributo **existente** de la asociación es *FALSE*;
- el atributo **legible** de la asociación es *FALSE* y la *expresión utilización* entrega *READONLY* o *READWRITE*;
- el atributo **escribible** de la asociación es *FALSE* y la *expresión utilización* proporciona *WRITEONLY* o *READWRITE*;
- el atributo **indexable** de la asociación es *FALSE* y la localización acceso tiene un modo **índice**;
- el atributo **secuenciable** de la asociación es *FALSE* y la localización acceso no tiene modo **índice**;
- la *expresión dónde* proporciona *SAME*, mientras que la asociación contenida en *localización asociación* no está conectada a una localización acceso;
- el atributo **variable** de la asociación es *FALSE* y la localización acceso tiene un modo **registro dinámico**, mientras que la *expresión utilización* proporciona *WRITEONLY* o *READWRITE*;
- el atributo **variable** de la asociación es *TRUE* y la localización acceso tiene un modo **registro estático**, mientras que la *expresión utilización* proporciona *READONLY* o *READWRITE*;
- la localización acceso no tiene modo **índice**, mientras que la *expresión utilización* proporciona *READWRITE*;
- la asociación contenida en la *localización asociación* no puede conectarse a la localización acceso, debido a condiciones definidas por la implementación.

CONNECT causa la excepción *RANGEFAIL* si el modo **índice** de la localización acceso es un modo intervalo y la *expresión índice* proporciona un valor que rebasa los límites de ese modo intervalo.

Se produce la excepción *EMPTY* si la **referencia de acceso** de la *localización texto* entrega el valor *NULL*.

ejemplos:

20.22 CONNECT (record_file, file_association, READWRITE); *(1.1)*

20.22 READWRITE *(3.1)*

7.4.7 Desconectar una localización acceso

sintaxis:

<llamada a rutina incorporada desconectar> ::= *(1)*
DISCONNECT (*<localización transferencia>*) *(1.1)*

semántica: *DISCONNECT* desconecta la localización acceso denotada por *localización transferencia* de la asociación a la que está conectada.

condiciones dinámicas: *DISCONNECT* causa la excepción *NOTCONNECTED* si la localización acceso denotada por *localización transferencia* no está conectada a una asociación.

7.4.8 Acceder a atributos de localizaciones acceso

sintaxis:

<llamada a rutina incorporada que entrega un atributo de acceso> ::= *(1)*
GETASSOCIATION (*<localización transferencia>*) *(1.1)*
| *GETUSAGE* (*<localización transferencia>*) *(1.2)*
| *OUTOFFILE* (*<localización transferencia>*) *(1.3)*

semántica: *GETASSOCIATION* devuelve un valor de referencia a la localización asociación a la cual está conectada la localización acceso denotada por la *localización transferencia*; retorna *NULL* si la localización acceso no está conectada a una asociación.

GETUSAGE retorna el valor del atributo **utilización**, es decir, *READONLY (WRITEONLY)* si la localización acceso está conectada solamente para operaciones de lectura (escritura) o *READWRITE* si la localización acceso está conectada para ambas operaciones, de lectura y escritura.

OUTOFFILE devuelve el valor del atributo **fuera de fichero** de la localización acceso, es decir, *TRUE* si la última operación de lectura calculó un índice de **transferencia** que no estaba en el fichero; en otro caso, entrega *FALSE*.

propiedades estáticas: La clase de una llamada a rutina incorporada *GETASSOCIATION* es la clase de referencia *ASSOCIATION*. La **regionalidad** de una llamada a rutina incorporada *GETASSOCIATION* es la de la *localización transferencia*.

La clase de una llamada a rutina incorporada *OUTOFFILE* es la clase *BOOL*-derivada.

La clase de una llamada a rutina incorporada *GETUSAGE* es la clase *USAGE*-derivada.

condiciones dinámicas: *GETUSAGE* y *OUTOFFILE* causan la excepción *NOTCONNECTED* si la localización acceso no está conectada a una asociación.

ejemplo:

21.47 OUTOFFILE (infiles (FALSE)) *(1.3)*

7.4.9 Operaciones de transferencia de datos

sintaxis:

<llamada a rutina incorporada leer registro> ::= *(1)*
READRECORD (*<localización acceso>* [, *<expresión índice>*]
[, *<localización almacenamiento>*]) *(1.1)*

<llamada a rutina incorporada escribir registro> ::= *(2)*
WRITERECORD (*<localización acceso>* [, *<expresión índice>*],
<expresión escribir>) *(2.1)*

<localización almacenamiento> ::= *(3)*
<localización modo estático> *(3.1)*

<expresión escribir> ::= *(4)*
<expresión> *(4.1)*

NOTA – Si la *localización acceso* tiene un modo **índice**, la ambigüedad sintáctica se resuelve interpretando el segundo argumento como una *expresión índice* y no como una *localización almacenamiento*.

semántica: Para la transferencia de datos hacia o desde un fichero, están definidas las rutinas incorporadas *WRITERECORD* y *READRECORD*. La *localización acceso* debe tener un modo **registro** y estar conectada a una asociación para transferencia de datos hacia o desde el fichero que está conectado a esa asociación. El sentido de transferencia no debe estar en contradicción con el valor efectivo del atributo **utilización** de la *localización acceso*.

Antes de que se produzca una transferencia, se calcula el índice de **transferencia**, es decir, la posición en el fichero del registro que ha de transferirse. Si la *localización acceso* no tiene modo **índice**, el índice de **transferencia** es el índice **vigente** incrementado en 1; si la *localización acceso* tiene un modo **índice**, el índice de **transferencia** se calcula como sigue:

índice de **transferencia** := índice de **base** + *NUM* (*v*) – *NUM* (*l*) + 1

siendo *l* el **límite inferior** del modo **índice** de la *localización acceso* y *v* el valor entregado por la *expresión índice*. Si la transferencia del registro o el índice de **transferencia** calculado se ha realizado con éxito, el índice **vigente** pasa a ser el índice de **transferencia**.

La operación lectura:

READRECORD transfiere datos de un fichero del mundo exterior al programa CHILL.

Si el índice de **transferencia** calculado no está en el fichero, el atributo **fuera del fichero** se pone a *TRUE*; en otro caso, se posiciona el fichero y se lee el registro, y el atributo **fuera del fichero** se pone a *FALSE*.

El registro que se lee no debe entregar un valor **indefinido**; el efecto de la operación lectura viene definido por la implementación si el registro que se lee desde el fichero no es un valor legal de acuerdo con el modo **registro** de la *localización acceso*.

Si se especifica una *localización almacenamiento*, se asigna a esta localización el valor del registro que se leyó. Si no se especifica *localización almacenamiento*, el valor se asignará a una localización creada implícitamente; el tiempo de vida de esta localización termina cuando la *localización acceso* se desconecta o reconecta. No está definido si la localización referenciada es creada una sola vez por la operación conectar o cada vez que se realiza una operación lectura.

READRECORD retorna en ambos casos un valor de referencia relativo a la localización (posiblemente de modo dinámico) a la que estaba asignado el valor.

Si el atributo **fuera del fichero** se pone a *TRUE* como resultado de la llamada a rutina incorporada, se devuelve el valor *NULL* como resultado de la llamada.

La operación escritura:

WRITERECORD transfiere datos del programa CHILL a un fichero del mundo exterior. El fichero se posiciona al registro que tiene el índice calculado y se escribe el registro.

Una vez escrito con éxito el registro, el número de registros se pone al índice de **transferencia**, si éste es mayor que el número efectivo de registros.

El registro escrito por *WRITERECORD* es el valor entregado por la *expresión escribir*.

propiedades estáticas: La clase del valor leído por *READRECORD* es la clase M-valuada, siendo M el modo **registro** de la *localización acceso*, si está tiene un modo **registro estático**, o una versión parametrizada dinámicamente de ella, si la localización tiene un modo **registro dinámico**; los parámetros de tal modo registro parametrizado dinámicamente son:

- la **longitud de cadena** dinámica del valor cadena que se leyó en el caso de un modo cadena;
- el **límite superior** dinámico del valor matriz que se leyó en el caso de un modo matriz;
- la lista de valores (marcadores) asociados con el modo del valor estructura que se leyó en el caso de una estructura **variable**.

La clase de la llamada a rutina incorporada *READRECORD* es la clase M-referencia si *localización almacenamiento* no está especificada; en otro caso, es la clase S-referencia, donde S es modo de la *localización almacenamiento*.

La **regionalidad** de una llamada a rutina incorporada *READRECORD* es la de la *localización almacenamiento* si está especificada, en otro caso, es la de la *localización acceso*.

condiciones estáticas: La *localización acceso* debe tener un modo **registro**.

Una *expresión índice* puede no estar especificada si la *localización acceso* no tiene un modo **índice**, y debe especificarse si la *localización acceso* tiene un modo **índice**; la clase del valor proporcionado por la *expresión índice* debe ser **compatible** con el modo **índice**.

La *localización almacenamiento* debe ser **referenciable**.

El modo de *localización almacenamiento* no debe tener la **propiedad de lectura solamente**.

Si se especifica *localización almacenamiento*, el modo de *localización almacenamiento* debe ser **equivalente** al modo **registro** de la *localización acceso*, si tiene un modo **registro estático** o un modo **registro** cadena **variable**, y en otro caso a una versión parametrizada dinámicamente de la misma; los parámetros de ese modo parametrizado dinámicamente son los del valor que ha sido leído.

La clase del valor entregado por *expresión escribir* debe ser **compatible** con el modo **registro** de la *localización acceso* si tiene un modo **registro estático** un modo **registro** cadena **variable**; en otro caso deberá existir una versión parametrizada dinámicamente del modo **registro** que sea **compatible** con la clase de *expresión escribir*. Se aplican las condiciones de asignación del valor de *expresión escribir* con respecto al modo citado anteriormente.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* o *TAGFAIL* si falla la parte dinámica de la comprobación de compatibilidad antes mencionada.

La llamada a rutina incorporada *READRECORD* y *WRITERECORD* causa la excepción *NOTCONNECTED* si la *localización acceso* no está conectada a una asociación.

La llamada a rutina incorporada *READRECORD* o *WRITERECORD* causa la excepción *RANGEFAIL* si el modo **índice** de *localización acceso* es un modo intervalo y la *expresión índice* entrega un valor que rebasa los límites de ese modo intervalo.

La llamada a rutina incorporada *READRECORD* causa la excepción *READFAIL* si se cumple una de las siguientes condiciones:

- el valor del atributo **utilización** es *WRITEONLY*;
- el valor del atributo **fuera de fichero** es *TRUE* y la *localización acceso* está conectada para operaciones de lectura secuencial;
- fracasa la lectura del registro con el índice calculado, debido a condiciones del mundo exterior.

La llamada a rutina incorporada *WRITERECORD* causa la excepción *WRITEFAIL* si se cumple una de las siguientes condiciones:

- el valor del atributo **utilización** es *READONLY*;
- fracasa la escritura del registro con el índice calculado, debido a condiciones del mundo exterior.

Si se produce la excepción *RANGEFAIL* o la excepción *NOTCONNECTED*, ello ocurre antes de que se cambie el valor de cualquier atributo y antes de que se posicione el fichero.

ejemplos:

20.24 *READRECORD (record_file, curindex, record_buffer);* *(1.1)*

22.25 *READRECORD (fileaccess);* *(1.1)*

20.32 *WRITERECORD (record_file, curindex, record_buffer);* *(2.1)*

21.61 *WRITERECORD (outfile, buffers(flag));* *(2.1)*

20.24 *record_buffer* *(3.1)*

21.61 *buffers(flag)* *(4.1)*

7.5 Entrada/salida de texto

7.5.1 Generalidades

Las operaciones de salida de texto permiten representar valores CHILL en forma legible por el ser humano: las operaciones de entrada de texto realizan la transformación inversa.

Las operaciones de transferencia de texto se definen en la parte superior del modelo básico CHILL de entrada/salida y actúan sobre ficheros a los que se puede acceder ya sea secuencial o aleatoriamente, y cuyos registros pueden ser de longitud fija o variable.

El modelo supone que cada registro trabaja con una información de posicionamiento (posiblemente vacía) que en las implementaciones a menudo se denomina control del carro o caracteres de control.

La manipulación de un fichero de texto en CHILL requiere una asociación: la transferencia de datos a partir de un fichero de texto o hacia el mismo requiere la conexión de una localización de **texto** a una asociación para ese fichero.

Las operaciones de transferencia de texto pueden aplicarse a valores CHILL que pueden convertirse en registros de algún fichero de texto, así como a las localizaciones CHILL que no están necesariamente relacionadas con una actividad de entrada/salida del programa

En general, no puede garantizarse la posibilidad de recuperar, a partir de una pieza de texto, los mismos valores CHILL que la originaron, sino que depende más bien de la representación específica que se haya utilizado.

Los valores de texto están contenidos en localizaciones de modo texto. Es necesaria una localización texto para transferir datos en forma legible por el ser humano.

Los valores de texto no tienen una denotación, pero están contenidos en localizaciones modo texto; no existe una expresión que denote un valor de modo texto. Los valores de texto sólo pueden ser manipulados por rutinas incorporadas que toman como parámetro una localización texto.

7.5.2 Atributos de valores de texto

Los valores de texto tienen atributos que describen sus propiedades dinámicas. Se definen los siguientes atributos:

- **índice efectivo**: indica la posición de carácter siguiente del **registro de texto** que va a leerse o escribirse. Tiene un modo ***RANGE*** *(0:L–1)*, donde *L* es la **longitud de texto** del modo del valor. Se inicializa a 0 cuando se crea una localización de texto.
- **referencia de registro de texto**: indica un valor de referencia a la sublocalización **registro de texto** de la localización texto. Tiene un modo que es **REF** *M*, donde *M* es el modo **registro de texto** del modo del valor.
- **referencia de acceso**: indica un valor de referencia a la sublocalización **acceso** de la localización texto. Tiene un modo que es **REF** *M*, donde *M* es el modo **acceso** del modo del valor.

7.5.3 Operaciones de transferencia de texto

sintaxis:

<llamada a rutina incorporada texto> ::= *(1)*
READTEXT (*<lista de argumentos de e/s texto>*) *(1.1)*
| *WRITETEXT* (*<lista de argumentos de e/s texto>*) *(1.2)*

<lista de argumentos de e/s texto> ::= *(2)*
<argumento de texto> [, *<expresión índice>*],
<argumento de formato> [, *<lista e/s>*] *(2.1)*

<argumento de texto> ::= *(3)*
<localización texto> *(3.1)*
| *<localización cadena de caracteres>* *(3.2)*
| *<expresión cadena de caracteres>* *(3.3)*

<argumento de formato> ::= *(4)*
<expresión cadena de caracteres> *(4.1)*

<lista de e/s> ::= *(5)*
<elemento de lista e/s> {, *<elemento lista e/s>* }* *(5.1)*

<elemento de lista e/s> ::= *(6)*
<argumento de valor> *(6.1)*
| *<argumento de localización>* *(6.2)*

<argumento de localización> ::= *(7)*
<localización discreta> *(7.1)*
<localización coma flotante> *(7.2)*
| *<localización cadena>* *(7.3)*

<argumento de valor> ::= *(8)*
<expresión discreta> *(8.1)*
<expresión coma flotante> *(8.2)*
| *<expresión cadena>* *(8.3)*

NOTA – Si el *elemento de lista de e/s* es una localización, la ambigüedad sintáctica se resuelve interpretando el *elemento de lista de e/s* como un *argumento de localización*, y no como un *argumento de valor*.

semántica: *READTEXT* aplica las funciones de conversión, edición y control de e/s, contenidas en el *argumento de formato*, al **registro de texto** denotado por el *argumento de texto*; esto produce (posiblemente) una lista de valores que se asignan a los elementos de la *lista de e/s* en la secuencia en que fueron especificados. *WRITETEXT* realiza la operación inversa. No se realizan operaciones implícitas de e/s.

Si el *argumento de texto* es una *localización cadena de caracteres* o una *expresión cadena de caracteres*, las funciones de conversión y edición se aplican sin relación alguna con el mundo exterior. En este caso, el **índice efectivo** denota una localización creada implícitamente al principio de la llamada a rutina incorporada, e inicializada a 0. El **registro de texto** es la cadena de caracteres denotada por la *localización cadena de caracteres* o la *expresión cadena de caracteres* y la **longitud de texto** es la **longitud de cadena**.

Los elementos de la *lista de e/s* pueden ser:

- *argumentos de valor* y *argumentos de localización*, o
- *anchuras de cláusula* **variables**, que se describen a continuación.

Relaciones entre un argumento formato y una lista de e/s

El valor entregado por un *argumento de formato* debe tener la forma de una *cadena de control de formato* (véase 7.5.4).

Durante la ejecución de una llamada a rutina incorporada de e/s texto, la *cadena de control de formato* (véase 7.5.4) denotada por el *argumento de formato* y la *lista de e/s* son exploradas de izquierda a derecha. Se interpreta cada ocurrencia de *texto de formato* y de *especificación de formato*, y se procede como sigue:

a) *texto de formato*

En *READTEXT*, el **registro de texto** debe contener en la posición de **índice efectivo** un segmento de cadena igual a la cadena entregada por *texto de formato*. En *WRITETEXT*, la cadena entregada por *texto de formato* se transfiere al **registro de texto**. La semántica es la misma que si se encontrara una *especificación de formato* que sea %*C* y un *elemento de lista de e/s* que entrega el mismo valor de cadena que el entregado por *texto de formato*.

b) *especificación de formato*

Si la *especificación de formato* contiene un *factor de repetición*, es equivalente a una secuencia formada por tantas ocurrencias de *elemento de formato* como indique el número denotado por *factor de repetición*.

Si la *especificación de formato* es una *cláusula de formato*, contiene un *código de control*. Si el *código de control* es un *cláusula de conversión*, entonces se toma un *elemento de lista de e/s*, de la *lista de e/s*, y se le aplica la función de conversión seleccionada por *código de conversión*, *calificadores de conversión* y *anchura de cláusula* (véase 7.5.5). Si el *código de control* es una *cláusula de edición* o una *cláusula de e/s*, se aplica entonces la función edición o la función de e/s seleccionada por el *código de edición* o el *código e/s* y *anchura de cláusula* se aplica al *argumento de texto* sin referencia a la *lista de e/s* (véanse 7.5.6 y 7.5.7).

Si la *anchura de cláusula* es **variable**, se toma de la lista un valor que denota el parámetro **anchura** de la función de conversión o de control de edición.

Si la *especificación de formato* es una *cláusula entre paréntesis*, se explora la *cadena de control de formato* en ella contenida.

La interpretación de la *cadena de control de formato* termina cuando se alcanza el final de la cadena entregada por *cadena de control de formato*.

Los *elementos de lista de e/s* de la *lista de e/s* se exploran en el orden en que están especificados.

condiciones estáticas: Si el *argumento de texto* es una *localización cadena*, su modo tiene que ser un modo cadena **variable**.

Una *expresión índice* no puede especificarse si el *argumento de texto*, no es una *localización texto* o si lo es y su modo **acceso** no tiene modo **índice**, y debe especificarse si el modo **acceso** tiene un modo **índice**; la clase del valor entregado por *expresión índice* debe ser **compatible** con ese modo **índice**.

Un *argumento de texto* en una llamada a rutina incorporada *WRITETEXT* debe ser una localización.

Una *localización cadena* en un *argumento de texto* debe ser **referenciable**.

condiciones dinámicas: Se produce la excepción *TEXTFAIL* si:

- el valor de cadena entregado por el *argumento de formato* no puede derivarse como una producción terminal de la *cadena de control de formato*, o
- se intenta asignar al **índice efectivo** un valor inferior a 0 o superior a la **longitud de texto**, o
- durante la interpretación, se alcanza el final de la *cadena de control de formato* y la *lista de e/s* no está completamente explorada, o no pueden tomarse más elementos de la *lista de e/s* y la *cadena de control de formato* contiene más *códigos de conversión* o *anchuras de cláusula* **variables**, o
- se encuentra una *cláusula de e/s* y el *argumento de texto* no es una *localización texto*, o
- se encuentra un *texto formato* en *READTEXT* y el **registro de texto** no contiene en la posición de **índice efectivo** una cadena igual a la cadena entregada por *texto de formato*.

Cualquier excepción definida para las llamadas a rutina incorporada *READRECORD* y *WRITERECORD* puede ocurrir si se ejecuta una función de control de e/s y se viola cualquiera de las condiciones dinámicas definidas.

ejemplo:

26.18 WRITETEXT (output,"%B%/",10) *(1.2)*

7.5.4 Cadena de control de formato

sintaxis:

<cadena de control de formato> ::= *(1)*
[*<texto de formato>*] { *<especificación de formato>* [*<texto de formato>*] }* *(1.1)*

<texto de formato> ::= *(2)*
{ *<carácter no-por ciento>* | *<por ciento>* }* *(2.1)*

<por ciento> ::= *(3)*
%% *(3.1)*

<especificación de formato> ::= *(4)*
% [*<factor de repetición>*] *<elemento de formato>* *(4.1)*

<factor de repetición> ::= *(5)*
{ *<dígito>* }$^{+}$ *(5.1)*

<elemento de formato> ::= *(6)*
<cláusula de formato> *(6.1)*
| *<cláusula parentizada>* *(6.2)*

<cláusula de formato> ::= *(7)*
<código de control> [*%*] *(7.1)*

<código de control> ::= *(8)*
<cláusula de conversión> *(8.1)*
| *<cláusula de edición>* *(8.2)*
| *<cláusula de e/s>* *(8.3)*

<cláusula parentizada> ::= *(9)*
(*<cadena de control de formato>* *%*) *(9.1)*

NOTA – Una *especificación de formato* es terminada por el primer carácter que no puede formar parte del *elemento de formato*. Los espacios y los caracteres de formato (o determinantes de formato) no pueden utilizarse dentro de *elementos de formato*. Para terminar una *cláusula de formato* puede usarse un punto (.). Éste pertenece a la *cláusula de formato* y sólo tiene un efecto delimitador. Para representar el carácter por ciento (*%*) dentro de un *texto de formato*, tiene que escribirse dos veces (*%%*).

semántica: Una *cadena de control de formato* especifica la forma externa de los valores que se transfieren y la organización de los datos en los registros. Una *cadena de control de formato* está compuesta por ocurrencias de *texto de formato*, que denotan partes fijas de los registros, y de las ocurrencias de *especificación de formato*, que denotan las representaciones externas de valores CHILL, lo que permite la edición del **registro de texto** o el control de las operaciones de e/s efectivas.

Una *especificación de formato* que contiene un *factor de repetición* y una *cláusula de formato* es equivalente a tantas ocurrencias de *especificación de formato* idénticas, de la *cláusula de formato*, como indica el *factor de repetición*. Por ejemplo, *"%3C4"* es equivalente a *"%C4%C4%C4"*.

Se supone la notación decimal para los *dígitos* de un *factor de repetición*.

Una *cadena de control de formato* en una *cláusula parentizada* se explora repetidamente según sea el *factor de repetición*. Si no se especifica ninguno, se toma *1* por defecto.

ejemplo:

26.20 size = %C%/ *(1.1)*

7.5.5 Conversión

sintaxis:

<cláusula de conversión> ::= *(1)*
<código de conversión> { *<calificador de conversión>* }*
[*<anchura de cláusula>*] *(1.1)*

<código de conversión> ::= *(2)*
B / *O* / *H* / *C* / *F* *(2.1)*

<calificador de conversión> ::= *(3)*
L / *E* / *P <carácter>* *(3.1)*

<anchura de cláusula> ::= *(4)*
{{ *<dígito>* }* | *V* } [*<anchura fraccional>*] [*<anchura de exponente>*] *(4.1)*

<anchura fraccional> ::= *(5)*
. { *<dígito>*}$^{+}$ *(5.1)*

<anchura de exponente> ::= *(6)*
: { *<dígito>* }$^{+}$ *(6.1)*

sintaxis derivada: Una *cláusula de conversión* en la que no esté presente una *anchura de cláusula* es una sintaxis derivada para una *cláusula de conversión* en la que se especifica una *anchura de cláusula* que es 0.

semántica: Una conversión en una llamada a rutina incorporada *READTEXT* transforma una cadena que es una representación externa en un valor CHILL. Una conversión en una llamada a rutina incorporada *WRITETEXT* realiza la transformación opuesta. El *código de conversión* junto con el *calificador de conversión* especifica el tipo de la conversión y los detalles de la operación solicitada, como la justificación, el tratamiento de desbordamiento y el relleno.

La representación externa es una cadena cuya longitud suele depender del valor que se convierte. Esa cadena puede contener el número mínimo de caracteres necesarios para representar el valor CHILL (formato libre) o puede tener una longitud determinada (formato fijo).

En el formato fijo, un segmento de la dimensión **anchura** que empieza en la posición **índice efectivo** es leído en el **registro de texto**, o escrito en el mismo, según la justificación y el relleno seleccionado por los *calificadores de conversión*, de la manera siguiente:

- en *READTEXT*: todos los caracteres de relleno (a la izquierda o a la derecha, según la justificación), si existen, se suprimen. Si embargo, cuando se leen caracteres o cadenas de caracteres **fijas**, el máximo número *N* de caracteres de relleno que son suprimidos es **anchura** – *L*, donde *L* es 1 o **longitud de cadena**, respectivamente. No se suprime ningún carácter si $N < 0$. Los caracteres restantes se toman como la representación externa;

- en *WRITETEXT*: si la longitud de una representación externa es inferior o igual a **anchura**, los caracteres se justifican a la izquierda o a la derecha en el segmento (según la justificación). Los elementos de cadena no utilizados, si existen, se rellenan con el carácter de relleno. En otro caso, la cadena es truncada (a la izquierda si se seleccionó la justificación a la derecha, y en otro caso a la derecha), o se transfieren caracteres indicadores de "desbordamiento" de **anchura** (*), si está presente el calificador *E*. La truncación se aplica a la representación externa, incluido el signo menos, el punto (.) y *E* (representación científica), si existe.

En el formato libre se cumple lo siguiente:

- en *READTEXT*: los caracteres de relleno, si los hay, se saltan, salvo cuando se está leyendo un carácter o una cadena de caracteres y no se especifica el *calificador de conversión P*. Se toma entonces la representación externa como el más largo segmento de caracteres que empieza en el **índice efectivo** y está formado por todos los caracteres ulteriores que pueden pertenecerle léxicamente, como se define a continuación;

- en *WRITETEXT*: la cadena entregada por la conversión se inserta a partir de la posición del **índice efectivo**.

En *WRITETEXT*, la cadena que es la representación externa se transfiere al **registro de texto** sin tener en cuenta su **longitud efectiva**. Después de la transferencia, el **índice efectivo** es avanzado automáticamente hasta la siguiente posición de carácter disponible y la **longitud efectiva** se fija al valor máximo entre el **índice efectivo** y la **longitud efectiva** (anterior).

Una *anchura de cláusula* es **constante** si está compuesta de *dígitos*. Se supone la notación decimal. En otro caso es **variable**.

Si la **anchura** es cero, se elige el formato libre; en otro caso la **anchura** es la longitud del formato fijo.

Si la **anchura** es demasiado pequeña para contener la cadena, se toman las disposiciones apropiadas según el *calificador de conversión*.

En un *READTEXT*, la representación externa que se aplica es la que se define a continuación para el modo del *argumento de localización*.

En un *WRITETEXT*, la representación externa que se aplica es la que se define a continuación para el modo M de la clase M-valuada o M-derivada del valor entregado por el *argumento de valor*.

Códigos de conversión

Los *códigos de conversión* se representan por letras simples. Se definen los siguientes *códigos de conversión*:

B: representación binaria;

O: representación octal;

H: representación hexadecimal;

C: conversión: indica la representación externa por defecto de valores CHILL, que depende del modo del valor que se convierte (véase más adelante).

F: representación científica, es decir, la representación de valores coma flotante con mantisa y exponente.

La representación externa depende del *código de conversión* y del modo del valor que se convierte.

Calificadores de conversión

Los *calificadores de conversión* se representan por letras simples. Se definen los siguientes *calificadores de conversión*:

L: justificación izquierda. Si no está presente, se supone la justificación derecha. En el formato libre, este calificador no tiene efecto.

E: evidencia de desbordamiento. En *WRITETEXT*, se selecciona la indicación de desbordamiento; si no está presente el calificador, se realiza una truncación. En *READTEXT* o en el formato libre, este calificador no tiene efecto.

P: relleno. El carácter que sigue al calificador especifica el carácter de relleno. Si *P* no está presente, se toma el espacio como carácter de relleno, por defecto. En *READTEXT*, si se selecciona el formato libre los espacios y HT (tabulación horizontal) se consideran como el mismo carácter a efectos de salto, ya sea cuando se especifican después del calificador o cuando se aplican por defecto.

Representación externa

A continuación se define la representación externa de valores CHILL:

a) *enteros*

Los valores enteros se representan léxicamente por uno o más dígitos en base decimal por defecto sin ceros iniciales y con signo inicial si son negativos. Los caracteres subrayados, en *READTEXT* se descartan un signo más inicial y los ceros iniciales. Existen los siguientes *códigos de conversión*: *B*, *O*, *C* y *H*. El *código de conversión C* selecciona la representación decimal. Los dígitos que pueden pertenecer a la representación son sólo los seleccionados por el código de conversión.

b) *coma flotante*

Los valores coma flotante pueden representarse de dos maneras:

- representación de coma fija (seleccionada por *código de conversión C*);
- representación científica (seleccionada por *código de conversión F*).

En la representación de coma fija, el valor coma flotante se representa léxicamente por una secuencia de uno o más dígitos (parte entera) seguidos por una secuencia facultativa de uno o más dígitos (parte fraccional) separados de la primera parte por un punto (.). Un signo menos inicial está presente si el valor es negativo.

En la representación científica, el valor coma flotante se representa por mantisa y exponente. La mantisa se representa léxicamente como un valor coma fija con una parte entera constituida por un solo dígito mayor que cero. El exponente se representa léxicamente por una *E* seguida de un posible signo y una secuencia de uno o más dígitos. En ambas representaciones, un signo más y ceros iniciales se descartan en *READTEXT*.

Si está presente *anchura fraccional*, el valor entregado por *dígitos* contenidos en la misma indica la longitud de la parte fraccional extendida con ceros finales si es necesario; en otro caso, la parte fraccional contiene el número mínimo de dígitos que son necesarios para representarla.

Si está presente *anchura de exponente*, el valor entregado por *dígitos* contenidos en la misma indica el número mínimo de dígitos que habrán de utilizarse para representar el exponente, incluidos ceros iniciales si son necesarios; en otro caso se supone un valor de 3, por defecto.

Se dispone de los siguientes *códigos de conversión*: *C, F.*

c) *booleanos*

Los valores booleanos se representan léxicamente por una de dos *cadenas de nombre simple*, *TRUE* y *FALSE* [en mayúsculas (por ejemplo, *TRUE*) o minúsculas (por ejemplo, *true*), según la representación elegida en la implementación para las cadenas de nombre simple **especiales**]. Se dispone del siguiente *código de conversión*: *C.*

d) *caracteres*

Los valores de caracteres se representan léxicamente por cadenas de longitud *1*. Se dispone del siguiente *código de conversión: C.*

e) *conjuntos*

Los valores de modo conjunto se representan léxicamente por cadenas de nombre simple, que son los literales de conjunto. Se dispone del siguiente *código de conversión: C*

f) *intervalos*

Los valores de intervalo tienen la misma representación que los valores de su modo **raíz**. Si embargo, sólo las representaciones de los valores definidos por el modo intervalo pertenecen al conjunto de representaciones externas asociadas con el modo intervalo.

g) *cadenas de caracteres*

Los valores de cadena de caracteres se representan léxicamente como cadenas de caracteres de longitud *L*. En *WRITETEXT*, *L* es la **longitud efectiva**. En *READTEXT*, *L* es la **longitud de cadena** si la cadena tiene una longitud **fija**; en otro caso es una cadena **variable** y *L* es la **longitud de cadena**, a menos que haya menos caracteres disponibles en el (segmento de) **registro de texto** en la posición de **índice efectivo**, en cuyo caso *L* es el número de caracteres disponibles. Se dispone del siguiente *código de conversión: C*.

h) *cadenas de bits*

Los valores de cadena de bits se representan léxicamente como cadenas de dígitos binarios. Se aplican las mismas reglas que en las cadenas de caracteres para determinar el número de dígitos. Se dispone del siguiente *código de conversión: C*

propiedades dinámicas: Una *anchura de cláusula* tiene una **anchura**, que es el valor entregado por *dígitos* o por un valor de la *lista de e/s* si la *anchura de cláusula* es **variable**.

condiciones dinámicas: Se produce la excepción *TEXTFAIL* si:

- en *READTEXT*, el **registro de texto** no contiene un segmento de cadena que empieza en el **índice efectivo** y que (después de suprimir o saltar los caracteres de relleno) puede ser interpretado como representación externa de uno de los valores del modo del *argumento de localización* vigente (incluido un intento de leer una representación externa no vacía a partir de un **registro de texto** cuando **índice efectivo** = **longitud efectiva**); o
- en *WRITETEXT*, un segmento de cadena que es la representación externa del *argumento de valor* vigente no puede transferirse al **registro de texto** que empieza en el **índice efectivo**; o
- en *READTEXT*, se encuentra un *código de conversión* y el elemento vigente de la *lista de e/s* no es una localización, o el modo de la localización tiene la propiedad **de lectura solamente**; o
- se especifica más de una vez el mismo *calificador de conversión*; o
- se encuentra una *anchura de cláusula* **variable** y el elemento de *lista de e/s* correspondiente de la *lista de e/s* no tiene una clase entero o es inferior a 0; o
- una *anchura de cláusula* tiene una *anchura fraccional* o una *anchura de exponente* y el correspondiente *elemento de lista de e/s* en la *lista de e/s* no tiene una clase coma flotante, o tiene una *anchura de exponente* y el *código de conversión* no es *F*.

ejemplo:

26.21 CL6 *(1.1)*

7.5.6 Edición

sintaxis:

<cláusula de edición> ::= *(1)*
<código de edición> [*<anchura de cláusula>*] *(1.1)*

<código de edición> ::= *(2)*
X | < | > | *T* *(2.1)*

sintaxis derivada: Una *cláusula de edición* en la que no esté presente una *anchura de cláusula* es la sintaxis derivada para una *cláusula de edición* en la que se especifica una *anchura de cláusula* que es igual a *1* si el *código de edición* no es *T*, y en otro caso 0, respectivamente.

semántica: Se definen las siguientes funciones de edición:

X: espacio: se insertan o saltan caracteres de espacio **anchura**.

> : salto a la derecha: el **índice efectivo** se desplaza hacia la derecha para posiciones **anchura**.

<: salto a la izquierda: el **índice efectivo** se desplaza hacia la izquierda para posiciones **anchura**.

T: tabulación: el **índice efectivo** se desplaza a la posición **anchura**.

En *WRITETEXT*, si el **índice efectivo** se desplaza a una posición que es mayor que la **longitud efectiva**, se añade al **registro de texto** una cadena de *N* caracteres de espacio, siendo *N* la diferencia entre el **índice efectivo** y la **longitud efectiva** (anterior). La **longitud efectiva** se pone al valor máximo entre el **índice efectivo** y la **longitud efectiva** (anterior).

condiciones dinámicas: Se produce la excepción *TEXTFAIL* si:

- el **índice efectivo** se desplaza a una posición menor que 0 o mayor que la **longitud de texto**, o
- en *READTEXT*, el **índice efectivo** se desplaza a una posición mayor que la **longitud efectiva**; o
- en *READTEXT*, se especifica el *código de edición X*, y una cadena de caracteres de espacio **anchura** o HT (tabulación horizontal) no está presente en el **registro de texto** en la posición de **índice efectivo**.

ejemplo:

26.22 X *(1.1)*

7.5.7 Control de E/S

sintaxis:

<cláusula e/s> ::= *(1)*
<código e/s> *(1.1)*

<código e/s> ::= *(2)*
/ | – | + | ? | ! | = *(2.1)*

semántica: Las funciones de control de e/s (salvo *%=*) realizan una operación de e/s. Permiten controlar con precisión la transferencia del **registro de texto**. En *READTEXT*, todas las funciones tienen el mismo efecto, leer el registro siguiente a partir del fichero. En *WRITETEXT*, se transfiere el **registro de texto** y la representación apropiada con la información de control del carro. La posición inicial del carro en el momento en que se conecta la *localización texto* es tal que el primer carácter del primer **registro de texto** se imprima al principio de la primera línea no ocupada (independientemente de la información de posicionamiento asociada al **registro de texto**).

El emplazamiento del carro se describe por medio de las siguientes operaciones abstractas sobre la columna, línea y página (*x, y, z*) vigentes, considerándose las columnas numeradas a partir de 0 desde el margen izquierdo y las líneas a partir de 0 desde el margen superior.

nl(*w*): el carro se desplaza *w* líneas hacia abajo, posicionándose al principio de la línea (nueva posición: (0, (*y* + *w*) mod *p*, *z* + (*y* + *w*)/*p*, donde *p* es el número de líneas por página));

np(*w*): el carro se desplaza *w* páginas hacia abajo posicionándose al principio de la línea (nueva posición: (0, 0, *z* + *w*)).

Se proporcionan las siguientes funciones de control:

/: registro siguiente: el registro se imprime en la línea siguiente (nl(1), imprimir registro, nl(0));

+: página siguiente: el registro se imprime en la parte superior de la página siguiente (np(1), imprimir registro, nl(0));

–: línea vigente: el registro se imprime en la línea vigente (imprimir registro, nl(0));

?: invitación: el registro se imprime en la línea siguiente. El carro se deja al final de la línea (nl(1), imprimir registro);

!: emitir: no se realiza ningún control del carro (imprimir registro);

=: fin de página: define el posicionamiento del registro siguiente, si existe, en la parte superior de la página siguiente (esta función prevalece sobre el posicionamiento realizado antes de la impresión del registro). No causa ninguna operación de e/s.

La transferencia de E/S se realiza como sigue:

- en *READTEXT*, la semántica es como si se ejecutara un *READRECORD (A, I, R)*, donde *A* es la sublocalización **acceso** de la *localización texto*, *I* es la *expresión índice* (si la hay) y *R* denota el **registro de texto**. Después de la transferencia de E/S, el **índice efectivo** se fija a 0 y la **longitud efectiva** a la **longitud de cadena** del valor de cadena leído.
- en *WRITETEXT*, la semántica es como si se ejecutara un *WRITERECORD (A, I, R)*, donde *A* es la sublocalización **acceso** de la *localización texto*, *I* es la *expresión índice* (si la hay) y *R* denota el **registro de texto**. La información de posicionamiento asociada también se transfiere. Si el modo **registro** del acceso no es **dinámico**, el **registro de texto** es rellenado al final con caracteres de espacio, y su **longitud efectiva** se fija a **longitud de texto** antes de que se efectúe la transferencia. Después de la transferencia de E/S, el **índice efectivo** y la **longitud efectiva** se fijan a 0.

ejemplo:

26.21 / *(1.1)*

7.5.8 Acceso a los atributos de una localización texto

sintaxis:

<llamada a rutina incorporada obtener texto> ::= *(1)*
GETTEXTRECORD (*<localización texto>*) *(1.1)*
| *GETTEXTINDEX* (*<localización texto>*) *(1.2)*
| *GETTEXTACCESS* (*<localización texto>*) *(1.3)*
| *EOLN* (*<localización texto>*) *(1.4)*

<llamada a rutina incorporada establecer texto> ::= *(2)*
SETTEXTRECORD (*<localización texto>* ,
<localización cadena de caracteres>) *(2.1)*
| *SETTEXTINDEX* (*<localización texto>* , *<expresión entera>*) *(2.2)*
| *SETTEXTACCESS* (*<localización texto>* , *<localización acceso>*) *(2.3)*

semántica: *GETTEXTRECORD* devuelve la **referencia de registro de texto** de la *localización texto*.

GETTEXTINDEX devuelve el **índice efectivo** de la *localización texto*.

GETTEXTACCESS devuelve la **referencia de acceso** de la *localización texto*.

EOLN entrega *TRUE* si no hay más caracteres disponibles en el **registro de texto** (es decir, si el **índice efectivo** es igual a la **longitud efectiva**).

SETTEXTRECORD almacena una referencia a la localización entregada por la *localización cadena de caracteres* en la **referencia de registro de texto** de la *localización texto*.

SETTEXTINDEX tiene la misma semántica que una *cláusula de edición* en *WRITETEXT* en la que el *código de edición* es *T* y *anchura de cláusula* entrega el mismo valor que *expresión entera*, aplicada al **registro de texto** denotado por *localización texto*.

SETTEXTACCESS almacena una referencia a la localización entregada por *localización acceso* en la **referencia de acceso** de la *localización texto*.

propiedades estáticas: La clase de la llamada a rutina incorporada *GETTEXTRECORD* es la clase M-referencia, donde M es el modo **registro de texto** de la *localización texto*.

La clase de la llamada a rutina incorporada *GETTEXTINDEX* es la clase *&INT*-derivada.

La clase de la llamada a rutina incorporada *GETTEXTACCESS* es la clase M-referencia, donde M es el modo **acceso** de la *localización texto*.

La clase de la llamada a rutina incorporada *EOLN* es la clase *BOOL*-derivada.

Una rutina incorporada *GETTEXTRECORD* o *GETTEXTACCESS* tiene la misma **regionalidad** que la *localización texto*.

condiciones estáticas: El modo del argumento de *localización cadena de caracteres* de *SETTEXTRECORD* debe ser de **lectura compatible** con el **modo registro de texto** de la *localización texto*.

El modo del argumento de *localización acceso* de *SETTEXTACCESS* debe ser de **lectura compatible** con el modo **acceso** de la *localización texto*.

El argumento de *localización* en *SETTEXTRECORD* y *SETTEXTACCESS* debe tener la misma **regionalidad** que la *localización texto*.

condiciones dinámicas: Se produce la excepción *TEXTFAIL* si el argumento de *expresión entera* de *SETTEXTINDEX* entrega un valor menor que 0 o mayor que la **longitud de texto** de la *localización texto*.

ejemplo:

26.23 GETTEXTINDEX (output) *(1.2)*

8 Manejo de excepciones

8.1 Generalidades

Una excepción puede ser una excepción definida por el lenguaje, en cuyo caso puede tener un nombre definido por el lenguaje, una excepción definida por el usuario, o una excepción definida por la implementación. Una excepción definida por lenguaje será causada por la violación dinámica de una condición dinámica. Puede producirse cualquier excepción mediante la ejecución de una acción causar.

Cuando se produce una excepción, ésta puede tratarse, es decir se ejecutará una lista de sentencias de acción de un manejador adecuado.

El manejo de excepciones se define de forma que en cualquier sentencia se conoce estáticamente qué excepciones podrían ocurrir (es decir, se conoce estáticamente qué excepciones no pueden ocurrir) y para qué excepciones puede encontrarse un manejador adecuado, o qué excepciones pueden pasarse al punto de llamada de un procedimiento. Si se produce una excepción y no puede encontrarse un manejador para la misma, el programa tiene error.

Cuando se produce una excepción en una sentencia de acción o en una sentencia de declaración, la ejecución de la sentencia se efectúa hasta un grado no determinado, a menos que se indique otra cosa en el punto adecuado.

8.2 Manejadores

sintaxis:

<manejador> ::= *(1)*
ON { *<alternativa a elegir>* } * [**ELSE** *<lista de sentencias de acción>*] **END** *(1.1)*

<alternativa a elegir> ::= *(2)*
(*<lista de excepciones>*) : *<lista de sentencias de acción>* *(2.1)*

semántica: Se entra en un manejador si es conveniente para una excepción E, de acuerdo con 8.3. Si E es mencionada en una *lista de excepciones* en una *alternativa a elegir* en el *manejador*, se entra en la correspondiente *lista de sentencias de acción*; en otro caso, se entra en **ELSE** y se introduce la correspondiente *lista de sentencias de acción*.

Cuando se llega al final de la lista de *sentencias de acción* elegida, se terminan el *manejador* y la construcción a la cual está agregado.

condiciones estáticas: Todos los *nombres de excepción* de todas las ocurrencias de *listas de excepciones* deben ser diferentes.

condiciones dinámicas: Se produce la excepción *SPACEFAIL* si se entra en una lista de sentencias de acción y no pueden cumplirse los requisitos de almacenamiento.

ejemplos:

10.47 **ON**

(*ALLOCATEFAIL*): **CAUSE** *overflow*;

END *(1.1)*

8.3 Identificación del manejador

Cuando se produce una excepción E en una acción o módulo A o en una sentencia de datos o región D, puede tratarse la excepción mediante el manejador adecuado; es decir, en el manejador se ejecutará una lista de sentencias de acción o puede pasarse la excepción al punto de llamada de un procedimiento o, si nada de ello es posible, el programa tiene error.

Para cualquier acción o módulo A, o sentencia de datos o región D, puede determinarse estáticamente si, para una excepción dada E en A o D, puede encontrarse un manejador adecuado, o si la excepción puede pasarse al punto de llamada.

Un manejador adecuado para A o D con respecto a un nombre de excepción E se determina como sigue:

1) si se agrega a o se incluye en A o a D un manejador que mencione E en una *lista de excepciones* o que especifique **ELSE**, y se produce E en el dominio que circunda directamente al manejador, entonces este manejador es el adecuado con respecto a E;
2) en otro caso, si A o D está encerrado directamente por una acción encorchetada, un módulo o una región, el manejador adecuado (si está presente) es el manejador adecuado para la acción encorchetada, módulo o región con respecto a E;
3) en otro caso, si A o D se halla en el dominio de una definición de procedimiento, entonces:
   - si se agrega a la definición de procedimiento, un manejador que mencione E en una lista de excepciones o especifica **ELSE**, entonces este manejador es el adecuado;
   - en otro caso, si se menciona E en la lista de excepciones de la definición de procedimiento, entonces se provoca E en el punto de llamada;
   - en otro caso, no existe manejador definido por el usuario; sin embargo, en esta situación puede ser apropiado un manejador definido por la implementación (véase 13.5);

4) en otro caso, si A o D se halla en el dominio de una definición de proceso, entonces:

   - si se agrega a la definición de proceso, un manejador que mencione E, en una lista de excepciones o especifica **ELSE**, este manejador es el adecuado;
   - en otro caso, no existe manejador; sin embargo, en esta situación puede ser apropiado un manejador definido por la implementación (véase 13.5);

5) en otro caso, si A es una acción de una lista de sentencias de acción de un manejador, el manejador adecuado es el adecuado para la acción A' o la sentencia de datos o región D' con respecto a E a la que está agregado o en la que está incluido pero considerado como si este manejador no estuviera especificado.

Si se produce una excepción y la transferencia de control al manejador adecuado implica una salida de bloques, se liberará el almacenamiento local al salir del bloque.

9 Supervisión de tiempo

9.1 Generalidades

Se supone que un concepto de tiempo existe externamente al programa CHILL (sistema). CHILL no especifica las propiedades precisas del tiempo, pero da mecanismos para permitir la interacción de un programa con la concepción del tiempo en el mundo exterior.

9.2 Procesos temporizables

El concepto de proceso **temporizable** existe para identificar los puntos precisos durante la ejecución del programa en que puede ocurrir una interrupción de tiempo, es decir, cuando una supervisión de tiempo puede interferir con la ejecución normal de un proceso.

Un proceso se vuelve **temporizable** cuando llega a un punto preciso de la ejecución de ciertas acciones. CHILL establece que un proceso se vuelve **temporizable** durante la ejecución de acciones específicas. Una implementación, puede decidir que se vuelva un proceso **temporizable** durante la ejecución de acciones ulteriores.

9.3 Acciones de temporización

sintaxis:

<acción de temporización> ::= *(1)*
    <acción de temporización relativa> *(1.1)*
  | *<acción de temporización absoluta>* *(1.2)*
  | *<acción de temporización cíclica>* *(1.3)*

semántica: Una acción de temporización especifica supervisiones de tiempo del proceso ejecutante. Una supervisión de tiempo puede iniciarse, expirar y dejar de existir. A causa de la acción de temporización cíclica y de la jerarquización de las acciones de temporización, podrían asociarse varias supervisiones de tiempo con el mismo proceso.

Se produce una interrupción de tiempo cuando un proceso es **temporizable** y ha expirado por lo menos una de sus supervisiones de tiempo asociadas. La ocurrencia de una interrupción de tiempo implica que la primera supervisión de tiempo expirada deja de existir; además, conduce a la transferencia de control asociada con esa supervisión de tiempo en el proceso supervisado. Si el proceso supervisado había sido demorado, es reactivado.

Las supervisiones de tiempo también dejan de existir cuando el control sale de la acción de temporización que las inició.

NOTA – Si la transferencia de control hace que el proceso abandone una región, la región será liberada (véase 11.2.1).

9.3.1 Acción de temporización relativa

sintaxis:

<acción de temporización relativa> ::= *(1)*
    AFTER *<valor primitivo de duración>* [**DELAY**] **IN**
    <lista de sentencias de acción> *<manejador de temporización>* **END** *(1.1)*

<manejador de temporización> ::= *(2)*
    TIMEOUT *<lista de sentencias de acción>* *(2.1)*

semántica: Se evalúa *el valor primitivo de duración*, se inicia una supervisión de tiempo, y a continuación se entra en la *lista de sentencias de acción*.

Si no se especifica **DELAY**, la supervisión de tiempo se inicia antes de entrar en la *lista de sentencias de acción*; en otro caso se inicia cuando el proceso de ejecución se vuelve **temporizable** en el punto de ejecución especificado por la *sentencia de acción* de la *lista de sentencias de acción*.

Si se especifica **DELAY**, la supervisión de tiempo deja de existir si se ha iniciado y el proceso ejecutante deja de ser **temporizable**.

La supervisión de tiempo expira si no ha dejado de existir una vez transcurrido el periodo de tiempo especificado desde la iniciación.

El control asociado con la supervisión de tiempo se transfiere a la *lista de sentencias de acción* del *manejador de temporización*.

condiciones estáticas: Si se especifica **DELAY**, la *lista de sentencias de acción* ha de consistir precisamente en una sentencia de acción que puede a su vez hacer que el proceso ejecutante se vuelva **temporizable**.

condiciones dinámicas: Se produce la excepción *TIMERFAIL* si la iniciación de la supervisión de tiempo falla por una razón definida por la implementación.

9.3.2 Acción de temporización absoluta

sintaxis:

<acción de temporización absoluta> ::= *(1)*
AT*<valor primitivo tiempo absoluto>* **IN**
<lista de sentencias de acción> *<manejador de temporización>* **END** *(1.1)*

semántica: Se evalúa el *valor primitivo tiempo absoluto*, se inicia una supervisión de tiempo, y a continuación se entra en la *lista de sentencias de acción*.

La supervisión de tiempo expira si no ha dejado de existir en el punto de tiempo especificado (o después del mismo).

El control asociado con la supervisión de tiempo se transfiere a la *lista de sentencias de acción* del *manejador de temporización*.

condiciones dinámicas: Se produce la excepción *TIMERFAIL* si la iniciación de la supervisión de tiempo falla por una razón definido por la implementación.

9.3.3 Acción de temporización cíclica

sintaxis:

<acción de temporización cíclica> ::= *(1)*
CYCLE *<valor primitivo de duración>* **IN**
<lista de sentencias de acción> **END** *(1.1)*

semántica: La acción de temporización cíclica tiene por objeto asegurar que el proceso ejecutante introduzca la lista de sentencias de acción a intervalos precisos sin derivas acumuladas (esto implica que el tiempo de ejecución para la *lista de sentencias de acción* debe, en promedio ser menor que el valor de duración especificado). Se evalúa el *valor primitivo de duración*, se inicia una supervisión de tiempo relativo, y a continuación, se introduce la *lista de sentencias de acción*.

La supervisión de tiempo expira si no ha dejado de existir una vez transcurrido el tiempo desde la iniciación. Indivisiblemente con la expiración, se inicia una nueva supervisión de tiempo con el mismo valor de duración.

El control asociado con la supervisión de tiempo se transfiere al comienzo de la *lista de sentencias de acción*.

Obsérvese que la acción de temporización cíclica sólo puede terminar por una transferencia de control fuera de la misma.

propiedades dinámicas: El proceso ejecutante se vuelve **temporizable** cuando el control alcanza el final de la *lista de sentencias de acción*.

condiciones dinámicas: Se produce la excepción *TIMERFAIL* si cualquier iniciación de una supervisión de tiempo falla por una razón definida por la implementación.

9.4 Rutinas incorporadas relacionadas con el tiempo

sintaxis:

<llamada a rutina incorporada que entrega un valor de tiempo> ::= *(1)*
<llamada a rutina incorporada que entrega una duración> *(1.1)*
| *<llamada a rutina incorporada que entrega un tiempo absoluto>* *(1.2)*

semántica: Es probable que las implementaciones tengan exigencias y capacidades muy distintas en términos de precisión y de gama de valores de tiempo. Las rutinas incorporadas que se definen más adelante tienen por objeto tratar esas diferencias de una manera portable.

9.4.1 Rutinas incorporadas que entregan una duración

sintaxis:

<llamada a rutina incorporada que entrega una duración> ::= *(1)*
MILLISECS (*<expresión entera>*) *(1.1)*
| *SECS* (*<expresión entera>*) *(1.2)*
| *MINUTES* (*<expresión entera>*) *(1.3)*
| *HOURS* (*<expresión entera>*) *(1.4)*
| *DAYS* (*<expresión entera>*) *(1.5)*

semántica: Una llamada a rutina incorporada que entrega una duración entrega un valor de duración con una precisión definida por la implementación que puede variar de una a otra (por ejemplo, *MILLISECS (1000)* y *SECS (1)* pueden entregar valores de duración diferentes); este valor es la aproximación más cercana, en la precisión elegida, al periodo de tiempo indicado. El argumento de *MILLISECS, SECS, MINUTES, HOURS* y *DAYS* indica un punto en el tiempo expresado en milisegundos, segundos, minutos, horas y días, respectivamente.

propiedades estáticas: La clase de una llamada a rutina incorporada que entrega una duración es la clase *DURATION*-derivada.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si la implementación no puede entregar un valor de duración que denote el periodo de tiempo indicado.

9.4.2 Rutina incorporada que entrega un tiempo absoluto

sintaxis:

<llamada a rutina incorporada que entrega un tiempo absoluto> ::= *(1)*
ABSTIME ([[[[[[*<expresión año>* ,] *<expresión mes>* ,]
<expresión día> ,] *<expresión hora>* ,]
<expresión minuto> ,]
<expresión segundo>]) *(1.1)*

<expresión año> ::= *(2)*
<expresión entera> *(2.1)*

<expresión mes> ::= *(3)*
<expresión entera> *(3.1)*

<expresión día> ::= *(4)*
<expresión entera> *(4.1)*

<expresión hora> ::= *(5)*
<expresión entera> *(5.1)*

<expresión minuto> ::= *(6)*
<expresión entera> *(6.1)*

<expresión segundo> ::= *(7)*
<expresión entera> *(7.1)*

semántica: La llamada a rutina incorporada *ABSTIME* entrega un valor de tiempo absoluto que denota el punto de tiempo en el calendario gregoriano indicado en la lista de parámetros. Los parámetros indican los componentes de tiempo en el orden siguiente: año, mes, día, hora, minuto y segundo. Cuando se omiten parámetros de orden superior, el punto de tiempo indicado es el siguiente punto de tiempo que concuerda con los parámetros de orden inferior presentes (por ejemplo, *ABSTIME* (*15, 12, 00, 00*) denota mediodía del día 15 de este mes o del siguiente).

Cuando no se especifica ningún parámetro, se entrega un valor de tiempo absoluto que denota el punto de tiempo presente.

propiedades estáticas: La clase de la llamada a rutina incorporada que entrega un tiempo absoluto es la clase *TIME*-derivada.

condiciones dinámicas: Se produce la excepción *RANGEFAIL* si la implementación no puede entregar un valor de tiempo absoluto que denote el punto de tiempo indicado.

9.4.3 Llamada a rutina incorporada que entrega una temporización

sintaxis:

<llamada a rutina incorporada simple que entrega una temporización> ::= *(1)*
WAIT ( ) *(1.1)*
| *EXPIRED* ( ) *(1.2)*
| *INTTIME* (*<valor primitivo tiempo absoluto>*, [[[[*<localización año>*
<localización mes> ,] *<localización día>* ,]
<localización hora> ,] *<localización minuto>* ,]
<localización segundo>) *(1.3)*

<localización año> ::= *(2)*
<localización entera> *(2.1)*

<localización mes> ::= *(3)*
<localización entera> *(3.1)*

<localización día> ::= *(4)*
<localización entera> *(4.1)*

<localización hora> *(5)*
<localización entera> *(5.1)*

<localización minuto> *(6)*
<localización entera> *(6.1)*

<localización segundo> *(7)*
<localización entera> *(7.1)*

semántica: *WAIT* hace **temporizable** el proceso ejecutante incondicionalmente: su ejecución sólo puede terminarse por una interrupción de tiempo. (Obsérvese que el proceso permanece activo en el sentido de CHILL.)

EXPIRED hace **temporizable** el proceso ejecutante si una de sus supervisiones de tiempo asociado ha expirado; en otros casos no tiene efecto.

INTTIME asigna a las localizaciones enteras especificadas una representación entera del punto de tiempo en el calendario gregoriano especificado por el *valor primitivo tiempo absoluto*. Las localizaciones pasadas como argumentos reciben los componentes de tiempo en el orden siguiente: año, mes, día, hora, minuto y segundo.

condiciones estáticas: Todas las localizaciones enteras especificadas deben ser **referenciables** y sus modos pueden no tener la **propiedad de lectura solamente**.

propiedades dinámicas: *WAIT* hace **temporizable** el proceso ejecutante.

EXPIRED hace **temporizable** el proceso ejecutante si expira una supervisión de tiempo asociada con el mismo.

10 Estructura del programa

10.1 Generalidades

La estructura del programa está determinada por *acción condicional*, *acción de caso*, *acción hacer*, *acción demorar y elegir*, *bloque principio-fin*, *módulo*, *región*, *módulo de espec*, *región de espec*, *contexto*, *acción recibir y elegir*, *definición de procedimiento y definición de proceso*, los cuales determinan el alcance de los nombres y el tiempo de vida de las localizaciones creadas en los mismos.

- La palabra bloque se utilizará para denotar:
  - la *lista de sentencias de acción* en una *acción hacer*, incluido el *contador de bucle* y *control mientras*;
  - la *lista de sentencias de acción* en una *cláusula entonces* en una *acción condicional*;
  - la *lista de sentencias de acción* en una *alternativa de caso* en una *acción de caso*;
  - la *lista de sentencias de acción* en una *alternativa de demora* en una *acción demorar y elegir*;
  - el *bloque principio-fin*;
  - la *definición de procedimiento*, excluida la *espec de resultado* y la *espec de parámetro* de todos los *parámetros formales* de la *lista de parámetros formales*;
  - la *definición de proceso*, excluida la *espec de parámetro* de todos los *parámetros formales* de la *lista de parámetros formales*;
  - la *lista de sentencias de acción*, en una *alternativa recibir tampón*, o en una *alternativa recibir señal*, incluidas las *ocurrencias de definición de la lista de ocurrencias de definición* después de **IN**;
  - la *lista de sentencias de acción* después de **ELSE** en una *acción condicional* o *acción de caso*, o una *acción recibir y elegir* o un *manejador*;
  - la *alternativa a elegir* en un *manejador*;
  - la *lista de sentencias de acción* en una *acción de temporización relativa*, una *acción de temporización absoluta*, una *acción de temporización cíclica* o en un *manejador de temporización*.
- La palabra modulión se utilizará para denotar:
  - un *módulo* o *región*, excluidos *lista de contexto* y *ocurrencia de definición*, si existen;
  - un *módulo de espec* o *región de espec*, excluido *lista de contexto*, si existen;
  - un *contexto*;
  - la especificación junto con el cuerpo correspondiente de un *modo moreta*;
  - una *plantilla* junto con el cuerpo correspondiente.
- La palabra grupo denota un bloque o un modulión.
- La palabra dominio o dominio de un grupo denota la parte del grupo no circundada (véase 10.2) por un grupo interno. Si BM es un modo moreta y DM es un sucesor directo de BM, entonces BM_p -$BM_{CD} \cup DM_p$ forman un dominio. Para la visibilidad de los componentes internos de modos moreta, el dominio de un sucesor está anidado inmediatamente en la parte especificación de su predecesor directo; este anidamiento aparece al final de la parte especificación.

Un grupo influye en el alcance de cada nombre creado en su dominio. Los nombres son creados por *ocurrencias de definición*:

- Una *ocurrencia de definición* en la *lista de ocurrencias de definición* de una *declaración, definición de modo*, o *definición de sinónimo*, o que aparece en una *definición de señal* crea un *nombre* en el dominio en que está situada la *declaración*, *definición de modo*, *definición de sinónimo* o *definición de señal*, respectivamente.
- Una *ocurrencia de definición* en un *modo conjunto* crea un nombre en el dominio que circunda directamente al *modo conjunto*.
- Una *ocurrencia de definición* que aparece en la *lista de ocurrencias de definición* en una *lista de parámetros formales* crea un nombre en el dominio de la *definición de procedimiento* o la *definición de proceso* asociada.
- Una *ocurrencia de definición* que precede a un signo dos puntos seguido por una *acción, región*, *definición de procedimiento* o *definición de proceso* crea un nombre en el dominio en que está situada la *acción*, *región*, *definición de procedimiento*, *definición de proceso*, respectivamente.
- Una *ocurrencia de definición* (virtual) introducida por una *parte con* o en un *contador de bucle* crea un nombre en el dominio del bloque de la *acción hacer* asociada.
- Una *ocurrencia de definición* en la *lista de ocurrencias de definición* de una *alternativa recibir tampón* o una *alternativa recibir señal* crea un nombre en el dominio del bloque de la *alternativa recibir tampón* o la *alternativa recibir señal* asociada, respectivamente.
- Una *ocurrencia de definición* (virtual) para un nombre predefinido por el lenguaje o definido por la implementación crea un nombre en el dominio del proceso imaginario más externo (véase 10.8).

Los lugares en que se utiliza un nombre se denominan ocurrencias aplicadas del nombre. Las reglas de vinculación asocian una *ocurrencia de definición* con cada ocurrencia aplicada del nombre (véase 12.2.2).

Un nombre tiene un cierto alcance, es decir aquella parte del programa donde puede verse su definición o declaración y, en consecuencia, donde puede usarse libremente. Se dice que el nombre es **visible** en dicha parte. Las localizaciones tienen un cierto tiempo de vida que es la parte del programa donde existen. Los bloques determinan la visibilidad de los nombres y el tiempo de vida de las localizaciones creadas en ellos. Los moduliones determinan la visibilidad solamente; el tiempo de vida de las localizaciones creadas en el dominio de un modulión será el mismo que si hubieran sido creadas en el dominio del primer bloque circundante. Los moduliones permiten restringir la visibilidad de los nombres. Por ejemplo, un nombre creado en el dominio de un módulo no será automáticamente **visible** en módulos internos o externos aunque su tiempo de vida lo permita.

10.2 Dominios y anidamiento

sintaxis:

<cuerpo de principio-fin> ::= *(1)*
<lista de sentencias de datos> <lista de sentencias de acción> *(1.1)*

<cuerpo de procedimiento> ::= *(2)*
<lista de sentencias de datos><lista de sentencias de acción> *(2.1)*

<cuerpo de proceso> ::= *(3)*
<lista de sentencias de datos> <lista de sentencias de acción> *(3.1)*

<cuerpo de módulo> ::= *(4)*
{ *<sentencia de datos>* | *<sentencia de visibilidad>* | *<región>* |
<región de espec> }* *<lista de sentencias de acción>* *(4.1)*

<cuerpo de región> ::= *(5)*
{ *<sentencia de datos>* | *<sentencia de visibilidad>* }* *(5.1)*

<cuerpo de módulo de espec> ::= *(6)*
{ *<cuasi sentencia de datos>* | *<sentencia de visibilidad>* |
<módulo de espec> | *<región de espec>* }* *(6.1)*

<cuerpo de región de espec> ::= *(7)*
{*<cuasi sentencia de datos>* | *<sentencia de visibilidad>* }* *(7.1)*

<cuerpo de contexto> ::= *(8)*
{ *<cuasi sentencia de datos>* | *<sentencia de visibilidad>* |
<módulo de espec> |*<región de espec>* }* *(8.1)*

<lista de sentencias de acción> ::= *(9)*
{ *<sentencia de acción>* }* *(9.1)*

<lista de sentencias de datos> ::= *(10)*
{ *<sentencia de datos>*}* *(10.1)*

<sentencia de datos> ::= *(11)*
<sentencia de declaración> *(11.1)*
| *<sentencia de definición>* *(11.2)*

<sentencia de definición> ::= *(12)*
<sentencia de definición de sínmodo> *(12.1)*
| *<sentencia de definición de neomodo>* *(12.2)*
| *<sentencia de definición de sinónimo>* *(12.3)*
| *<sentencia de definición de procedimiento>* *(12.4)*
| *<sentencia de definición de proceso>* *(12.5)*
| *<sentencia de definición de señal>* *(12.6)*
| *<plantilla>* *(12.7)*
| *<vacía>*; *(12.8)*

semántica: Cuando se produce la entrada del dominio de un bloque, se efectúan todas las inicializaciones ligadas al tiempo de vida de las localizaciones creadas por la entrada en el bloque. Seguidamente, se realizan las inicializaciones ligadas al dominio del bloque, las evaluaciones, posiblemente dinámicas, de las declaraciones de identidad-loc, las inicializaciones ligadas al dominio en las regiones y las acciones, en el orden en que están textualmente especificadas.

Cuando se produce la entrada del dominio de un modulión, se efectúan las inicializaciones ligadas al dominio, las evaluaciones posiblemente dinámicas de las declaraciones de identidad-loc, la inicialización ligada al dominio en las regiones y las acciones (si el modulión es un módulo) en el dominio del modulión, en el orden en que están textualmente especificadas.

Una sentencia de datos, una acción, un módulo o una región se termina completando o terminando un manejador que lleva agregado.

Cuando ha de terminarse un grupo G, primero se cierran todas las localizaciones TASK y REGION (localizaciones RTL), que dependen de G (véase 12.2.6). La terminación de G se finaliza cuando todas esas RTL son *completadas* (véase 11.6).

Cuando se termina una inicialización ligada al dominio, una declaración de identidad-loc, acción, módulo, procedimiento o proceso, la ejecución se reanuda de la manera siguiente, dependiendo de la sentencia o del tipo de terminación:

- si la sentencia se termina completando la ejecución de un manejador, la ejecución se reanuda con la sentencia siguiente;
- en otro caso, si es una acción que implique una transferencia de control, la ejecución se reanuda con la sentencia definida para esa acción (véanse 6.5, 6.6, 6.8 y 6.9);
- en otro caso, si es un procedimiento, el control se retorna al punto de llamada (véase 10.4);
- en otro caso, si es un proceso, la ejecución de ese proceso (o del programa, si es el proceso más exterior) finaliza (véase 11.1) y la ejecución se reanuda (posiblemente) con otro proceso;
- en otro caso, el control se pasa a la sentencia siguiente.

propiedades estáticas: Todo dominio está directamente encerrado en cero o más grupos, como sigue:

- Si el dominio es el de una *acción hacer*, un *bloque principio-fin*, una *definición de procedimiento*, o una *definición de proceso*, entonces está directamente encerrado en el grupo en cuyo dominio están situadas la *acción hacer*, *bloque principio-fin*, *definición de procedimiento*, o *definición de proceso*, respectivamente, y solamente en ese grupo.
- Si el dominio es la *lista de sentencias de acción* de una *acción de temporización* o el *manejador de temporización*, o una de las *listas de sentencias de acción* de una *acción condicional*, una *acción de caso* o una *acción demorar y elegir*, entonces está directamente encerrado en el grupo en cuyo dominio está situada la *acción de temporización*, *manejador de temporización*, *acción condicional*, *acción de caso* o *acción demorar y elegir*, y solamente en ese grupo.
- Si el dominio es la *lista de sentencias de acción*, o una *alternativa recibir tampón* o una *alternativa recibir señal*, o la *lista de sentencias de acción* que sigue a **ELSE** en una *acción recibir tampón* o una *acción recibir señal y elegir*, entonces está directamente encerrado en el grupo en cuyo dominio está situada la *acción recibir tampón y elegir* o la *acción recibir señal y elegir*, y solamente en ese grupo.
- Si el dominio es la *lista de sentencias de acción* en una *alternativa a elegir* o la *lista de sentencias de acción* que sigue a **ELSE** en un *manejador* no agregado a un grupo, entonces está directamente encerrado en el grupo en cuyo dominio está situada la sentencia a la cual está agregado el *manejador*, y solamente en ese grupo.
- Si el dominio es una *alternativa a elegir* o una *lista de sentencias* de acción situada después de **ELSE** de un *manejador* agregado a un grupo, entonces está directamente encerrado en el grupo al cual está agregado el *manejador*, y solamente en ese grupo.
- Si el dominio es un *módulo*, *región*, *módulo de espec* o *región de espec*, entonces está directamente encerrado en el grupo en cuyo dominio está situado, y también encerrado directamente en el contexto que precede inmediatamente al *módulo*, *región*, *módulo de espec* o *región de espec*, si existe. Este es el único caso en que un dominio tiene más de un grupo directamente circundante.
- Si el dominio es un *contexto*, entonces está directamente encerrado en el *contexto* que lo precede inmediatamente. Si no hay tal *contexto*, no tiene grupo directamente circundante.

Un dominio tiene dominios directamente circundantes que son los dominios de los grupos directamente circundantes. Una sentencia tiene un grupo directamente circundante único, que es el grupo donde está situada la sentencia. Se dice que un dominio encierra directamente un grupo (dominio) única y exclusivamente si el dominio es un dominio directamente circundante del grupo (dominio).

Se dice que una sentencia (dominio) está rodeada por un grupo única y exclusivamente si el grupo es el grupo directamente circundante de la sentencia (dominio) o un dominio directamente circundante está rodeado por el grupo.

Se dice que se entra en un dominio cuando:

- Dominio de módulo: se ejecuta el módulo como una acción (por ejemplo, no se dice que se entra en el módulo cuando una acción ir a transfiere el control a un nombre de **etiqueta** definido dentro del módulo).
- Dominio principio-fin: se ejecuta el bloque principio-fin como una acción.
- Dominio de región: se encuentra la región (es decir, no se dice que se entra en la región cuando se llama a uno de sus procedimiento **críticos)**.
- Dominio de procedimiento: se produce la entrada del procedimiento mediante una llamada a procedimiento.
- Dominio de proceso: se activa el proceso mediante la evaluación de una expresión arrancar.
- Dominio de hacer: se ejecuta la acción hacer como una acción después de la evaluación de las expresiones o localizaciones en la parte de control.
- Dominio de alternativa recibir tampón, dominio de alternativa recibir señal: se ejecuta la alternativa al recibir un valor tampón o una señal.
- Dominio de alternativa a elegir: se ejecuta la alternativa a elegir a causa de una excepción.
- Otros dominios de bloque: se produce la entrada de una lista de sentencias de acción.

Se dice que se entra en una lista de sentencias de acción solamente en el caso en que la primera acción, si está presente, reciba el control desde fuera de la lista de sentencias de acción.

Un dominio es un **cuasi** dominio si es un *módulo de espec*, una *región de espec* o un *contexto*, y en los demás casos es un dominio **real**.

Una *ocurrencia de definición* es una **cuasi** *ocurrencia de definición* si:

- está rodeada por un *contexto* y no por un módulo o una región; o
- está rodeada por un *módulo de espec simple* o por una *región de spec simple*; o
- no está rodeada por uno de los dominios citados anteriormente y está rodeada por una *espec de módulo* o una *espec de región*, y está contenida en una *cuasi declaración*, una *cuasi sentencia de definición de procedimiento* o una *cuasi sentencia de definición de proceso*,

en otro caso, es una *ocurrencia de definición* **real**.

10.3 Bloques principio-fin

sintaxis:

<bloque principio-fin> ::= *(1)*
BEGIN *<cuerpo de principio-fin>* **END** *(1.1)*

semántica: Un bloque principio-fin es una acción (compuesta) que posiblemente contiene declaraciones y definiciones locales. Determina la visibilidad de los nombres creados localmente y los tiempos de vida de las localizaciones creadas localmente (véanse 10.9 y 12.2).

condiciones dinámicas: Se produce una excepción *SPACEFAIL* si no pueden cumplirse los requisitos de almacenamiento.

ejemplos: véanse *15.73 - 15.90*

10.4 Definiciones de procedimiento

sintaxis:

<sentencia de definición de procedimiento> ::= *(1)*
<ocurrencia de definición> : *<definición de procedimiento>*
[*<manejador>*] [*<cadena de nombre simple>*] ; *(1.1)*
| *<ejemplificación de procedimiento genérico>* *(1.2)*

<definición de procedimiento> ::= *(2)*
PROC ([*<lista de parámetros formales>*]) [*<espec de resultado>*]
[**EXCEPTIONS** (*<lista de excepciones>*)]
<lista de atributos de procedimiento>;
<cuerpo de procedimiento> **END** *(2.1)*

<lista de parámetros formales> ::= *(3)*
<parámetro formal> { , *<parámetro formal>* }* *(3.1)*

<parámetro formal> ::= *(4)*
<lista de ocurrencias de definición> <espec de parámetro> *(4.1)*

<lista de atributos de procedimiento> ::= *(5)*
[*<generalidad>*] *(5.1)*

<generalidad> ::= *(6)*
GENERAL *(6.1)*
| **SIMPLE** *(6.2)*
| **INLINE** *(6.3)*

<sentencia de firma de procedimiento guardado> ::= *(7)*
<ocurrencia de definición> :
<firma de procedimiento guardado> [*<cadena de nombre simple >*] ; *(7.1)*

<firma de procedimiento guardado> ::= *(8)*
PROC ([*< lista de parámetros>*]) [*<espec de resultado>*]
[**EXCEPTIONS** (*<lista de excepciones >*)]
<lista de atributos de procedimientos guardados > **END** *(8.1)*

<sentencia de definición de procedimiento guardado> ::= *(9)*
<ocurrencia de definición> : *<definición de procedimiento guardado >*
[[*<manejador>*] [*<cadena de nombre simple>*] ; *(9.1)*

<definición de procesamiento guardado > ::= *(10)*
PROC ([*<lista de parámetros formales>*]) [*<espec de resultado>*]
[**EXCEPTIONS** (*<lista de excepciones>*)]
<lista de atributos de procedimientos guardados> ;
<cuerpo de procedimiento> **END** *(10.1)*

<lista de atributos de procedimientos guardados> ::= *(11)*
[**GENERAL**] *(11.1)*
| [**SIMPLE**] [*<lista de atributos de procedimientos componentes simples>*]
<parte afirmación> *(11.2)*
| [**INLINE**] [*<lista de atributos de procedimientos componentes en línea>*] *(11.3)*

<lista de atributos de procedimientos componentes simples> ::= *(12)*
<lista de atributos de procedimientos componentes en línea> *(12.1)*
| **DESTR** *(12.2)*
| **INCOMPLETE** *(12.3)*
| [**REIMPLEMENT**] [**FINAL**] *(12.4)*

<lista de atributos de procedimientos componentes en línea> ::= *(13)*
CONSTR *(13.1)*
| **FINAL** *(13.2)*

<parte afirmación > ::= *(14)*
[**PRE** (*<expresión booleana>*)]
[**POST** (*<expresión booleana>*)] *(14.1)*

sintaxis derivada: Un *parámetro formal*, cuando la *lista de ocurrencias de definición* contiene más de una *ocurrencia de definición*, se deriva de varias ocurrencias de *parámetros formales* separadas por comas, una para cada *ocurrencia de definición*, cada una de las cuales tiene la misma *espec de parámetro*. Por ejemplo: *i*, *j INT* **LOC** se deriva de *i INT* **LOC**, *j INT* **LOC**.

semántica: Una sentencia de definición de procedimiento define una secuencia (posiblemente) parametrizada de acciones que pueden llamarse desde diferentes lugares del programa. Se termina el procedimiento y se devuelve el control al punto de llamada ejecutando una acción retornar alcanzando el final del *cuerpo de procedimiento*, o terminando un manejador añadido a la definición de procedimiento (traspaso de los bornes). Pueden especificarse distintos grados de complejidad del procedimiento, que son los siguientes:

a) los procedimientos **simples** (**SIMPLE**) son procedimientos que no pueden manipularse dinámicamente. No se tratan como valores, es decir, no pueden almacenarse en una localización procedimiento ni transferirse como parámetros a una llamada a procedimiento ni obtenerse como resultado de una llamada a procedimiento;

b) los procedimientos **generales** (**GENERAL**) no tienen las restricciones de los procedimientos **simples**, pudiendo tratarse como valores de procedimiento;

c) los procedimientos **en línea** (**INLINE**) tienen las mismas restricciones que los procedimientos **simples** y no pueden ser **recursivos**. Poseen la misma semántica que los procedimientos normales, pero el compilador insertará el código objeto generado en el punto de invocación, en lugar de generar el código para llamar efectivamente al procedimiento.

Sólo los procedimientos **simples** y **generales** son **recursivos**.

Una sentencia de definición de procedimiento guardado define una secuencia (posiblemente) parametrizada de acciones que puede ser llamadas desde diferentes lugares en el programa. El procedimiento se termina y el control se retorna al punto de llamada sea ejecutando una *acción retornar* o llegando al final del *cuerpo de procedimiento,* o terminando un *manejador* agregado a la definición de procedimiento.

Cuando el procedimiento se define en un *modo moreta*, se denomina un **procedimiento componente**. Diferentes tipos de procedimientos componentes **simples** o **en línea** definidos en modos moreta pueden especificarse como sigue:

a) un procedimiento componente **constr (CONSTR)** es un constructor que puede utilizarse para inicializar localizaciones moreta automáticamente cuando se crean estática o dinámicamente;

b) un procedimiento componente **destr (DESTR)** es un destructor que puede utilizarse para finalizar localizaciones moreta automáticamente cuando se destruyen estática o dinámicamente;

c) un procedimiento componente **incompleto (INCOMPLETE)** sólo tiene una firma y no tiene cuerpo;

d) un procedimiento componente **reimplementan (REIMPLEMENT)** al que se da un nuevo cuerpo y posiblemente nuevas afirmaciones;

e) un procedimiento componente **final (FINAL)** es un procedimiento que no puede reimplementarse en un modo moreta derivado.

Pueden especificarse diferentes tipos de *parte afirmación* para procedimientos componentes **simples**:

a) una parte **pre**afirmación (**PRE**) que se verifica automáticamente antes de ejecutar el cuerpo del procedimiento correspondiente;

b) una **pos**afirmación (**POST**) que se verifica automáticamente después de ejecutar el cuerpo del procedimiento correspondiente y antes del retorno al punto de llamada.

Sólo los procedimientos **simples** (salvo los *procedimientos componentes* con los atributos **constr** o **destr** o con visibilidad **pública** en un modo **región**) y los procedimientos **generales** pueden ser **recursivos**.

Un procedimiento puede retornar un valor, o puede retornar una localización (indicada por el atributo **LOC** en la espec de resultado).

La *ocurrencia de definición* que precede a la definición de procedimiento define el nombre del procedimiento.

transferencia de parámetros

Hay básicamente dos procedimientos de transmisión de parámetros: la "transferencia por valor" (**IN**, **OUT** e **INOUT**) y la "transferencia por localización" (**LOC**).

transmisión por valor

En la transferencia de parámetros por valor, se pasa un valor como parámetro a un procedimiento y se almacena en una localización local del modo parámetro especificado. El efecto es el mismo que si, al principio de la llamada a procedimiento, se encontrase la declaración de localización:

DCL <*ocurrencia de definición de parámetro formal*> <*modo*> *::=* <*parámetro efectivo*>

para las *ocurrencias de definición* del *parámetro formal*. Sin embargo, el procedimiento comienza después de que se hayan evaluado los parámetros efectivos. Puede especificarse, facultativamente, la palabra clave **IN** para indicar explícitamente la transferencia por valor.

Si se especifica el atributo **INOUT**, el valor del parámetro efectivo se obtiene de una localización, restituyéndose en la localización efectiva el valor actual del parámetro formal inmediatamente antes del retorno.

El efecto de **OUT** es el mismo de **INOUT**, salvo que el valor inicial de la localización efectiva no se copia en la localización del parámetro formal tras el comienzo del procedimiento. En consecuencia, el parámetro formal tiene un valor inicial **indefinido**. No es necesario efectuar la operación de restitución del almacenamiento si el procedimiento produce una excepción en el punto de llamada.

transferencia por localización

En la transferencia del parámetro por localización, se transfiere una localización (posiblemente en modo dinámico) en forma de parámetro al cuerpo de procedimiento. Solo localizaciones **referenciables** pueden transferirse de esta forma. El efecto es el mismo que en el punto de comienzo del procedimiento si se encontrase la sentencia de declaración de identidad-loc:

DCL <*ocurrencia de definición*> <*modo*>

LOC [**DYNAMIC**] *::=* <*parámetro efectivo*> ;

para las *ocurrencias de definición* del *parámetro formal*. Sin embargo, el procedimiento comienza después de que se hayan evaluado los parámetros efectivos.

Si se especifica un *valor* que no es una *localización*, se creará implícitamente y se pasará en el momento de la llamada una localización que contenga el valor especificado. El tiempo de vida de la localización creada es la llamada a procedimiento. El modo de la localización creada es dinámico si el valor tiene una clase dinámica.

transmisión de resultado

Un procedimiento puede proporcionar un valor o una localización. En el primer caso, se especifica un *valor* en cualquier *acción resultar*, y en el segundo, una *localización* (véase 6.8). Si no se da el atributo **NONREF** en la *espec de resultado*, la *localización* debe ser **referenciable**. El valor o localización proporcionado están determinados por la acción resultar ejecutada más recientemente antes del retorno. Si se produce el retorno de un procedimiento con una espec de retorno sin haberse ejecutado una acción resultar, el procedimiento entrega un valor o localización **indefinida**. En este caso la llamada a procedimiento no puede utilizarse como una llamada a procedimiento que entrega una localización (véase 4.2.11), ni como una llamada a procedimiento que entrega un valor (véase 5.2.13), sino simplemente como una acción llamar (véase 6.7).

propiedades estáticas: Una *ocurrencia de definición* en una *sentencia de definición de procedimiento* define un nombre **de procedimiento**.

Un nombre de **procedimiento** lleva adjunta una *definición de procedimiento*, que es la *definición de procedimiento* en la sentencia en que se define el nombre de **procedimiento**.

Un nombre de **procedimiento** tiene asociadas las propiedades que siguen, definidas por su *definición de procedimiento:*

- Tiene una lista de **especs de parámetro**, definidas por las ocurrencias de *espec de parámetro* en la *lista de parámetros formales*; cada parámetro consta de un modo y posiblemente de un atributo de parámetro.
- Tiene, posiblemente una **espec de resultado**, que consta de un modo y de un atributo de resultado opcional.
- Tiene una lista, posiblemente vacía, de nombres de excepción, que son los mencionados en la *lista de excepciones*.
- Tiene una **generalidad**, que es, si se especifica *generalidad*, ya sea **general**, o **simple**, o **en línea**, según que se especifique **GENERAL**, **SIMPLE** o **INLINE**; en otro caso, se especifica por defecto en la implementación **general** o **simple**. Si el nombre de **procedimiento** se define dentro de un bloque o una región, su **generalidad** es **simple**. Si un procedimiento se define en un modo moreta y tiene visibilidad **pública**, su generalidad es **simple** o **en línea**.
- Tiene una **recursividad** que es **recursiva**. Sin embargo, si la **generalidad** es **en línea**, o si el nombre de **procedimiento** es **crítico** (véase 11.2.1), la **recursividad** es **no recursiva**.
- Un procedimiento componente tiene la generalidad **en línea** si se especifica el atributo **INLINE**. En otro caso tiene la generalidad **SIMPLE** por defecto.

Un nombre de **procedimiento** que es **general**, es un **nombre** de **procedimiento general**. Un nombre **de procedimiento general** tiene asociado un modo procedimiento formado como sigue:

PROC ([<*lista de parámetros*>]) [<*espec de resultado*>]

[**EXCEPTIONS** (<*lista de excepciones*>)]

donde <*espec de resultado*>, si existe, y <*lista de excepciones*> son las mismas que en su *definición de procedimiento* y *lista de parámetros* es la secuencia de ocurrencias de <*espec de parámetro*> en la *lista de parámetros formales*, separados por comas.

Un nombre definido en una *lista de ocurrencias de definición* en el *parámetro formal* es un nombre de **localización** única y exclusivamente si la *espec de parámetro* del *parámetro formal* no contiene el atributo **LOC**. Si lo contiene, se trata de un nombre de **identidad-loc**. Cualquiera de estos nombres de **localización** o de **identidad-loc**, es **referenciable**.

Un procedimiento componente modo moreta de un modo moreta M tiene una poscondición completa CPM que se define como sigue:

a) si M no tiene un modo base inmediato, CMP = parte **post**;

b) Si M tiene el modo base inmediato B, CPM = CPB ^ parte **post**, donde CPB es la poscondición completa de B.

condiciones estáticas: Si un nombre de **procedimiento** es **intrarregional** (véase 11.2.2), su definición de procedimiento no debe especificar **GENERAL**.

Si un nombre de **procedimiento** es **crítico** (véase 11.2.1), su definición no puede especificar **GENERAL**.

Si un procedimiento componente **simple** tiene cualquier *parte afirmación*, el nombre del procedimiento debe tener visibilidad **pública**.

Si un procedimiento componente guardado **simple** o **en línea** tiene el atributo **FINAL**, el nombre del procedimiento no debe tener visibilidad **privada**.

La ocurrencia de definición de un procedimiento componente **constr** tiene que ser la misma que la de su modo **moreta** asociado. Un procedimiento componente **constr** no debe especificar una *espec de resultado* y debe ser **no recursivo**.

La ocurrencia de definición de un procedimiento componente **destr** tiene que ser la misma que la de su modo **moreta** asociado. Un procedimiento componente **destr** no debe especificar una *lista de parámetros formales* ni una *espec de resultado* y debe ser **no recursivo**.

La *cadena de nombre simple*, si se especifica, debe ser igual a la cadena de nombre de la *ocurrencia de definición* que precede a la *definición de procedimiento*.

Sólo si se especifica **LOC** en la *espec de parámetro* o en la *espec de resultado* puede el modo en ella tener la **propiedad de no-valor**.

Todos los nombres de excepción mencionados en la *lista de excepciones* deben ser diferentes.

Si P1 y P2 son procedimientos componentes o procesos componentes, entonces P1 concuerda con P2 únicamente si:

a) P1 y P2 son del mismo tipo; y

b) P1 y P2 tienen la misma cadena de nombre simple; y

c) las listas de parámetros formales de P1 y P2 son sintáctica y semánticamente equivalentes; y

d) las specs de resultado de P1 y P2 son sintáctica y semánticamente equivalentes.

Si P es un procedimiento componente o un proceso componente, entonces P_B *corresponde* a P_S únicamente si:

a) P_B concuerda con P_S; y

b) las listas de excepciones de P_S y P_B son sintáctica y semánticamente equivalentes; y

c) las listas de atributos de P_S y P_B son sintáctica y semánticamente equivalentes.

Dos procedimientos P1 y P2 son *conformes* uno con otro únicamente si:

a) ambos tienen el mismo número de parámetros y los nombres de los modos de parámetros correspondientes son conformes uno con otro; y

b) (ambos tienen el modo resultado y los nombres de esos modos resultado conformes unos con otros, o ninguno de los dos tiene modo resultado).

ejemplos:

1.4 *add:*

PROC *(i,j INT)* **RETURNS** *(INT)* **EXCEPTIONS** *(OVERFLOW)*;
RESULT *i+j*;

END *add*; *(1.1)*

put:

PROC(*p RANGE(1:10)*) **PRE**((*p* > *0*) *AND* (*p* < *11*));

...;

END *put*; *(10.1)*

10.5 Especificaciones y definiciones de proceso

sintaxis:

<sentencia de definición de proceso> ::= *(1)*
<ocurrencia de definición> : *<definición de proceso>*
[*<manejador>*] [*<cadena de nombre simple>*] ; *(1.1)*
| *<ejemplificación de proceso genérico>* ; *(1.2)*

<definición de proceso> ::= *(2)*
PROCESS ([*<lista de parámetros formales>*])
<cuerpo de proceso> **END** *(2.1)*

semántica: Una sentencia de definición de proceso define una secuencia, posiblemente parametrizada, de acciones que pueden ponerse en marcha para su ejecución concurrente desde diferentes lugares del programa (véase la cláusula 11).

propiedades estáticas: Una *ocurrencia de definición* en una *sentencia de definición de proceso* define un nombre de **proceso**.

Un nombre de **proceso** tiene la siguiente propiedad asociada, definida por su *definición de proceso*:

- Tiene una lista de **especs de parámetro** que son definidos por las ocurrencias de *espec de parámetro* en la *lista de parámetros formales*, consistiendo cada parámetro en un modo y posiblemente un atributo de parámetro.

condiciones estáticas: Una *cadena de nombre simple*, si se especifica, debe ser igual a la cadena de nombre de la *ocurrencia de definición* que precede la *definición de proceso*.

Una *sentencia de definición de proceso* no puede estar rodeada por una región o por un bloque distinto de la definición de proceso imaginario más externo (véase 10.8).

Los atributos de parámetro de la *lista de parámetros formales* no pueden ser **INOUT** ni **OUT**.

Sólo si se especifica **LOC** en la *espec de parámetro* en la *lista de parámetros formales*, puede el modo en ella tener la **propiedad de no-valor**.

ejemplos:

14.13 **PROCESS** *();*

wait:

PROC (*x INT);*

*/*some wait action*/*

END *wait;*

DO FOR EVER;

wait(10 / seconds */)*;

CONTINUE *operator_is_ready*;

OD;

END *(2.1)*

10.6 Módulos

sintaxis:

<módulo> ::= *(1)*
[*<lista de contextos>] [<ocurrencia de definición>* :]
MODULE [BODY] *<cuerpo de módulo>* ***END***
[*<manejador>*] [*<cadena de nombre simple>*]; *(1.1)*
| *<modulión remoto>* *(1.2)*
| *<ejemplificación de módulo genérico>* *(1.3)*

semántica: Un módulo es una sentencia de acción que contiene posiblemente declaraciones y definiciones locales. Un módulo es un medio de restringir la visibilidad de las cadenas de nombre; no influye en el tiempo de vida de las localizaciones declaradas localmente.

Las reglas de visibilidad detalladas de los módulos se dan en 12.2.

propiedades estáticas: Una *ocurrencia de definición* en un *módulo* define un nombre de **módulo** y un nombre de **etiqueta**. El nombre tiene asociado el *módulo* (visto como un modulión, es decir, excluyendo la *lista de contextos* y la *ocurrencia de definición*, si las hubiere).

Un *módulo* se desarrolla por piezas si y sólo si se especifica una *lista de contextos*.

Un *módulo* es un **cuerpo de módulo** si y sólo si se especifica **BODY**.

condiciones estáticas: La *cadena de nombre simple*, si se especifica, debe ser igual a la cadena de nombre de la *ocurrencia de definición*.

Un *modulión remoto* en un *módulo* debe referirse a un *módulo*.

ejemplos:

7.48
```
MODULE
    SEIZE convert;
    DCL n INT INIT:= 1979;
    DCL rn CHARS (20) INIT:= (20)" ";
    GRANT n,rn;
    convert();
    ASSERT rn = "MDCCCCLXXVIIII"//(6)" ";
END                                                    (1.1)
```

10.7 Regiones

sintaxis:

```
<región> ::=                                           (1)
        [ <lista de contextos> ] [ <ocurrencia de definición> :]
        REGIÓN [ BODY ] <cuerpo de región> END
        [ <manejador> ] [ <cadena de nombre simple> ];  (1.1)
|       <modulión remoto>                               (1.2)
|       <instanciación de región genérica>              (1.3)
```

semántica: Una región es un medio de proporcionar acceso mutuamente exclusivo a sus objetos de datos declarados localmente para la ejecución concurrente de procesos (véase la claúsula 11). Determina la visibilidad de los nombres creados localmente en la misma forma que un módulo.

propiedades estáticas: Una *ocurrencia de definición* en una *región* define un nombre de **región**. Tiene asociada la región (vista como un modulión es decir, excluyendo la *lista de contextos* y la *ocurrencia de definición*, si las hubiere).

Una *región* se desarrolla por piezas si y sólo si se especifica una *lista de contextos*.

Una *región* es un **cuerpo de región** si y sólo si se especifica **BODY**.

condiciones estáticas: La *cadena de nombre simple*, si se especifica, debe ser igual a la cadena de nombre de la *ocurrencia de definición*.

Una *región* no debe estar rodeada por un bloque distinto de la definición de proceso imaginario más externo.

Un *modulión remoto* en una *región* debe referirse a una *región*.

ejemplos: Véanse 13.1-13.28.

10.8 Programa

sintaxis:

```
<programa> ::=                                         (1)
        { <módulo> | <módulo de espec> | <región> | <región de espec>
    |   <sentencia de declaración moreta>
    |   <sentencia de definición de sínmodo moreta>
    |   <sentencia de definición de neomodo moreta>
    |   <plantilla> } +                                 (1.1)
```

semántica: Los programas consisten en una lista de módulos o regiones rodeados por una definición del proceso imaginario más externo.

Se considera que las definiciones de los nombres predefinidos CHILL (véase III.2) y las rutinas incorporadas definidas por la implementación y modos enteros, a efectos de tiempo de vida, están definidos en el dominio de la definición del proceso imaginario más externo. En cuanto a su visibilidad, véase 12.2.

10.9 Asignación de espacio de almacenamiento y tiempo de vida

Se denomina tiempo de vida de una localización o procedimiento al tiempo durante el cual éstos existen dentro de un programa.

Se crea una localización mediante una declaración o ejecutando una llamada a rutina incorporada *GETSTACK* o *ALLOCATE*.

El tiempo de vida de una localización declarada en el dominio de un bloque es el tiempo durante el cual el control reside en dicho bloque, o en un procedimiento cuya llamada procede de ese bloque, a menos que se declare con el atributo **STATIC**. El tiempo de vida de una localización declarada en el dominio de un modulión es el mismo que si se hubiera declarado en el dominio del bloque más próximo que rodea al modulión. El tiempo de vida de una localización declarada con el atributo **STATIC** es el mismo que si se hubiese declarado en el dominio de la definición del proceso imaginario más externo. Esto significa que, para una declaración de localización con el atributo **STATIC**, la asignación de memoria se efectúa una sola vez, concretamente al principio del proceso imaginario más externo. Si tal declaración aparece dentro de una definición de procedimiento o de proceso, solamente existirá una localización para todas las invocaciones o activaciones.

El tiempo de vida de una localización creada ejecutando una llamada a rutina incorporada *GETSTACK* finaliza cuando termina el bloque directamente circundante.

El tiempo de vida de una localización creada por una llamada a rutina incorporada *ALLOCATE* es el tiempo que transcurre entre el instante en que se produce la llamada a *ALLOCATE* y el instante en que la localización deja de poder ser accedida por un programa CHILL cualquiera. Esto último siempre ocurre cuando se aplica una llamada a rutina incorporada *TERMINATE* sobre un valor de referencia **atribuido** que hace referencia a la localización.

El tiempo de vida de un acceso creado en una declaración de identidad-loc es el bloque directamente circundante de la declaración de identidad-loc.

El tiempo de vida de un procedimiento es el bloque directamente circundante de la definición de procedimiento.

propiedades estáticas: Se dice que una *localización* es **estática**, si y sólo si es una *localización modo estático* de uno de los tipos siguientes:

- Un *nombre de localización* declarado con el atributo **STATIC** o cuya definición no está rodeada por un bloque diferente de la definición del proceso imaginario más externo.
- Un *elemento de cadena* o *segmento de cadena* en el que la *localización cadena* es **estática** y, o bien el *elemento de la izquierda* y el *elemento de la derecha*, o el *elemento de arranque* y el *tamaño de segmento*, son **constantes**.
- Un *elemento de matriz* en el que la *localización matriz* es **estática** y la *expresión* es **constante**.
- Un *segmento de matriz* en el que la *localización matriz* es **estática** y, o bien el *elemento inferior* y el *elemento superior*, o el *primer elemento* y el *tamaño de segmento*, son **constantes**.
- Un *campo de estructura* en el que la *localización estructura* es **estática**.
- Una *conversión de localización* en la que la *localización* que aparece en ella es **estática**.

10.10 Construcciones para programación por piezas (o programación separada)

Los módulos y regiones son las unidades elementales (piezas) en que puede subdividirse un programa CHILL completo que se desarrolla por piezas. El texto de esas piezas se indica por construcciones remotas (véase 10.10.1). CHILL define la sintaxis y la semántica de programas completos, en los que todas las ocurrencias de piezas remotas se han sustituido virtualmente por el texto referenciado.

10.10.1 Piezas remotas

sintaxis:

<modulión remoto> ::= *(1)*
[*<cadena de nombre simple>* :] **REMOTE** *<designador de pieza>* ; *(1.1)*

<espec remota> ::= *(2)*
[*<cadena de nombre simple>* :] **SPEC REMOTE** *<designador de pieza>* ; *(2.1)*

<contexto remoto> ::= *(3)*
CONTEXT REMOTE *<designador de pieza>*
[*<cuerpo de contexto>*] **FOR** *(3.1)*

<módulo de contexto> ::= *(4)*
CONTEXT MODULE REMOTE *<designador de pieza>* ; *(4.1)*

<designador de pieza> ::= *(5)*
<literal de cadena de caracteres> *(5.1)*
| *<nombre de referencia de texto>* *(5.2)*
| *<vacío>* *(5.3)*

<unidad de programa remoto> ::= *(6)*
[*<cadena de nombre simple>* :] **REMOTE** *<designador de pieza>* ; *(6.1)*

sintaxis derivada: La notación:

CONTEXT MODULE REMOTE *<designador de pieza>*

es sintaxis derivada para:

CONTEXT REMOTE *<designador de pieza>* **FOR**
MODULE SEIZE ALL; **END**;

NOTA – Esta construcción es redundante pero puede utilizarse para asegurar la comprobación de consistencia.

semántica: Los *moduliones remotos*, *especs remotas*, *contextos remotos*, *módulos de contexto* y *unidades de programa remoto* son medios para representar el texto fuente de un programa como un conjunto de ficheros (interconectados).

Un *designador de pieza* hace referencia, en una forma definida por la implementación, a una descripción de una pieza de texto fuente CHILL, como sigue:

- Si el *designador de pieza* está vacío, el texto fuente se extrae de un lugar determinado por la estructura del programa.
- Si el *designador de pieza* contiene un *literal de cadena de caracteres*, el *literal de cadena de caracteres* se utiliza para extraer el texto fuente.
- Si el *designador de pieza* contiene un *nombre de referencia de texto*, el *nombre de referencia de texto* se interpreta, de una manera definida por la implementación, para obtener el texto fuente.

Un programa con: 1. *moduliones remotos*, 2. *especs remotas*, 3. *unidades de programa remoto* es equivalente al programa elaborado sustituyendo cada uno de los 1. *moduliones remotos*, 2. *espec remotas*, 3. *unidad de programa remoto* por la pieza de texto CHILL a la que hace referencia su *designador de pieza*.

Un programa con *contextos remotos* es equivalente al programa elaborado sustituyendo cada *contexto remoto* por la pieza de texto CHILL a la que hace referencia su *designador de pieza* en que el *cuerpo de contexto* se ha insertado virtualmente inmediatamente después de la última ocurrencia de *cuerpo de contexto* en la *lista de contextos* a que hace referencia el *designador de pieza*.

Si la pieza designada no está disponible como texto CHILL, se considera que el *designador de pieza* en la misma se refiere a una pieza equivalente de texto CHILL que se introduce virtualmente.

Aunque la semántica de una pieza remota se define en términos de un reemplazo, CHILL no implica ninguna sustitución textual.

condiciones estáticas: El *designador de pieza* en un 1. *modulión remoto*, 2. *espec remota*, 3. *contexto remoto*, 4. *módulo de contexto*, 5. *unidad de programa remoto* debe referir a una descripción de una pieza de texto fuente que es una producción terminal de un 1. *modulión* o *región* que no es un *modulión remoto*, 2. *módulo de espec* o *región* que no es una *espec remota*, 3., 4. *lista de contextos* que no es un *contexto remoto,* 5. *unidad de programa* que no es remoto.

Cuando el texto fuente al que hace referencia el *designador de pieza* en un *modulión remoto* arranca con una *ocurrencia de definición*, el *modulión remoto* debe arrancar con una *cadena de nombre simple* que es cadena de nombre de esa *ocurrencia de definición*.

Cuando el texto fuente al que hace referencia el *designador de pieza* en una *espec remota* arranca con una cadena de *nombre simple*, la *espec remota* debe arrancar con la misma *cadena de nombre simple*.

Cuando el texto fuente a que hace referencia el *designador de pieza* en una *unidad de programa remoto* comienza por una *cadena de nombre simple*, la primera ocurrencia de definición en la *unidad de programa remoto* debe ser la misma *cadena de nombre simple*.

ejemplos:

25.9 *stack:* **REMOTE** *"example 27 or 28"* ; *(1.1)*

25.9 *"example 27 or 28"* *(5.1)*

10.10.2 Módulos de espec, regiones de espec y contextos

sintaxis:

<módulo de espec> ::= *(1)*
<módulo de espec simple> *(1.1)*
| *<espec de módulo>* *(1.2)*
| *<espec remota>* *(1.3)*

<módulo de espec simple> ::= *(2)*
[*<lista de contextos>*] [*<cadena de nombre simple>* :] **SPEC MODULE**
<cuerpo de módulo de espec> ***END*** [*<cadena de nombre simple>*] ; *(2.1)*

<espec de módulo> ::= *(3)*
[*<lista de contextos>*] *<cadena de nombre simple>* : **MODULE SPEC**
<cuerpo de módulo de espec> **END** [*<cadena de nombre simple>*] ; *(3.1)*

<región de espec> ::= *(4)*
<región de espec simple> *(4.1)*
| *<espec de región>* *(4.2)*
| *<espec remota>* *(4.3)*

<región de espec simple> ::= *(5)*
[*<lista de contextos>*] [*<cadena de nombre simple>* :] **SPEC REGION**
<cuerpo de región de espec> **END** [*<cadena de nombre simple>*] ; *(5.1)*

<espec de región> ::= *(6)*
[*<lista de contextos>*] *<cadena de nombre simple>* : **REGION SPEC**
<cuerpo de región de espec> **END** [*<cadena de nombre simple>*] ; *(6.1)*

<lista de contextos> ::= *(7)*
<contexto> { *<contexto>* }* *(7.1)*
| *<contexto remoto>* *(7.2)*

<contexto> ::= *(8)*
CONTEXT *<cuerpo de contexto>* **FOR** *(8.1)*

semántica: Se utilizan *módulos de espec simples*, *regiones de espec simples* y *contextos* para especificar las propiedades estáticas de nombres. Son redundantes, pero pueden utilizarse para la programación por piezas (denominada también programación separada).

Las *cadenas de nombre simple* en *módulos de espec* y *regiones de espec* no son nombres, no están **ligadas**, y no tienen reglas de visibilidad.

Los 1. *módulos de espec*, 2. *regiones de espec* en un dominio **real** indican las propiedades de uno o más 1. *módulos*, 2. *regiones* que están *compiladas* por piezas y se consideran encerradas en ese dominio. Los textos de tales 1. *módulos*, 2. *regiones* se indican por ocurrencias de *moduliones remotos*. Una *lista de contextos* indica los dominios circundantes (obsérvese que un modulión que está desarrollado por piezas tiene siempre delante una *lista de contextos*).

Para cada *cadena de nombre OP ! NS* **visible** en el dominio de una 1. *espec de módulo*, 2. *espec de región* y **enlazado** a una ocurrencia de **cuasi** definición y que está otorgada dentro de un dominio **real** como *NP ! NS*, se considera que se introduce una sentencia de otorgamiento (virtual) con la misma *cadena de nombre OP ! NS* **antigua** y la **nueva** *cadena de nombre NP ! NS* en el dominio del correspondiente 1. **cuerpo de módulo**, 2. **cuerpo de región**.

condiciones estáticas: En un *módulo de espec* o una *región de espec*, la *cadena de nombre simple* facultativo que sigue a **END** sólo puede estar presente si la *cadena de nombre simple* facultativa antes de **SPEC** está presente. Cuando ambos están presentes, deben tener cadenas de nombres iguales.

Un *contexto* que no tiene un grupo directamente circundado puede no contener sentencias de visibilidad.

Un dominio **real** que contiene un 1. *módulo de espec*, 2. *región de espec* tiene que contener también por lo menos un *modulión remoto*, y viceversa.

Si un **dominio real r** contiene un 1. *módulo* que es un **cuerpo de módulo**, 2. *región* que es un **cuerpo de región**, debe entonces contener también 1. *espec de módulo*, 2. *espec de región*, tal que la *cadena de nombre simple* que los precede tenga iguales cadenas de nombre. Se dice que la 1. *espec de módulo*, 2. *espec de región* tiene un 1. **cuerpo de módulo**, 2. **cuerpo de región**, **correspondiente**.

Una *espec remota* en un 1. *módulo de espec*, 2. *región de espec* tiene que referir a un 1. *módulo de espec*, 2. *región de espec*.

Un *módulo de espec* o una *región de espec* no pueden estar circundados por un bloque diferente de la definición del proceso imaginario más externo.

ejemplos:

23.2 *letter_count:*

SPEC MODULE

SEIZE *max;*

count: **PROC** *(input* **ROW CHARS** *(max)* **IN**,

output **ARRAY** *('A':'Z') INT* **OUT***)* **END**;

GRANT *count;*

END *letter_count;* *(1.1)*

10.10.3 Cuasi sentencias

sintaxis:

<cuasi sentencia de datos> ::= *(1)*
<cuasi sentencia de declaración> *(1.1)*
| *<cuasi sentencia de definición>* *(1.2)*

<cuasi sentencia de declaración> ::= *(2)*
DCL *<cuasi declaración>* { , *<cuasi declaración>* }* ; *(2.1)*

<cuasi declaración> ::= *(3)*
<cuasi declaración de localización> *(3.1)*
| *<cuasi declaración de identidad-loc>* *(3.2)*

<cuasi declaración de localización> ::= *(4)*
<lista de ocurrencias de definición> <modo> *(4.1)*

<cuasi declaración de identidad-loc> ::= *(5)*
<lista de ocurrencias de definición> <modo>
LOC [**NONREF**] [**DYNAMIC**] *(5.1)*

<cuasi sentencia de definición> ::= *(6)*
<sentencia de definición de sínmodo> *(6.1)*
| *<sentencia de definición de neomodo>* *(6.2)*
| *<sentencia de definición de sinónimo>* *(6.3)*
| *<cuasi sentencia de definición de sinónimo>* *(6.4)*
| *<cuasi sentencia de definición de procedimiento>* *(6.5)*
| *<cuasi sentencia de definición de proceso>* *(6.6)*
| *<cuasi sentencia de definición de señal>* *(6.7)*
| *<sentencia de definición de señal>* *(6.8)*
| *<vacía>*; *(6.9)*

<cuasi sentencia de definición de sinónimo> ::= *(7)*
SYN *<cuasi definición de sinónimo>* { , *<cuasi definición de sinónimo>* }* ; *(7.1)*

<cuasi definición de sinónimo> ::= *(8)*
<lista de ocurrencias de definición> { *<modo>* = [*<valor constante>*] |
[*<modo>*] = *<expresión literal>* } *(8.1)*

<cuasi sentencia de definición de procedimiento> ::= *(9)*
<ocurrencia de definición> : **PROC** ([*<cuasi lista de parámetros formales>*])
[*<espec de resultado>*] [**EXCEPTIONS** (*<lista de excepciones>*)]
<lista de atributos de procedimiento> [**END** [*<cadena de nombre simple>*]] ; *(9.1)*

<cuasi lista de parámetros formales> ::= *(10)*
<cuasi parámetro formal> { , *<cuasi parámetro formal>* }* *(10.1)*

<cuasi parámetro formal> ::= *(11)*
<cadena de nombre simple> { , *<cadena de nombre simple>* }*
<espec de parámetro> *(11.1)*

<cuasi sentencia de definición de proceso> ::= *(12)*
<ocurrencia de definición> : **PROCESS** ([*<cuasi lista de parámetros formales>*])
[**END** [*<cadena de nombre simple>*]] ; *(12.1)*

<cuasi sentencia de definición de señal> ::= *(13)*
SIGNAL *<cuasi definición de señal>* { , *<cuasi definición de señal>* }* ; *(13.1)*

<cuasi definición de señal> ::= *(14)*
<ocurrencia de definición> [= (*<modo>* { , *<modo>* }*)] [**TO**] *(14.1)*

semántica: Se utilizan cuasi sentencias en *módulos de espec*, *regiones de espec* y *contextos* para especificar propiedades estáticas de nombres. *Módulos de espec*, *regiones de espec* y *contextos* pueden contener cuasi sentencias y sentencias reales. Estas especificaciones son redundantes, pero pueden utilizarse cuasi sentencias para la programación por piezas.

Una implementación que no pueda garantizar la igualdad de los valores entre nombres de **cuasi sinónimo constante** y los **reales** correspondientes puede desautorizar la indicación del *valor constante*.

Obsérvese que en CHILL no existen **cuasi** *ocurrencias de definición* para nombres de **etiqueta**.

propiedades estáticas: Las cuasi sentencias son formas restringidas de las *sentencias* correspondientes, y están sujetas a sus propiedades estáticas.

El nombre definido por una *ocurrencia de definición* en una *cuasi declaración de identidad-loc* es **referenciable** si no está especificado **NONREF**.

condiciones estáticas: Las cuasi sentencias son formas restringidas de las sentencias correspondientes y cumplen sus mismas condiciones estáticas.

Una *cuasi sentencia de definición de sinónimo* o una *cuasi sentencia de definición de señal* sólo puede estar encerrada directamente en un *módulo de espec simple*, una *región de espec simple* o un *contexto*. Una *sentencia de definición de sinónimo* o una *sentencia de definición de señal* en una *cuasi sentencia de definición* sólo puede estar encerrada directamente en una *espec de módulo* o una *espec de región*.

10.10.4 Concordancia entre cuasi ocurrencias de definición y ocurrencias de definición

Se dice que dos *ocurrencias de definición* **concuerdan** si tienen categorías semánticas idénticas, y:

- si son nombres de **sinónimos**, entonces deben tener la misma **regionalidad** y el mismo valor, el modo **raíz** de sus clases debe ser **igual**, deben tener un M-valor, M-derivado, M-referencia, de clase **nula** o **general**, y si el que es cuasi es **literal**, también debe serlo el otro;
- si son nombres de **neomodo** o nombres de **sínmodo**, sus modos deben ser **iguales**;
- si son nombres de **localización** o nombres de **identidad-loc**, deben tener la misma **regionalidad**, ser ambos o ninguno **referenciables**, ser ambos o ninguno **estáticos**, y sus modos deben ser **iguales**;
- si son nombres de **procedimiento**, deben tener la misma **regionalidad** y **generalidad**, ser ambos o ninguno **críticos** y satisfacer las mismas condiciones de igualdad que los modos procedimiento, y las *cadenas de nombre simple* correspondientes (en posición) en la *lista de parámetros formales* y la *cuasi lista de parámetros formales* deben ser las mismas;
- si son nombres de **proceso**, los parámetros de sus definiciones de proceso deben satisfacer las mismas condiciones de concordancia e igualdad que los parámetros de los nombres de **procedimiento**;
- si son nombres de **señal**, deben ambos o ninguno especificar **TO**, sus listas de modos deben tener el mismo número de modos, y los modos correspondientes deben ser **iguales**.

Si los dos modos de estructura están **ligados por novedad** en un dominio R, deberán tener el mismo conjunto de nombres de campo **visibles** en R.

Se aplican las reglas siguientes:

- Si una *cadena de nombre* en un dominio que no es el dominio de un *módulo de espec*, *región de espec* o *contexto*, está **ligada** a una **cuasi** *ocurrencia de definición*, debe estar también **ligada** a una *ocurrencia de definición* que no es una **cuasi** *ocurrencia de definición*, y además:
  - Sea una *cadena de nombre* **ligada** a una **cuasi** *ocurrencia de definición* QD y **ligada** también a una *ocurrencia de definición* **real** RD en el dominio R, entonces:
    1) QD y RD deben **concordar** como se ha definido más arriba, y
    2) RD y QD deben ambos estar encerrados en un grupo encerrado de R, o no estar ninguno encerrado en el grupo de R o, si R es el dominio de un *módulo* o una *región* que es un **cuerpo de módulo** o **cuerpo de región**, entonces QD debe estar encerrada en el grupo de *espec de módulo* o la *espec de región* **correspondiente** y RD debe estar encerrada en el grupo de R.

- Si una *cadena de nombre* en un dominio **real** R está **ligada** a una **cuasi** *ocurrencia de definición* que está encerrada en el grupo de R (es decir, rodeada por un modulión de spec), debe estar también **ligada** a una *ocurrencia de definición* **real** que esté rodeada por el grupo de un *módulo* o una *región* indicados por un *modulión remoto* encerrado directamente en R (de modo informal, si la interfaz otorga, también debe hacerlo la implementación). Si la **cuasi** *ocurrencia de definición* está encerrada en el grupo de una *spec de módulo* o de una *spec de región*, la *ocurrencia de definición* **real** debe estar encerrada en el grupo del modulión **correspondiente**.
- Para cada *cadena de nombre* en el dominio Q de un *módulo de spec* o *región de spec* directamente encerrada en un dominio **real** R que está **ligada** a una *ocurrencia de definición* no rodeada por Q, debe haber una *cadena de nombre* idéntica en el dominio de un *módulo* o una *región* indicado por un *modulión remoto* directamente encerrado en R que está **ligado** por la misma *ocurrencia de definición* (de modo informal, si el interfaz hace la toma, también debe hacerlo la implementación).

- Si dos *cadenas de nombres* están **ligadas** a la misma 1. *ocurrencia de definición* **real**, 2. **cuasi** *ocurrencia de definición* en un dominio, ambas *cadenas de nombre* deben estar **ligadas** a la misma 1. **cuasi** *ocurrencia de definición*, 2. *ocurrencia de definición* **real**, o ambos dejarán de estar **ligados**.
- Una **novedad real** puede no ser una **novedad ligada** a dos **cuasi novedades** en un dominio cualquiera.

  Sean QN una **cuasi novedad** y RN una **novedad real** que están **ligadas por novedad** entre si en un dominio R; entonces, RN y QN estarán ambas encerradas en un grupo encerrado de R, o ninguna estará encerrada en el grupo de R, o si R es el dominio de un *módulo* o una *región* que es un **cuerpo de módulo** o **cuerpo de región**, entonces RN debe estar encerrada en el grupo de R y QN debe estar encerrada en el grupo de la *spec de módulo* o *spec de región* **correspondiente**.

10.11 Genericidad

Muchos algoritmos resuelven problemas sobre artículos de datos estructurados similarmente cuyos modos componentes son diferentes. La genericidad proporciona un medio para implementar esos algoritmos como esquemas de programas que son ejemplificados sustituyendo definiciones de modos formales por definiciones de modos efectivos.

sintaxis:

<plantilla> ::= *(1)*
<plantilla de módulo genérico> *(1.1)*
| *< plantilla de región genérica>* *(1.2)*
| *< plantilla de procedimiento genérico>* *(1.3)*
| *< plantilla de proceso genérico>* *(1.4)*
| *< plantilla de modo de módulo genérico>* *(1.5)*
| *< plantilla de modo de región genérico>* *(1.6)*
| *< plantilla de modo de tarea genérico>* *(1.7)*
| *< plantilla de modo de interfaz genérico>* *(1.8)*
| *<unidad de programa remoto>* *(1.9)*

<plantilla de módulo genérico> ::= *(2)*
[*<lista de contextos>*] [*<ocurrencia de definición>* :]
<parte genérica> **MODULE** [**BODY**] *<cuerpo de módulo>* **END**
[*<manejador>*] [*<cadena de nombre simple>*] ; *(2.1)*

<plantilla de región genérica> ::= *(3)*
[*<lista de contextos>*] [*<ocurrencia de definición>* :]
<parte genérica> **REGION** [**BODY**] *<cuerpo de región>* **END**
[*<manejador>*] [*<cadena de nombre simple>*] ; *(3.1)*

<plantilla de procedimiento genérico> ::= *(4)*
<ocurrencia de definición> : *<parte genérica> <definición de procedimiento>*
[*<manejador>*] [*<cadena de nombre simple>*] ; *(4.1)*

<plantilla de proceso genérico> ::= *(5)*
<ocurrencia de definición> : *<parte genérica> <definición de proceso>*
[*<manejador>*] [*<cadena de nombre simple>*] ; *(5.1)*

<plantilla de modo módulo genérico> ::= *(6)*
<parte genérica> <especificación de modo módulo> *(6.1)*

<plantilla de modo región genérico> ::= *(7)*
<parte genérica> <especificación de modo región> *(7.1)*

<plantilla de modo tarea genérico> ::= *(8)*
<parte genérica> <especificación de modo tarea> *(8.1)*

<plantilla de modo interfaz genérico> ::= *(9)*
<parte genérica> <modo interfaz> *(9.1)*

<parte genérica> ::= *(10)*
GENERIC { *<sentencia de toma>* }***** *<lista de parámetros formales genéricos>* *(10.1)*

<lista de parámetros formales genéricos> ::= *(11)*
{ *<parámetro formal genérico>* }***** *(11.1)*

<parámetro formal genérico> ::= *(12)*
SYN *<lista de sinónimos formales genéricos>* ; *(12.1)*
| **MODE** *<lista de modos formales genéricos>* ; *(12.2)*
| **PROC** *<espec de procedimiento formal genérico>* ; *(12.3)*

<lista de sinónimos formales genéricos> ::= *(13)*
<sinónimo formal genérico> { ,*<sinónimo formal genérico>* }***** *(13.1)*

<lista de modos formales genéricos> ::= *(14)*
<modo formal genérico> { ,*<modo formal genérico>* }***** *(14.1)*

<sinónimo formal genérico > ::= *(15)*
<lista de ocurrencias de definición> =
{*<modo>* | **ANY_DISCRETE** | **ANY_INT** | **ANY_REAL** } *(15.1)*

<modo formal genérico> ::= *(16)*
<lista de ocurrencias de definición> = *<indicación de modo formal genérico>* *(16.1)*

<indicación de modo formal genérico> ::= *(17)*
ANY *(17.1)*
| **ANY_ASSIGN** *(17.2)*
| **ANY_DISCRETE** *(17.3)*
| **ANY_INT** *(17.4)*
| **ANY_REAL** *(17.5)*
| *<nombre de modo moreta>* *(17.6)*

<espec de procedimiento formal genérico> ::= *(18)*
<cadena de nombre simple> ([*<lista de parámetros formales>*])
[*<espec de resultado>*] [**EXCEPTIONS** (*<lista de excepciones>*)] *(18.1)*

<ejemplificación de módulo genérico> ::= *(19)*
<cadena de nombre simple> : **MODULE** = **NEW** <nombre de módulo genérico>
{ *<sentencia de toma>* }*****
<lista de parámetros efectivos genéricos> **END** [*<cadena de nombre simple>*] ; *(19.1)*

< ejemplificación de región genérica> ::= *(20)*
<cadena de nombre simple > : **REGION = NEW** *<nombre de región genérica>*
{ *<sentencia de toma>* }*****
<lista de parámetros efectivos genéricos> **END** [*<cadena de nombre simple>*] ; *(20.1)*

< ejemplificación de procedimiento genérico> ::= *(21)*
<cadena de nombre simple> : **PROC = NEW** *<nombre de procedimiento genérico>*
{ *<sentencia de toma>* }*****
<lista de parámetros efectivos genéricos > **END** [*<cadena de nombre simple>*] ; *(21.1)*

< ejemplificación de proceso genérico> ::= *(22)*
<cadena de nombre simple> : **PROCESS** = **NEW** *<nombre de proceso genérico>*
{ *<sentencia de toma>* }*****
<lista de parámetros efectivos genéricos> **END** [*<cadena de nombre simple>*] ; *(22.1)*

< ejemplificación de modo moreta genérico> ::= *(23)*
NEW *<nombre de modo moreta genérico>*
{ *<sentencia de toma>* }*****
<lista de parámetros efectivos genéricos> **END** [*<cadena de nombre simple>*] ; *(23.1)*

<lista de parámetros efectivos genéricos> ::= *(24)*
<parámetro efectivo genérico> { *<parámetro efectivo genérico>* }***** *(24.1)*

<parámetro efectivo genérico> ::= *(25)*
<sentencia de definición de sinónimo> *(25.1)*
| *<sentencia de definición de sínmodo>* *(25.2)*
| *<sentencia de definición de neomodo>* *(25.3)*
| *<procedimiento efectivo genérico>* *(25.4)*

<procedimiento efectivo genérico> ::= *(26)*
PROC *<lista de ocurrencias de definición>* = *<nombre de procedimiento>* ; *(26.1)*

semántica: La palabra *unidad* significa un módulo, una región, un procedimiento, un proceso, o un modo moreta.

Una unidad genérica es una unidad que contiene una parte genérica.

Una unidad genérica es una plantilla de la que pueden obtenerse unidades no genéricas mediante un proceso denominado ejemplificación genérica.

Una unidad genérica puede contener parámetros formales genéricos. Durante la ejemplificación genérica, se genera una copia de la unidad genérica y los parámetros formales genéricos se reemplazan por los parámetros efectivos genéricos en la totalidad de la unidad. Después de este reemplazo, la parte genérica seleccionada se borra, con lo que se obtiene una unidad no genérica.

propiedades estáticas: Los sinónimos formales genéricos se caracterizan por dos tipos de propiedades:

a) las propiedades que un parámetro formal genérico tiene en la unidad genérica;

b) las propiedades que un correspondiente parámetro efectivo genérico debe tener para ser aceptado:

modo:	prop formal:	propiedades del modo dado que no deben tener la propiedad **no-valor**.
	prop efectiva:	el valor de parámetro efectivo genérico debe ser un valor del modo.
ANY_DISCRETE:	prop formal:	operaciones disponibles: :=, relacional, PRED, SUCC, NUM, SIZE.
	prop efectiva:	el valor del parámetro efectivo genérico debe ser un valor de un modo discreto.
ANY_INT:	prop formal:	ANY_DISCRETE y +, –, *, /, mod, abs, rem.
	prop efectiva:	el valor del parámetro efectivo genérico debe ser un valor de un modo entero.
ANY_REAL:	prop formal:	operaciones disponibles: ANY_ASSIGN y relacional, +, –, *, /.
	prop efectiva:	el valor del parámetro efectivo genérico debe ser un valor de un modo real.

Los modos formales genéricos se caracterizan por dos tipos de propiedades:

a) las propiedades que un parámetro formal genérico tiene en la unidad genérica;

b) las propiedades que un correspondiente parámetro efectivo genérico debe tener para ser aceptado:

ANY:	prop formal:	SIZE; no puede utilizarse como el modo de una localización o de un parámetro; (puede utilizarse como un modo referenciado).
	prop efectiva:	cualquier modo aceptable.
ANY_ASSIGN:	prop formal:	operaciones disponibles: :=, comparación, SIZE.
	prop efectiva:	el modo debe poseer propiedades formales.
ANY_DISCRETE:	prop formal:	operaciones disponibles: :=, relacional, PRED, SUCC, NUM, SIZE.
	prop efectiva:	el modo debe poseer propiedades formales.
ANY_INT:	prop formal:	ANY_DISCRETE y +, –, *, /, mod, abs, rem.
	prop efectiva:	el modo debe poseer propiedades formales.
ANY_REAL:	prop formal:	operaciones disponibles: ANY_ASSIGN y relacional, +, –, *, /.
	prop efectiva:	el modo debe poseer propiedades formales.
nombre de modo moreta:	prop formal:	las del modo.
	prop efectiva:	el mismo modo o cualquier sucesor.

Los procedimientos formales genéricos se caracterizan por dos tipos de propiedades:

a) las propiedades que un parámetro formal genérico tiene en la unidad genérica;

b) las propiedades que un correspondiente parámetro efectivo genérico debe tener para ser aceptado:

prop formal: de acuerdo con la espec de procedimiento formal genérico dado.

prop efectiva: la espec de procedimiento formal genérico dada debe ser **compatible** con la clase de parámetro efectivo genérico.

condiciones estáticas: Para derivaciones en que intervienen plantillas de modo moreta genérico se aplican las siguientes restricciones: si la base es una plantilla, toda entidad derivada debe ser una plantilla. Si la base no es una plantilla, una entidad derivada puede ser una plantilla.

En una ejemplificación genérica debe haber exactamente un parámetro efectivo genérico para cada parámetro formal genérico de la unidad que se ejemplifica.

Las restricciones sobre anidamiento se dan en el cuadro siguiente. Es aplicable a grupos ordinarios, grupos genéricos y ejemplificaciones genéricas.

grupo interior / grupo exterieur	MÓDULO	REGIÓN	PROCED.	PROCESO	Modo Móduli	Modo Región	Modo Tarea	Modo Interfaz
Principio/Fin	Sí	No	Sí	No	Sí	No	No	Sí
PROCED.	Sí	No	Sí	No	Sí	No	No	Sí
PROCESO	Sí	No	Sí	No	Sí	No	No	Sí
MÓDULO	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
REGIÓN	Sí	No	Sí	No	Sí	No	No	Sí
Modo Módulo	No	No	Sí	Sí	No	No	No	No
Modo Región	No	No	Sí	No	No	No	No	No
Modo Tarea	No	No	Sí	No	No	No	No	No
Modo Interfaz	No	No	No	No	No	No	No	No
Programa	Sí	Sí	Sí	No	Sí	Sí	Sí	Sí

El cuadro se basa en la siguiente correspondencia entre plantillas y entidades de CHILL. Para una plantilla en la columna izquierda se aplican las restricciones de la correspondiente entidad en la columna de la derecha:

plantilla de módulo genérico	sentencia de definición de procedimiento
plantilla de región genérica	región
plantilla de procedimiento genérico	sentencia de definición de procedimiento
plantilla de proceso genérico	sentencia de definición de proceso
plantilla de modo módulo genérico	sentencia de definición de procedimiento
plantilla de modo región genérico	región
plantilla de modo tarea genérico	sentencia de definición de proceso
plantilla de mod interfaz genérico	sentencia de definición de procedimiento

11 Ejecución concurrente

11.1 Procesos, tareas, hilos y sus definiciones

Un hilo es un proceso o una tarea. Un proceso es la ejecución secuencial de una serie de sentencias. Dicha ejecución secuencial puede ser concurrente con otros hilos. El comportamiento de un proceso se describe mediante una definición de proceso (véase 10.5), que describe los objetos locales de un proceso y la serie de sentencias de acción que deben ejecutarse secuencialmente.

Un proceso se crea mediante la evaluación de una expresión arrancar (véase 5.2.15). Se vuelve activo (es decir en ejecución) y se considera que se ejecuta concurrentemente con otros hilos. El proceso creado es una activación de la definición indicada por el nombre de **proceso** en la definición de proceso. Pueden crearse y ejecutarse concurrentemente un número no especificado de procesos con la misma definición. Cada proceso se identifica unívocamente por un valor

de ejemplar, obtenido como resultado de la expresión arrancar o la evaluación del operador **THIS**. La creación de un proceso produce la creación de sus localizaciones declaradas localmente, excepto las declaradas con el atributo **STATIC** (véase 10.9), y de los valores y procedimientos definidos localmente. Se dice que las localizaciones, valores y procedimientos declarados localmente tienen la misma activación que el proceso creado al que pertenecen. El proceso imaginario más externo (véase 10.8) que constituye la totalidad del programa CHILL en ejecución, se considera creado por una expresión arrancar ejecutada por el sistema bajo cuyo control se está ejecutando el programa. En la creación de un proceso, sus parámetros formales, si están presentes, denotan los valores y localizaciones según son entregados por los parámetros efectivos correspondientes de la expresión arrancar.

Un proceso se termina mediante la ejecución de una acción parar, llegando al final del cuerpo de proceso, o terminando un manejador especificado al final de la definición del proceso (traspaso de bornes). Si el proceso imaginario más externo ejecuta una acción parar o traspasa los bornes se completará la terminación cuando y sólo cuando finalicen todos los demás hilos del programa.

Una tarea es una ejecución secuencial de una serie de sentencias. Se puede ejecutar concurrentemente con otros hilos. El comportamiento de una tarea se describe por una definición de modo tarea.

Una tarea se crea como parte de la creación e inicialización de una localización de modo tarea (véase 4.1). Es llamada para que pertenezca a esta localización de modo tarea. Una tarea termina si se destruye su localización de modo tarea (véase 10.2). En el nivel de programación CHILL, un hilo siempre está en uno de dos estados: o activo (es decir, en ejecución) o demorado (véase 11.3). La transición de activo a demorado se denomina demorar el hilo, la transición de demorado a activo, se denomina reactivar el hilo.

11.2 Exclusión mutua y regiones

11.2.1 Generalidades

Las regiones (véase 10.7) y las localizaciones de regiones (véase 3.15) constituyen un medio de proporcionar hilos con accesos indirectos mutuamente exclusivos a las localizaciones declaradas en las regiones o localizaciones de regiones por procedimientos otorgados (o concedidos). Las condiciones estáticas de contexto (véase 11.2.2) son tales que los accesos por un hilo (que no es el hilo imaginario más externo) a las localizaciones declaradas en una región solamente pueden realizarse mediante llamadas a procedimientos definidos dentro de la región o modo región y otorgados por éstas.

NOTA – La única situación en la que las localizaciones declaradas dentro de una región o localización de región pueden ser accedidas directamente por un hilo T se da cuando se entra en la región o localización de región y sus inicializaciones ligadas al dominio (si existen) son efectuadas por T. Se dice que un nombre de **procedimiento** denota un **procedimiento crítico** (y es un nombre de **procedimiento crítico**) si está definido dentro de una región y otorgado por ésta.

Se dice que un nombre de **procedimiento componente** denota un procedimiento componente **crítico** (y es un nombre de **procedimiento componente crítico**) si está definido dentro de un modo región y otorgado por el modo región. Se dice que una región es libre si y sólo si el control no reside en ninguno de sus procedimientos **críticos** ni en la propia región que efectúa inicializaciones ligadas al dominio.

Se dice que una localización de región está libre si y sólo si el control no reside en ninguno de sus procedimientos componentes **críticos** ni en la propia localización de región que efectúa las inicializaciones ligadas al dominio. La región estará bloqueada (para evitar una ejecución concurrente) si:

- Se entra en la región (obsérvese que, debido a que las regiones no están rodeadas por un bloque, no pueden hacerse tentativas concurrentes para entrar en la región).
- Se llama a un procedimiento **crítico** de la región.
- Se reactiva un proceso demorado en la región.

La localización de región será bloqueada (para impedir una ejecución concurrente) si:

- Se ha entrado en la localización de región.
- Se llama a un procedimiento componente crítico de la localización de región.
- Un hilo, que está demorado en la localización de región, se reactiva.

La región será liberada y volverá a estar libre si:

- Se abandona la región después de haberse efectuado las inicializaciones ligadas al dominio.
- Un procedimiento **crítico** retorna.
- Un procedimiento **crítico** ejecuta una acción que causa la demora del proceso ejecutante (véase el 11.3). En el caso de llamadas de procedimiento **crítico** dinámicamente anidadas sólo se liberará la última región bloqueada.
- El proceso que ejecuta el procedimiento **crítico** termina. En el caso de llamadas a procedimiento **crítico** dinámicamente anidadas, se liberarán todas las regiones bloqueadas por el proceso.

La localización de región será liberada y volverá a estar libre si:

- Se abandona la localización de región después de haberse efectuado las inicializaciones ligadas al dominio.
- Un procedimiento crítico retorna.
- Un procedimiento crítico ejecuta una acción que causa la demora del hilo ejecutante (véase el 11.3). En el caso de llamadas de procedimiento crítico dinámicamente anidadas sólo se liberará la última región bloqueada.
- Un hilo que ejecuta un procedimiento crítico termina. En el caso de llamadas a procedimiento crítico dinámicamente anidadas, se liberarán todas las localizaciones de región bloqueadas por el hilo. Si, estando bloqueada la región, un hilo intenta llamar a uno de sus procedimientos **críticos**, o se reactiva un hilo demorado en la región, se suspende el hilo hasta que se libere la región (obsérvese que el hilo permanece activo en el sentido CHILL).

Si, estando bloqueada la localización de región, un hilo intenta llamar a uno de sus procedimientos componentes **críticos**, o se reactiva un hilo demorado en la localización de región, se suspende el hilo hasta que la localización de región sea liberada (obsérvese que el hilo permanece activo en el sentido CHILL). Cuando se libera una región y más de un hilo han sido suspendidos mientras intentaban llamar a uno de sus procedimientos **críticos**, o se encuentra demorado en espera de ser reactivado en uno de sus procedimientos **críticos**, solamente se seleccionará un proceso para bloquear la región de acuerdo con un algoritmo de calendarización definido por la implementación.

Cuando se libera una localización de región y más de un hilo han sido suspendidos mientras intentaban llamar a uno de sus procedimientos **críticos** o están demorados en espera de ser reactivados en uno de sus procedimientos componentes **críticos**, un solo hilo será seleccionado para bloquear la localización de región de acuerdo con un algoritmo de calendarización definido por la implementación.

11.2.2 Regionalidad

Para permitir la verificación estática de que solamente puede accederse a una localización declarada en una región llamando a procedimientos **críticos** o entrando en la región para efectuar inicializaciones ligadas al dominio, debe asegurarse el cumplimiento de las siguientes condiciones de contexto estático:

- los requisitos de **regionalidad** mencionados en las secciones apropiadas (acción de asignación, llamada a un procedimiento, acción enviar, acción resultar, etc.);
- los procedimientos **intrarregionales** no son **generales** (véase 10.4);
- los procedimientos **críticos** no son **generales** ni **recursivos** (véase 10.4).

Para permitir la verificación estática de que solamente puede accederse a una localización componente declarada en una localización de región llamando a procedimientos **componentes críticos** o entrando en la región para efectuar inicializaciones ligadas al dominio, debe asegurarse el cumplimiento de las siguientes condiciones estáticas de contexto:

- los requisitos de regionalidad mencionados en las secciones apropiadas (acción de asignación, llamada a un procedimiento, acción enviar, acción resultar, etc.);
- los procedimientos intrarregionales no son generales (véase 10.4);
- los procedimientos críticos no son generales ni recursivos (véase 10.4);
- también los procedimientos componentes no son **en línea** (véase 3.15).

Una *localización* y una *llamada a procedimiento* tienen una **regionalidad** que es **intrarregional** o **extrarregional**. Un *valor* tiene una **regionalidad** que es **intrarregional** o **extrarregional** o **nula**. Estas propiedades se definen como sigue:

1) **Localización**

Una *localización* es **intrarregional** si y sólo si se cumple una cualquiera de las condiciones siguientes:

- Es un *nombre de acceso* que es:
  - un *nombre de localización* declarado textualmente dentro de una *región* o *región de spec* y no definido en un *parámetro formal* de un procedimiento **crítico**;
  - un *nombre de* localización declarado textualmente dentro de un *modo de región* y no definido en un *parámetro formal* de un procedimiento componente **crítico**.
  - un *nombre de identidad-loc*, tal que la *localización* en su declaración es **intrarregional** o que se ha definido en un *parámetro formal* de un procedimiento **intrarregional**;

  - un *nombre de identidad-loc*, tal que la *localización* en su declaración es **intrarregional** o que se ha definido en un *parámetro formal* de un procedimiento componente **intrarregional**;
  - un *nombre de enumeración de localización*, en el que la *localización matriz* o *localización cadena* en la *acción hacer* asociada es **intrarregional**;
  - un *nombre de localización hacer-con*, en el que la *localización estructura* de la *acción hacer* asociada es **intrarregional**.
- Es una *referencia ligada desreferenciada*, en la que *el valor primitivo de referencia ligada* es **intrarregional**.
- Es una *referencia libre desreferenciada*, en la que el *valor primitivo de referencia libre* es **intrarregional**.
- Es un *descriptor desreferenciado*, en el que el *valor primitivo descriptor* es **intrarregional**.
- Es un *elemento de matriz* o *segmento de matriz*, en el que la *localización matriz* es **intrarregional**.
- Es un *elemento de cadena* o *segmento de cadena*, en el que la *localización cadena* es **intrarregional**.
- Es un *campo de estructura*, en el que la *localización estructura* es **intrarregional**.
- Es una *llamada a procedimiento que entrega una localización*, en la que en la *llamada a procedimiento* se especifica *un nombre de procedimiento* que es **intrarregional**.
- Es una *llamada a una rutina incorporada que entrega una localización*, que la definición CHILL o la implementación especifica que es **intrarregional**.
- Es una *conversión de localización* en la que la *localización modo estático* es **intrarregional**.

Una *localización* que no es **intrarregional** es **extrarregional**.

2) **Valor**

Un *valor* tiene una **regionalidad** que depende de su clase. Si tiene la clase M-derivada o la clase **general** o la clase **nula**, su **regionalidad** es **nula**. En otro caso, tiene la clase M-valuada o la clase M-referenciada y tiene una **regionalidad** que depende del modo M como sigue:

Si el *valor* tiene la clase M-valuada y M no tiene la **propiedad de referenciación**, la **regionalidad** es **nula**; en otro caso, el *valor* es un *operando-7* (y tiene la **propiedad de referenciación**) o una *expresión condicional*:

Si es un *valor primitivo,* entonces:

- Si es un *contenido de localización* que es una *localización*, entonces es el de esa *localización*.
- Si es un *contenido de localización componente* que es una *localización componente*, entonces es el de esa *localización* componente.
- Si es un *nombre de valor*, entonces:
  - si es un *nombre de sinónimo*, es el del *valor constante* en su definición;
  - si es un *nombre de valor hacer-con*, es el del *valor primitivo estructura* en la acción hacer asociada;
  - si es un *nombre de valor recibir*, es **extrarregional**.
- Si es una *tupla*, entonces, si una de las ocurrencias de *valor* en ella tiene **regionalidad** no **nula**, es la de ese *valor* (no importa la elección que se haga, véase 5.2.5 condiciones estáticas); en otro caso es **nula**.
- Si es un *valor elemento de matriz*, o un *valor segmento de matriz*, es el del *valor primitivo de matriz* en el mismo.
- Si es un *valor campo de estructura*, es el del *valor primitivo de estructura* en el mismo.
- Si es una *conversión de expresión*, es el de la *expresión* en el mismo.
- Si es una *llamada a procedimiento que entrega un valor*, es la de la *llamada a procedimiento* en el mismo.
- Si es una *llamada a procedimiento componente que entrega un valor*, es la de la *llamada a procedimiento* componente en el mismo.
- Si es una *llamada a rutina incorporada que entrega un valor*, que la definición CHILL o la implementación especifica que es **intrarregional** o **extrarregional**.

Si es una *localización referenciada*, entonces es el de la *localización* en el mismo.

Si es una *expresión condicional*, entonces si una de las ocurrencias de *subexpresión* en ella tiene **regionalidad** no **nula**, es la de esa *subexpresión* (no importa la elección que se haga, véase 5.3.2, condiciones estáticas); en todos los demás casos es **nula**.

3) **Nombre de procedimiento**

Un *nombre de procedimiento* es **intrarregional** si y sólo si está definido dentro de una *región* o *región de spec*, y no es **crítico** (es decir, no está otorgado por la región). En otro caso, es **extrarregional**.

Un *nombre de procedimiento componente* es **intrarregional** si y sólo si está definido dentro de un *modo región* y no es **crítico** (es decir, no está otorgado por el modo región). En otro caso, es **extrarregional**.

4) **Llamada a procedimiento**

Una *llamada a procedimiento* es **intrarregional** si contiene un *nombre de procedimiento* que es **intrarregional**; en otro caso, es **extrarregional**.

Una *llamada a procedimiento componente* es **intrarregional** si contiene un *nombre de procedimiento componente* que es **intrarregional**; en otro caso, es **extrarregional**.

Un *valor* es **regionalmente seguro** para un no-terminal (utilizado solamente para *localización*, *llamada a procedimiento* y *nombre de procedimiento*) si y sólo si:

- el no-terminal es **extrarregional** y el *valor* no es **intrarregional**;
- el no-terminal es **intrarregional** y el *valor* no es **extrarregional**;
- el no-terminal tiene **regionalidad nula**.

11.3 Demora de un hilo

Un proceso activo puede ser demorado ejecutando (evaluando) una de las siguientes acciones (expresiones):

- Acción demorar (véase 6.16).
- Acción demorar y elegir (véase 6.17).
- Acción recibir señal y elegir (véase 6.19.2).
- Acción recibir tampón y elegir (véase 6.19.3).
- Acción enviar tampón (véase 6.18.3).
- Acción llamar a un procedimiento componente de una localización de región (véase 3.15.3).
- Acción llamar a un procedimiento componente de una localización de tarea en el caso de que no hay suficiente espacio de almacenamiento para efectuar el paso c) 2) de 6.7 (véase 3.15.4).

Cuando un hilo es demorado mientras que el control sobre el mismo se encuentra dentro de un procedimiento **crítico**, o dentro de un procedimiento **componente crítico**, se liberará la región asociada. El contexto dinámico del proceso es retenido hasta que sea reactivado. El hilo trata entonces de bloquear de nuevo la región, lo que puede provocar que se suspenda.

11.4 Reactivación de un hilo

Un hilo demorado puede ser reactivado si es objeto de una supervisión de tiempo y si se produce una interrupción de tiempo (véase la cláusula 9). Puede también ser reactivado si otro hilo ejecuta una de las siguientes acciones:

- Acción continuar (véase 6.15).
- Acción enviar señal (véase 6.18.2).
- Acción enviar tampón (véase 6.18.3).
- Acción recibir tampón y elegir (véase 6.19.3).
- Liberación de una localización de región (véase 3.15.3).
- Al principio de la ejecución de un procedimiento componente llamado externamente de una localización de tarea (véase 3.15.4).

Cuando un hilo, después de haber bloqueado una región o localización de región, reactiva otro hilo, aquél sigue estando activo, es decir, no liberará la región o localización de región en ese punto.

11.5 Sentencias de definición de señal

sintaxis:

<sentencia de definición de señal> ::= *(1)*
SIGNAL *<definición de señal>* { , *<definición de señal>* }* ; *(1.1)*

<definición de señal> ::= *(2)*
<ocurrencia de definición> [= (*<modo>*{ ,*<modo>*}*)] [**TO***<nombre de proceso>*] *(2.1)*

semántica: Una definición de señal define una función de composición y descomposición para valores que han de transmitirse entre procesos. Si se envía una señal, se transmite la lista de valores especificada. Si en una acción recibir y elegir no hay ningún proceso en espera de la señal, se guardan los valores hasta que un proceso los reciba.

propiedades estáticas: Una *ocurrencia de definición* en una *definición de señal* define un nombre de **señal**.

Un nombre de **señal** tiene las siguientes propiedades:

- Tiene asociada una lista facultativa de modos, que son los mencionados en la *definición de señal*.
- Tiene asociado un nombre de **proceso** facultativo, que es el *nombre de proceso* especificado después de **TO**.

condiciones estáticas: En una *definición de señal*, ningún *modo* puede tener la **propiedad de no-valor**.

ejemplos:

15.27 **SIGNAL** *initiate* = (*INSTANCE*),

terminate; *(1.1)*

11.6 Compleción de localizaciones de región y de tarea

Una localización de REGIÓN L y una localización de TAREA L contienen colas de espera que contienen hilos que esperan para ejecutar un procedimiento de L.

Tras su creación L está *abierta*, es decir, las llamadas que no pueden ejecutarse inmediatamente se ponen en una cola de espera de L. Si L es un objeto de REGIÓN, el hilo llamante es también bloqueado.

Una localización de REGIÓN o TAREA (*RTL, REGION or TASK location*) puede estar *cerrada*. Si está cerrada no se ponen llamadas en cola, sino que se produce una excepción en la correspondiente acción llamar.

Una L RTL cerrada se ejecuta hasta que todas sus colas estén vacías. Después, L se pasa al estado *vacío*. Si se completan todas las RTL que dependen de L (véase 12.2.6), se *completa* la propia L.

Una RTL completada puede ser destruida. Si una RTL está en uno de los estados "abierta", "cerrada" o "vacía", no puede ser destruida.

12 Propiedades semánticas generales

12.1 Reglas de modos

12.1.1 Propiedades de modos y clases

12.1.1.1 Propiedad de lectura solamente

Informal

Un modo tiene la **propiedad de lectura solamente** si es un modo de **lectura solamente** o contiene un componente o un subcomponente, etc., que es un modo de **lectura solamente**.

Definición

Un modo tiene la **propiedad de lectura solamente** si y sólo si es:

- un modo matriz con un modo **elemento** que tiene la **propiedad de lectura solamente**;
- un modo de estructura en el que al menos uno de sus modos de **campo** tiene la **propiedad de lectura solamente**, y en el cual el campo no es un campo **marcador** con un modo de **lectura solamente** implícito de un modo **estructura parametrizada**;
- un modo de **lectura solamente**.

12.1.1.2 Modos parametrizables

Informal

Un modo es **parametrizable** si puede parametrizarse.

Definición

Un modo es **parametrizable** si y sólo si es:

- un modo cadena;
- un modo matriz;
- un modo estructura **variable parametrizable**.

12.1.1.3 Propiedad de referenciación

Informal

Un modo tiene la **propiedad de referenciación** si es un modo referencia o contiene un componente o un subcomponente, etc., que es un modo referencia.

Definición

Un modo tiene la **propiedad de referenciación** si y sólo si es:

- un modo referencia;
- un modo matriz con un modo **elemento** que tiene la **propiedad de referenciación**;
- un modo estructura en el que al menos uno de sus modos **campo** tiene la **propiedad de referenciación**.

12.1.1.4 Propiedad parametrizada con marcadores

Informal

Un modo tiene la **propiedad parametrizada con marcadores** si es un modo de estructura **parametrizada con marcadores** o contiene un componente o un subcomponente, etc., que es un modo de estructura **parametrizada con marcadores**.

Definición

Un modo tiene la **propiedad parametrizada con marcadores** si y sólo si es:

- un modo matriz con un modo **elemento** que tiene la **propiedad parametrizada con marcadores**;
- un modo estructura en el que al menos uno de sus modos de **campo** tiene la **propiedad parametrizada con marcadores**;
- un modo estructura **parametrizada con marcadores**.

12.1.1.5 Propiedad de no-valor

Informal

Un modo tiene la **propiedad de no-valor** si no existe para el mismo una expresión o denotación de valor primitivo.

Definición

Un modo tiene la **propiedad de no-valor** si y sólo si es:

- un modo evento, un modo tampón, un modo acceso, un modo asociación o un modo texto;
- un modo matriz con un modo **elemento** que tiene la **propiedad de no-valor**;
- un modo estructura en el que al menos uno de sus modos de **campo** tiene la **propiedad de no-valor**;
- un modo moreta **no asignable**;
- un modo moreta **abstracto**;
- un modo moreta en el que al menos uno de sus componentes tiene la **propiedad de no-valor**.

12.1.1.6 Modo raíz

Todo modo M tiene un modo **raíz** definido como:

- si M no es un modo intervalo discreto ni un modo intervalo coma flotante;
- el modo **progenitor** de M, si M es un modo intervalo discreto ni un modo intervalo coma flotante.

Toda clase M-valuada o M-derivada tiene un modo **raíz** que es el modo **raíz** de M.

12.1.1.7 Clase resultante

Dadas dos clases **compatibles** (véase 12.1.2.16) la primera de las cuales es la clase **general**, una clase M-valuada o una clase M-derivada, siendo M y N un modo discreto, un modo coma flotante, un modo conjuntista, o un modo cadena, la **clase resultante** se define como:

- la **clase resultante** de la clase M-valuada y la clase N-valuada es la clase R-valuada;
- la **clase resultante** de la clase M-valuada y la clase N-derivada o la clase **general** es la clase P-valuada;
- la **clase resultante** de la clase M-derivada y la clase N-derivada es la clase R-derivada;

- la **clase resultante** de la clase M-derivada y la clase **general** es la clase P-derivada;
- la **clase resultante** de la clase **general** y la clase **general** es la clase **general**;

siendo R el modo **resultante** de M y N, y P el modo **raíz** de M.

Dados dos modos **similares** M y N, el modo **resultante** R se define así:

- si el modo **raíz** de uno es un modo cadena **fijo** y el del otro es un modo cadena **variable**, el modo resultante es el modo **raíz** de aquél (M o N) cuyo modo **raíz** es un modo cadena **variable**;
- en otro caso es P.

Dada una lista C_i de clases **compatibles** dos a dos (i = 1, ..., n), la **clase resultante** de esta lista de clases se define recursivamente como la **clase resultante** de la **clase resultante** de la lista C_i (i = 1, ..., n – 1) y la clase C_n si n > 1; en otro caso, como la **clase resultante** de C_1 y C_1.

12.1.2 Relaciones sobre modos y clases

12.1.2.1 Generalidades

En las subcláusulas siguientes, se definen las relaciones de compatibilidad entre modos, entre clases y entre modos y clases. Estas relaciones se utilizan en todo esta Recomendación | Norma Internacional para definir condiciones estáticas.

Las propias relaciones de compatibilidad se definen en términos de otras relaciones que se utilizan principalmente en esta cláusula para la finalidad antes mencionada.

12.1.2.2 Relaciones de equivalencia sobre modos

Informal

Las siguientes relaciones de equivalencia desempeñan un papel en la formulación de las relaciones de compatibilidad:

- Dos modos son **similares** si son de misma naturaleza, es decir, tienen las mismas propiedades hereditarias.
- Dos modos son **v-equivalentes** (equivalentes en valor) si son **similares** y tienen también la misma **novedad**.
- Dos modos son **equivalentes** si, además de ser **v-equivalentes**, se tienen en cuenta posibles diferencias en la representación de valores en el almacenamiento en memoria o el tamaño mínimo de almacenamiento.
- Dos modos son **l-equivalentes** (equivalentes en localización) si, además de ser **equivalentes**, tienen la misma especificación de **lectura solamente**.
- Dos modos son **iguales** si no pueden distinguirse uno de otro, es decir, si todas las operaciones que pueden aplicarse a objetos de uno de los modos pueden aplicarse también a objetos del otro modo, a condición de que no se tenga en cuenta la **novedad**.
- Dos modos están **ligados por novedad** si son **iguales** y tienen la misma especificación de **novedad**.

Definición

En las subcláusulas siguientes se dan las relaciones de equivalencia entre modos en forma de un conjunto (parcial) de relaciones. Los algoritmos de equivalencia total se obtienen tomando el cierre simétrico, reflexivo y transitivo de este conjunto de relaciones. Los modos mencionados en las relaciones pueden ser virtualmente introducidos o dinámicos. En el segundo caso, la verificación de equivalencia completa sólo puede realizarse en el momento de la ejecución. El fallo de la verificación de la parte dinámica provocará la excepción *RANGEFAIL* o *TAGFAIL* (véanse las subcláusulas pertinentes).

La verificación de dos modos recursivos para cualquier equivalencia requiere la comprobación de los modos asociados en los caminos correspondientes del conjunto de modos recursivos mediante los cuales se definen. Existe equivalencia entre los modos si no se encuentra ninguna contradicción. (En consecuencia, un camino del algoritmo de verificación termina satisfactoriamente si se comparan dos modos que ya han sido comparados.)

12.1.2.3 La relación similar

Dos modos son **similares** si y sólo si:

- son modos enteros;
- son modos coma flotante;
- son modos booleanos;

- son modos carácter;
- son modos conjunto tales que:
  1) definen el mismo **número de valores**;
  2) para cada nombre de **elemento de conjunto** definido por un modo hay un nombre de **elemento de conjunto** definido por el otro modo que tiene la misma cadena de nombre y el mismo valor de representación;
  3) ambos son modos conjunto **numerados** o modos conjunto **no numerados**;
- son modos intervalo con modos **progenitores similares**;
- son modos intervalo coma flotante;
- uno es un modo intervalo discreto o un modo intervalo coma flotante cuyo modo **progenitor** es **similar** al otro modo;
- son modos conjuntistas tales que sus modos **miembro** son **equivalentes**;
- son modos referencia ligada tales que sus modos **referenciados** son **equivalentes**;
- son modos referencia libre;
- son modos descriptor tales que sus modos **origen referenciados** son **equivalentes**;
- son modos procedimiento tales que:
  1) tienen el mismo número de **especs de parámetro** y las **especs de parámetro** correspondientes (en posición) tienen modos **l-equivalentes** y los mismos atributos de parámetro, si existen;
  2) ambos tienen o ambos no tienen una **espec de resultado**. Si existen, las **especs de resultado** deben tener modos **l-equivalentes** y los mismos atributos, si existen;
  3) tienen la misma lista de nombres de **excepción**;
  4) tienen la misma **recursividad**;
- son modos ejemplar;
- son modos evento tales que ninguno tiene **longitud de evento** o ambos tienen la misma **longitud de evento**;
- son modos tampón tales que:
  1) ninguno tiene **longitud de tampón** o ambos tienen la misma **longitud de tampón**;
  2) tienen modos **elemento tampón l-equivalentes**;
- son modos asociación;
- son modos acceso tales que:
  1) ninguno tiene modo **índice** o ambos tienen modos **índice** que son **equivalentes**;
  2) uno al menos no tiene modo **registro**, o ambos tienen modos **registro** que son **l-equivalentes** y que ambos son modos **registro estáticos** o modos **registro dinámicos**;
- son modos texto tales que:
  1) tienen la misma **longitud de texto**;
  2) tienen modos **registro de texto l-equivalentes**;
  3) tienen modos **acceso l-equivalentes**;
- son modos duración;
- son modos tiempo absoluto;
- son modos cadena tales que sus modos **elemento** son **equivalentes**;
- son modos matriz tales que:
  1) sus modos **índice** son **v-equivalentes**;
  2) sus modos **elemento** son **equivalentes**;
  3) sus **organizaciones de elementos** son **equivalentes**;
  4) tienen el mismo **número de elementos**. Esta verificación es dinámica si uno o ambos modos es (son) dinámico(s). El fallo de la verificación producirá la excepción *RANGEFAIL*;

- son modos estructura que no son modos estructura **parametrizada** tales que:
  1) en la sintaxis estricta, tienen el mismo número de *campos* y los *campos* correspondientes (en posición) son **equivalentes**;
  2) si ambos son modos estructura **variable parametrizable**, sus listas de clases deben ser **compatibles**;
- son modos estructura **parametrizada** tales que:
  1) sus modos estructura **variable origen** son **similares**;
  2) sus valores correspondientes (en posición) son los mismos. Esta verificación es dinámica si uno o ambos modos son dinámicos. El fallo de la verificación producirá la excepción *TAGFAIL*.
- son modos moreta cuyos nombres de modo son sinónimos.

12.1.2.4 La relación v-equivalente

Dos modos son **v-equivalentes** si y sólo si son **similares** y tienen la misma **novedad**.

12.1.2.5 La relación equivalente

Dos modos son **equivalentes** si y sólo si son **v-equivalentes** y:

- si uno es un modo intervalo discreto, el otro debe ser también un modo intervalo discreto, y tanto los **límites superiores** como los **límites inferiores** deben ser iguales;
- si uno es un modo intervalo coma flotante, el otro debe ser también un modo intervalo coma flotante, y tanto los **límites superiores** como los **límites inferiores** deben ser iguales y tener la misma **precisión**;
- si uno es un modo cadena **fijo**, el otro debe ser también un modo cadena **fijo**, y ambos deben tener la misma **longitud de cadena**. Esta verificación es dinámica si uno o ambos modos son dinámicos. El fallo de la verificación producirá la excepción *RANGEFAIL*;
- si uno es un modo cadena **variable**, el otro debe ser también un modo cadena **variable**, y ambos deben tener la misma **longitud de cadena**. Esta verificación es dinámica si uno o ambos modos son dinámicos. El fallo de la verificación producirá la excepción *RANGEFAIL*.

12.1.2.6 La relación l-equivalente

Dos modos son **l-equivalentes** si y sólo si son **equivalentes**, y si uno es un modo de **lectura solamente**, el otro también debe ser un modo de **lectura solamente**, y:

- si ambos modos son de referencia ligada, sus modos **referenciados** deben ser **l-equivalentes**;
- si ambos son modos descriptor, sus modos **origen referenciados** deben ser **l-equivalentes**;
- si ambos son modos matriz, sus modos **elemento** deben ser **l-equivalentes**;
- si ambos son modos estructura no **parametrizada**, los *campos* correspondientes (en posición), en la sintaxis estricta, tienen que ser **l-equivalentes**. Si son modos estructura **parametrizada**, sus modos de estructura **variable origen** deben ser **l-equivalentes**.

12.1.2.7 Las relaciones equivalente y l-equivalente para campos

Dos *campos* (ambos en el contexto de dos modos estructura dados) son 1. **equivalentes**, 2. **l-equivalentes** si y sólo si ambos *campos* son *campos fijos* que son 1. **equivalentes**, 2. **l-equivalentes**, o ambos son *campos de alternativa* que son 1. **equivalentes**, 2. **l-equivalentes**.

Las relaciones **equivalente** y **l-equivalente** se definen recursivamente para los correspondientes *campos fijos*, *campos variables*, *campos de alternativa* y *alternativas variables*, respectivamente, de la siguiente forma:

- *Campos fijos* y *campos variables*
  1) Los *campos fijos* y los *campos variables* deben tener una **organización de campo equivalente**.
  2) Ambos modos **campo** deben ser 1.**equivalentes**, 2. **l-equivalentes**.
- *Campos de alternativa*
  1) Ambos *campos de alternativa* tienen *listas de marcadores* o ninguno tiene *listas de marcadores*. En el primer caso, las *listas de marcadores* deben tener el mismo número de nombres de **campo marcador** y los correspondientes nombres (en posición) de **campo marcador** deben designar los *campos fijos* correspondientes.

2) Ambos deben tener el mismo número de *alternativas variables* y las **alternativas variables** correspondientes (en posición) deben ser 1. **equivalentes**, 2. **l-equivalentes**.

3) Ninguno debe tener especificado **ELSE** o ambos deben tener especificado **ELSE**. En el último caso, debe seguir el mismo número de *campos variables* y los *campos variables* correspondientes (en posición) deben ser 1. **equivalentes**, 2. **l-equivalentes**.

- *Alternativas variables*

1) Ambas *alternativas variables* deben tener el mismo número de *listas de etiquetas de caso* y las *listas de etiquetas de caso* correspondientes (en posición) deben ser ambas *irrelevantes*, o definir ambas el mismo conjunto de valores.

2) Ambas *alternativas variables* deben tener el mismo número de *campos variables* y los *campos variables* correspondientes (en posición) deben ser 1. **equivalentes**, 2. **l-equivalentes**.

12.1.2.8 La relación equivalente para organización

En el resto de esta subcláusula, se supondrá que cada *pos* es de la forma:

POS (<*número*>, <*bit inicial*>, <*longitud*>)

y que cada *paso* es de la forma:

STEP (<*pos*>, <*tamaño de paso*>)

La subcláusula 3.13.5 da las reglas adecuadas para poner *pos* o *paso* en la forma requerida.

- *Organización de campo*

Dos **organizaciones de campo** son **equivalentes** si son ambas **NOPACK**, ambas **PACK** o ambas *pos*. En el último caso, una *pos*, debe ser **equivalente** a la otra (véase más adelante).

- *Organización de elementos*

Dos **organizaciones de elementos** son **equivalentes** si son ambas **NOPACK**, ambas **PACK** o ambas *paso*. En el último caso, la *pos* de un *paso* debe ser **equivalente** a la *pos* del otro (véase más adelante) y el *tamaño de paso* debe entregar los mismos valores para las dos **organizaciones de elementos**.

- *Pos*

Una *pos* es **equivalente** a otra *pos* si y sólo si ambas ocurrencias de *palabra* entregan el mismo valor, ambas ocurrencias de *bit inicial* entregan el mismo valor, y ambas ocurrencias de *longitud* entregan el mismo valor.

12.1.2.9 La relación igual

Dos modos son **iguales** si y sólo si ambos o ninguno son de **lectura solamente** y ambos tienen la **novedad nula** o ambos tienen la misma **novedad** y:

- son modos enteros;
- son modos booleanos;
- son modos carácter;
- son modos conjunto **similares**;
- son modos intervalo discreto con **límites superiores** iguales y **límites inferiores** iguales;
- son modos intervalo coma flotante con **límites superiores** iguales y **límites inferiores** iguales e igual **precisión**;
- son modos conjuntistas tales que sus modos **miembro** son **iguales**;
- son modos referencia ligada tales que sus modos **referenciados** son **iguales**;
- son modos referencia libre;
- son modos descriptor tales que sus modos **origen referenciados** son **iguales**;

- son modos procedimiento tales que:
  1) tienen el mismo número de **especs de parámetro** y las **especs de parámetro** correspondientes (en posición) tienen modos **iguales** y los mismos atributos de parámetro, si existen;
  2) ambos o ninguno tienen una **espec de resultado**. Si están presentes, las **especs de resultado** deben tener modos **iguales** y los mismos atributos, si existen;
  3) ambos tienen la misma lista de nombres de **excepción**;
  4) tienen la misma **recursividad**;
- son modos ejemplar;
- son modos evento tales que ninguno tiene **longitud de evento** o ambos tienen la misma **longitud de evento**;
- son modos tampón tales que:
  1) ninguno tiene **longitud de tampón** o ambos tienen la misma **longitud de tampón**;
  2) tienen modos **elemento tampón** que son **iguales**;
- son modos asociación;
- son modos acceso tales que:
  1) ninguno tiene modo **índice** o ambos tienen modos **índice** que son **iguales**;
  2) uno al menos no tiene modo **registro** o ambos tienen modos **registro** que son **iguales**, y que ambos son modos **registro estáticos** o modos **registro dinámicos**;
- son modos texto tales que:
  1) tienen la misma **longitud de texto**;
  2) sus modos **registro de texto** son **iguales**;
  3) sus modos **acceso** son **iguales**;
- son modos duración;
- son modos tiempo absoluto;
- son modos cadena tales que:
  1) sus modos **elemento** son **iguales**;
  2) tienen la misma **longitud de cadena**;
  3) ambos son modos cadena **fijos** o modos cadena **variables**;
- son modos matriz tales que:
  1) sus modos **índice** son **iguales**;
  2) sus modos **elemento** son **iguales**;
  3) sus **organizaciones de elementos** son **equivalentes**;
  4) tienen el mismo **número de elementos**;
- son modos estructura que no son modos estructura **parametrizada** tales que:
  1) en la sintaxis estricta, tienen el mismo número de *campos*, y los *campos* correspondientes (en posición) son **iguales**;
  2) si ambos son modos estructura **variable parametrizable**, sus listas de clases deben ser **compatibles**;
- son modos estructura **parametrizada** tales que:
  1) sus modos estructura **variable origen** son **iguales**;
  2) sus valores correspondientes (en posición) son los mismos.

12.1.2.10 La relación igual para campos

Dos *campos* (ambos *campos* en el contexto de dos modos estructura dados) son **iguales** si y sólo si ambos *campos* son *campos fijos* que son **iguales** o ambos son *campos de alternativa* que son **iguales**.

La relación **igual** se define recursivamente para los correspondientes *campos fijos*, *campos variables*, *campos de alternativa* y *alternativas variables*, respectivamente, de la siguiente forma:

- *Campos fijos* y *campos variables*
  1) Los *campos fijos* y los *campos variables* deben tener una **organización de campo equivalente**.
  2) Ambos modos **campo** deben ser **iguales**.
  3) Los *campos fijos* y los *campos variables* deben tener asociada la misma *cadena de nombre*.
- *Campos de alternativa*
  1) Ambos *campos de alternativa* tienen *listas de marcadores* o ninguno tiene *listas de marcadores*. En el primer caso, las *listas de marcadores* deben tener el mismo número de nombres de **campo marcador**, y los correspondientes nombres de **campo marcador** (en posición) deben designar los *campos fijos* correspondientes.
  2) Ambos deben tener el mismo número de *alternativas variables* y las *alternativas variables* correspondientes (en posición) deben ser **iguales**.
  3) Ninguno debe tener especificado **ELSE** o ambos deben tener especificado **ELSE**. En el último caso, debe seguir el mismo número de *campos variables*, y los *campos variables* correspondientes (en posición) deben ser **iguales**.
- *Alternativas variables*
  1) Ambas *alternativas variables* deben tener el mismo número de *listas de etiquetas de caso* y las *listas de etiquetas de caso* correspondientes (en posición) deben ser ambas *irrelevantes* o definir ambas el mismo conjunto de valores.
  2) Ambas *alternativas variables* deben tener el mismo número de *campos variables*, y los *campos variables* correspondientes (en posición) deben ser **iguales**.

12.1.2.11 La relación ligado por novedad

Informal

En un programa, cada **cuasi** neomodo debe representar, a lo sumo un neomodo **real**. Esto se establece de la manera siguiente: cuando una *cadena de nombre* está **ligada** a una *ocurrencia de definición* **real** y a una **cuasi** *ocurrencia de definición*, todos los neomodos que intervienen están apareados. La relación **ligado por novedad** se establece entonces entre **novedades**.

Definición

La relación **apareados por novedad** se aplica entre dos modos y un dominio. Para cada *cadena de nombre* **ligada** en un dominio R a una *ocurrencia de definición* **real** y una **cuasi** *ocurrencia de definición*:

- si son nombres de **sinónimo**, los modos **raíz** de sus clases están **apareados por novedad** en R;
- si son nombres de **localización** o de **identidad-loc**, los modos de localización están **apareados por novedad** en R;
- si son nombres de **procedimiento**, los modos de las **especs de parámetro** y de las **especs de resultado**, si existen, están **apareados por novedad** en R;
- si son nombres de **proceso**, los modos de las **especs de parámetro** están **apareados por novedad** en R;
- si son de nombres de **señal**, los modos en la lista de modos están **apareados por novedad** en R.

Si dos modos están **apareados por novedad** en un dominio R, entonces:

- si son modos conjuntistas, sus modos **miembro** están **apareados por novedad** en R;
- si son modos referencia ligada, sus modos **referenciados** están **apareados por novedad** en R;
- si son modos descriptores, sus modos **origen referenciados** están **apareados por novedad** en R;
- si son modos procedimiento, los modos de sus **especs de parámetro** y de sus **especs de resultado**, si existen, están **apareados por novedad** en R;
- si son modos tampón, sus modos **elemento tampón** están **apareados por novedad** en R;
- si son modos acceso, sus modos **índice**, si existen, y modos **registro**, si existen, están **apareados por novedad** en R;
- si son modos texto, sus modos **índice**, si existen, están **apareados por novedad** en R;
- si son modos matriz, sus modos **índice** y modos **elemento** están **apareados por novedad** en R;

- si son modos estructura **parametrizada**, sus modos estructura **variable origen** están **apareados por novedad** en R;
- si son modos estructura **variable parametrizada**, sus modos **campo** y los modos de las clases en su lista de clases están **apareados por novedad** en R;
- en otro caso, si son modos estructura, sus modos **campo** están **apareados por novedad** en R.

Si dos modos están **apareados por novedad** en un dominio R y sus **novedades** no son iguales, la **novedad real** y la **cuasi novedad** del modo están ligadas **por novedad**, entre sí, en R.

Dos **novedades** se consideran las mismas si son:

- la misma **novedad real**, o
- una **novedad real** y una **cuasi novedad** que están **ligadas por novedad**.

12.1.2.12 La relación de lectura compatible

Informal

La relación de **lectura compatible** es aplicable a modos **equivalentes**. Se dice que un modo M es de **lectura compatible** con un modo N si M o sus posibles (sub)componentes tienen especificaciones de **lectura solamente** iguales o más restrictivas, y, si son modos referencia, se refieren a localizaciones **l-equivalentes**. Esta relación es por tanto no simétrica.

Ejemplo:

READ REF READ *CHAR* es de **lectura compatible** con **REF READ** *CHAR*

Definición

Se dice que un modo M es de **lectura compatible** con un modo N (relación no simétrica) si y sólo si M y N son **equivalentes** y, si N es un modo de **lectura solamente**, entonces M debe ser también un modo de **lectura solamente** y además:

- si M y N son modos referencia ligada, el modo **referenciado** de M debe ser **l-equivalente** con el modo **referenciado** de N;
- si M y N son modos descriptor, el modo **origen referenciado** de M debe ser **l-equivalente** con el modo **origen referenciado** de N;
- si M y N son dos modos matriz, el modo **elemento** de M debe ser de **lectura compatible** con el modo **elemento** de N;
- si M y N son modos estructura que no son modos estructura **parametrizada**, todo modo **campo** de M debe ser de **lectura compatible** con el modo **campo** correspondiente de N. Si M y N son modos estructura **parametrizada**, el modo estructura **variable origen** de M tiene que ser de **lectura compatible** con el modo estructura **variable origen** de N.

12.1.2.13 Las relaciones equivalente dinámica y de lectura compatible dinámica

Informal

Las relaciones 1. **equivalente dinámica**, 2. de **lectura compatible dinámica** se refieren únicamente a modos que pueden ser dinámicos, a saber, modos cadena, matriz y estructura **variable**. Se dice que un modo M **parametrizable** es 1. **equivalente dinámico**, 2. de **lectura compatible dinámica** con un modo N (posiblemente dinámico), si existe una versión dinámicamente parametrizada de M que sea 1. **equivalente**, 2. de **lectura compatible** con N.

Definición

Un modo M es 1. **equivalente dinámico** a un modo N, 2. de **lectura compatible dinámica** con un modo N (una relación no simétrica) si y sólo si se cumple una de las condiciones siguientes:

- M y N son modos cadena tales que M(p) es 1. **equivalente**, 2. de **lectura compatible** con N, donde p es la longitud (posiblemente dinámica) de N. El valor p no debe ser mayor que la **longitud de cadena** de M. Esta comprobación es dinámica si N es un modo dinámico. El fallo de la comprobación producirá una excepción *RANGEFAIL*.
- M y N son modos matriz tales que M(p) es 1. **equivalente**, 2. de **lectura compatible** con N, donde p es tal que *NUM* (p) – *LOWER* (M) + 1 es el **número de elementos** (posiblemente dinámico) de N. El valor p no debe ser mayor que el **límite superior** de M. Esta comprobación es dinámica si N es un modo dinámico. El fallo de la comprobación producirá una excepción *RANGEFAIL*.
- M es un modo estructura **variable parametrizable** y N es un modo estructura **parametrizable**, tales que M(p_1, ..., p_n) es 1. **equivalente**, 2. de **lectura compatible** con N, donde p_1, ..., p_n denota la lista de valores de N.

12.1.2.14 La relación restringible

Informal

La relación **restringible** se refiere a modos **equivalentes** con la **propiedad de referenciación**. Se dice que un modo M es **restringible** a un modo N, si él o sus posibles (sub)componentes se refieren a localizaciones con una especificación de **lectura solamente** igual o más restrictiva que los referenciados por N. Esta relación es por tanto no simétrica.

Ejemplo:

REF READ *INT* es **restringible** a **REF** *INT*

STRUCT *(P **REF READ** BOOL)* es **restringible** a **STRUCT** *(Q **REF** BOOL)*

Definición

Un modo M es **restringible** a un modo N (relación no simétrica) si y sólo si M y N son **equivalentes** y, además:

- si M y N son modos referencia ligada, el modo **referenciado** de M debe ser de **lectura compatible** con el modo **referenciado** de N;
- si M y N son modos descriptor, el modo **referenciado origen** de M debe ser de **lectura compatible** con el modo **origen referenciado** de N;
- si M y N son modos matriz, el modo **elemento** de M debe ser **restringible** al modo **elemento** de N;
- si M y N son modos estructura, cada modo **campo** de M debe ser **restringible** al modo **campo** correspondiente de N.

12.1.2.15 Compatibilidad entre un modo y una clase

- Todo modo M es **compatible** con la clase **general**.
- Un modo M es **compatible** con la clase **nula** si y sólo si M es un modo referencia, un modo procedimiento o un modo ejemplar.
- Un modo M es **compatible** con la clase N-referencia si y sólo si es un modo referencia y se cumple una de las siguientes condiciones:
  1) N es un modo no moreta estático y M un modo referencia ligada cuyo modo **referenciado** es de **lectura compatible** con N;
  2) N es un modo moreta estático y M un modo referencia ligada REF MM, y MM y N están en el mismo camino;
  3) N es un modo estático y M un modo referencia libre;
  4) M es un modo descriptor cuyo modo **origen referenciado** es de **lectura dinámica compatible** con N.
- Un modo M es **compatible** con la clase N-derivada si y sólo si M y N son **similares**.
- Un modo M es **compatible** con la clase de N-valuada si y sólo si se cumple una de las siguientes condiciones:
  1) si M no tiene la **propiedad de referenciación**, M y N deben ser **v-equivalentes**.
  2) si M tiene la **propiedad de referenciación**, M debe ser **restringible** a N.

12.1.2.16 Compatibilidad entre clases

- Toda clase es **compatible** consigo misma.
- La clase **general** es **compatible** con cualquier otra clase.
- La clase **nula** es **compatible** con cualquier clase de referencia M.
- La clase **nula** es **compatible** con la clase M-derivada o la clase M-valuada si y sólo si M es un modo referencia, un modo procedimiento o un modo ejemplar.
- La clase M-referencia es **compatible** con la clase N-referencia si y sólo si M y N son **equivalentes**. Si M y/o N es (son) un modo dinámico, se ignora la parte dinámica de la verificación de equivalencia, es decir, no pueden ocurrir excepciones.
- La clase M-referencia es **compatible** con la clase N-valuada si y sólo si N es un modo referencia y se cumple una de las condiciones siguientes:
  1) M es un modo estático y N un modo referencia ligada cuyo modo **referenciado** es **equivalente** a M.
  2) M es una localización de modo estático y N es un modo referencia libre.
  3) N es un modo descriptor cuyo modo **origen referenciado** es **equivalente dinámico** con M.

- La clase M-derivada es **compatible** con la clase N-derivada o con la clase N-valuada si y sólo si M y N son **similares**.
- La clase de M-valuada es **compatible** con la clase N-valuada si y sólo si M y N son **v-equivalentes**.

Dos listas de clases son **compatibles** si y sólo si ambas tienen el mismo número de clases y las clases correspondientes (en posición) son **compatibles**.

12.1.2.17 Conformidad de nombres de modo

Dos nombres de modo A y B son conformes uno con otro si y sólo si:

- o bien ambos denotan modos del tipo "REF MM", donde MM es un modo moreta y A y B están en el mismo camino;
- o bien A syn B (A sinónimo B).

12.1.3 Definiciones para modos moreta

Si M es un modo moreta, entonces:

M_S	=	la parte especificación de M (también el conjunto de componentes en esta parte);
M_B	=	la parte cuerpo de M (también el conjunto de componentes en esta parte);
M_P	=	el conjunto de componentes públicos de M_S definidos directamente en M_S;
M_{P+}	=	el conjunto de todos los componentes públicos de M_S (incluidos los heredados);
M_I	=	el conjunto de componentes internos de M_S;
M_{I+}	=	el conjunto de todos los componentes internos de M_S (incluidos los heredados);
M_R	=	el conjunto de componentes privados de M_B;
M_{R+}	=	el conjunto de todos los componentes privados de M_S (incluidos los heredados);
M_{CD}	=	el conjunto de constructores y destructores de M_S;
M_{inv}	=	la invariable de M_S;
M_O	=	el conjunto de componentes contenidos (lógicamente) en una localización de modo M.

Si P es un procedimiento componente de un modo moreta, entonces:

PS	=	la parte firma de P;
PD	=	la definición (completa) de P;
PPre	=	la precondición de P;
PPost	=	la poscondición de P;
PE	=	el conjunto de excepciones especificado en PS.

Si X es un procedimiento o un modo moreta, entonces:

attr(X, A)	=	X contiene el atributo A: por ejemplo attr(P, INLINE);
prop(X, P)	=	X tiene la propiedad P: por ejemplo. prop(P, asignable);
GRANTed	=	exportado explícitamente;
granted	=	GRANTed $\vee$ exportado implícitamente.

12.1.3.1 Nombres calificados de componentes de modo moreta y de localizaciones moreta

Si M es la cadena de nombre simple de un modo moreta, L es la cadena de nombre simple de una localización moreta, y C es el nombre simple de un componente de M o de un componente público de L, entonces el nombre M.C o L.C puede utilizarse como un nombre único para C con el fin de distinguir C de componentes con la misma cadena de nombre simple. Si es necesario, se supone el nombre calificado.

12.1.3.2 Relaciones de sucesor y de predecesor para nombres de modo moreta

Un nombre de modo moreta DM es un sucesor directo (dsucc) de un nombre de modo moreta BM si y sólo si existen nombres de modo moreta D y B: (B syn BM) $\wedge$ (D syn DM) $\wedge$ (B se menciona en una cláusula de herencia de D).

Un nombre de modo moreta DM es un sucesor (succ) de un nombre de modo moreta BM si y solo si o bien DM syn BM o ($\exists$MM: (DM succ MM) $\wedge$ (MM dsucc BM)).

La relación "predecesor" es la inversa de la relación "sucesor".

Dos nombres de modo moreta A y B están en el mismo camino si y sólo si (A succ B) ∨ (B succ A).

Estas relaciones se cumplen isomórficamente para modos del tipo "REF MM", donde MM es un modo moreta.

12.1.3.3 Concordancia entre firmas de procedimiento y definiciones de procedimiento

Una firma de procedimiento guardado S concuerda con una definición de procedimiento guardado D si y sólo si:

S.<lista de parámetros> concuerda con D.<lista de parámetros formales> ∧

S.<espec de resultado> y D.<espec de resultado> difieren como máximo en la ocurrencia de RESULT ∧

S.<lista de excepciones> = D.<lista de excepciones> ∧

S.<lista de atributos de procedimiento guardado> = D.<lista de atributos de procedimiento guardado>

Una lista de parámetros P concuerda con una lista de parámetros formales F con sintaxis estricta F′ si y sólo si:

|P| = |F′| ∧

todos los elementos correspondientes de P y F′ tienen el mismo modo y los mismos atributos de parámetros.

12.2 Visibilidad y vinculación de nombres

La definición de visibilidad y de vinculación de nombres (o identificación) se basa en la terminología siguiente:

- *cadena de nombre*: denota una cadena terminal que tiene asociada una cadena de nombre **canónica** (véase 2.7) y propiedades de visibilidad;
- nombre: denota una *cadena de nombre simple* asociada con la *ocurrencia de definición* que la ha creado (véase 10.1);
- *nombre*: denota una ocurrencia aplicada de un nombre (con una cadena de nombre posiblemente prefijada).

12.2.1 Grados de visibilidad

Las reglas de vinculación se basan en la visibilidad de *cadenas de nombre* en los dominios de un programa. Dentro de un dominio, cada *cadena de nombre* tiene uno de los siguientes grados de visibilidad:

Cuadro 1/Z.200 – Grados de visibilidad

Visibilidad	Propiedades (informales)
visible directamente	La *cadena de nombre* es **visible** por creación, otorgamiento o toma o por herencia de spec a cuerpo
visible indirectamente	La *cadena de nombre* es predefinida heredada por anidamiento de bloques
visible públicamente	La *cadena de nombre* es un nombre de un componente público de un *modo moreta* y se utiliza en un nombre componente moreta, o la cadena de nombre es un nombre de un componente de un modo moreta M y se utiliza en un nombre componente moreta que aparece dentro de M o de cualquier sucesor de M.
visible privadamente	La *cadena de nombre* es un nombre de una sentencia de definición de procedimiento guardado P contenido en un cuerpo de modo moreta B y la especificación de modo moreta de B no contiene una sentencia de firma de procedimiento guardado correspondiente.
invisible	La *cadena de nombre* no puede aplicarse

Se dice que una *cadena de nombre* es **visible** en un dominio si es **visible directamente** o **visible indirectamente** en ese dominio. En otro caso se dice que la *cadena de nombre* es **invisible** en ese dominio. Las sentencias de estructuración del programa y las sentencias de visibilidad determinan unívocamente la clase de visibilidad a la que pertenece cada *cadena de nombre*.

Cuando una *cadena de nombre* es **visible** en un dominio, puede estar **enlazada** (dícese también vinculada) **directamente** a otra *cadena de nombre* en otro dominio, o **enlazada indirectamente** a una *ocurrencia de definición* en el programa. Las reglas para el **enlace directo** se indican en 12.2.3. Obsérvese que toda aplicación de una regla introduce un nuevo **enlace directo** para una *cadena de nombre*.

Sobre la base del **enlace directo**, la noción de **enlace** (no necesariamente **directo**) se define como sigue:

Una *cadena de nombre* N_1, **visible** en el dominio R_1, se dice que está **enlazada** a una *cadena de nombre* N_2 en el dominio R_2 o a una *ocurrencia de definición* D, y sólo si se cumple una de las siguientes condiciones:

- N_1 en R_1 está **enlazada directamente** a N_2 en R_2 o a D. Sin embargo, si N_1 está **enlazada directamente** a más de una *ocurrencia de definición* en R_1, entonces todas estas *ocurrencias de definición*, excepto una, son superfluas, y N_1 está **enlazada** a una de ellas, cualquiera que ésta sea, en R_1. Esto no se aplica si N_1 es la cadena de nombre de una sentencia de firma de procedimiento guardado simple en una especificación de modo moreta.
- N_1 en R_1 está **enlazada directamente** a alguna N en alguna R, y N en R está **enlazada** a N_2 en R_2 o a D.

12.2.2 Condiciones de visibilidad y vinculación de nombres

En cada dominio de un programa, deben cumplirse las siguientes condiciones:

- Si una *cadena de nombre* es **visible** en un dominio y tiene más de un **enlace directo**, entonces debe estar **enlazada** a exactamente una *ocurrencia de definición* **real** y a una **cuasi** *ocurrencia de definición*, o a exactamente una *ocurrencia de definición* **real** en una sentencia de firma de procedimiento guardado **simple** en un modo M que no es un modo interfaz y a exactamente una ocurrencia de definición **real** en una sentencia de definición de procedimiento guardado **simple** correspondiente y posiblemente a varias *ocurrencias de definición* **reales** en una sentencia de firma de procedimiento guardado **simple** en uno o más modos interfaz que son modos base de M, o posiblemente a varias *ocurrencias de definición* **reales** en una sentencia de firma de procedimiento guardado **simple** en uno o más modos interfaz que son modos base de un modo moreta M y donde la cadena de nombre no tiene enlace directo en M.

Una *cadena de nombre* NS, **visible** en el dominio R, se dice que está **ligada** en R a varias *ocurrencias de definición* de acuerdo con las siguientes reglas:

- Si NS es **visible** en R, NS está **ligada** a las *ocurrencias de definición* con las que está **enlazada** en R (como una *cadena de nombre* **visible**). Si está **ligada** a una **cuasi** *ocurrencia de definición* y a una *ocurrencia de definición* **real**, la **cuasi** *ocurrencia de definición* es redundante y no participa adicionalmente en la visibilidad ni en la vinculación de nombre (es decir, no es tomada, otorgada ni heredada). Si está enlazada a exactamente una *ocurrencia de definición* **real** en una sentencia de firma de procedimiento guardado **simple** en un modo M que no es un modo interfaz y a exactamente una ocurrencia de definición **real** en una sentencia de definición de procedimiento guardado **simple** correspondiente y posiblemente a varias *ocurrencias de definición* **reales** en una sentencia de firma de procedimiento guardado **simple** en uno o más modos interfaz que son modos base de M, entonces está ligada a la ocurrencia en M.
- En otro caso, NS no está **ligada** en R.

condición estática: La *cadena de nombre* asociada a cada *nombre* directamente encerrado en un dominio debe estar **ligada** en ese dominio.

vinculación de nombres: Un *nombre* N con una *cadena de nombre* NS asociada en un dominio R está **ligado** a las *ocurrencias de definición* con las que NS está **ligada** en R.

12.2.3 Visibilidad en dominios

12.2.3.1 Generalidades

Una *cadena de nombre* es **visible directamente** en un dominio según las reglas siguientes:

- la *cadena de nombre* es introducida por toma en el dominio (véase 12.2.3.5);
- la *cadena de nombre* es introducida por otorgamiento en el dominio (véase 12.2.3.4);
- hay una *ocurrencia de definición* con esa *cadena de nombre* en el dominio. En ese caso, la *cadena de nombre* en el dominio está **enlazada directamente** a la *ocurrencia de definición*. (Obsérvese que la *cadena de nombre* puede estar **enlazada directamente** a varias *ocurrencias de definición* en el dominio);
- dentro de un constructor o destructor CD de un modo moreta M, la cadena de nombre de M no es ocultada por la ocurrencia de definición de la misma cadena de nombre en la definición de CD (pero puede estar todavía ocultada por otras ocurrencias de definición de la misma cadena de nombre);
- en un lugar dentro de un constructor o destructor CD de un modo moreta M, donde la cadena de nombre S de M no está oculta, S denota M o CD según el contexto;
- el dominio es un 1. *cuerpo de módulo*, 2. *cuerpo de región*, y la *cadena de nombre* es **visible directamente** en el dominio de un 1. *módulo de spec*, 2. *región de spec* **correspondiente**. La *cadena de nombre* está **enlazada directamente** a la *cadena de nombre* en el dominio correspondiente.

Una *cadena de nombre* que no sea **visible directamente** en un dominio es **visible indirectamente** en el mismo según las reglas siguientes:

- El dominio es un bloque, y la *cadena de nombre* es **visible directamente** en el dominio directamente circundante. Se dice que la *cadena de nombre* es heredada por el bloque, y está **enlazada directamente** a la misma *cadena de nombre* en el dominio directamente circundante.
- El dominio no es un bloque en que la *cadena de nombre* es heredada y la *cadena de nombre* es una *cadena de nombre* definida por el lenguaje (véase III.2) o por la implementación. Se considera que la *cadena de nombre* está **enlazada directamente** a una *ocurrencia de definición* en el dominio de la definición de proceso más imaginario más externo para su significado predefinido.

12.2.3.2 Sentencias de visibilidad

sintaxis:

<sentencia de visibilidad> ::= *(1)*
<sentencia de otorgamiento> *(1.1)*
| *<sentencia de toma>* *(1.2)*

semántica: La sentencias de visibilidad sólo están autorizadas en dominios de modulión y controlan la visibilidad de las *cadenas de nombre* mencionadas en los mismos.

propiedades estáticas: Una *sentencia de visibilidad* tiene asociados uno o dos dominios **origen** (véase 10.2) y uno o dos dominios **destino**, definidos como sigue:

- Si la *sentencia de visibilidad* es una *sentencia de toma*, su dominio **destino**, es el dominio que circunda directamente a la *sentencia de toma*, y sus dominios **origen** son los dominios que circundan directamente a ese dominio.
- Si la *sentencia de visibilidad* es una *sentencia de otorgamiento*, su dominio **origen** es el dominio que circunda directamente a la *sentencia de otorgamiento*, y sus dominios **destino** son los dominios que encierran directamente a ese dominio.
- Si la *sentencia de visibilidad* es una sentencia de otorgamiento en una especificación de modo moreta, su dominio **origen** es el dominio que circunda directamente a la *sentencia de otorgamiento*, y sus dominios **destino** no son los dominios que encierran directamente a ese dominio.

12.2.3.3 Cláusula de redenominación por prefijo

sintaxis:

<cláusula de redenominación por prefijo> ::= *(1)*
(*<prefijo antiguo>–> <prefijo nuevo>*) ! *<posfijo>* *(1.1)*

<prefijo antiguo> ::= *(2)*
<prefijo> *(2.1)*
| *<vacío>* *(2.2)*

<prefijo nuevo> ::= *(3)*
<prefijo> *(3.1)*
| *<vacío>* *(3.2)*

<posfijo> ::= *(4)*
<posfijo de toma> { , *<posfijo de toma>*}* *(4.1)*
| *<posfijo de otorgamiento>* { , *<posfijo de otorgamiento>*}* *(4.2)*

sintaxis derivada: Una *cláusula de redenominación por prefijo* en la cual el *posfijo* consiste en más de un *posfijo de toma (posfijo de otorgamiento)* es sintaxis derivada para varias *cláusulas de redenominación por prefijo*, una para cada *posfijo de toma (posfijo de otorgamiento)*, separadas por comas, con los mismos *prefijo antiguo* y *prefijo nuevo*.

Por ejemplo :

GRANT *(p –> q) ! a, b ;*

es la sintaxis derivada para:

GRANT *(p –> q) ! a, (p –> q) ! b ;*

semántica: Se utilizan cláusulas de redenominación por prefijo en sentencias de visibilidad para expresar cambio de prefijo en cadenas de nombre prefijadas que son otorgadas o tomadas. (Dado que las cláusulas de redenominación por prefijo pueden utilizarse sin cambios de prefijo – cuando el *prefijo antiguo* y el *nuevo* son vacíos – se toman como la base semántica para sentencias de visibilidad.)

propiedades estáticas: Una *cláusula de redenominación por prefijo* tiene asociados uno o dos dominios de **origen**, que son los dominios de **origen** de la *sentencia de visibilidad* en la que está escrita.

Una *cláusula de redenominación por prefijo* tiene asociados uno o dos dominios de **destino**, que son los dominios de **destino** de la *cláusula de visibilidad* en la que está escrita.

Un *posfijo* tiene asociado un conjunto de *cadenas de nombre*, que es el conjunto de *cadenas de nombre* asociado a su *posfijo de toma* o el conjunto de *cadenas de nombre* asociado a su *posfijo de otorgamiento*. Estas *cadenas de nombre* son las *cadenas de nombre* posfijo de la cláusula de *redenominación por prefijo*.

Una *cláusula de redenominación por prefijo* tiene asociados un conjunto de *cadenas de nombre* **antiguas** y un conjunto de *cadenas de nombre* **nuevas**. Cada *cadena de nombre posfijo* asociada a la *cláusula de redenominación por prefijo* da una *cadena de nombre* **antigua** y una *cadena de nombre* **nueva** asociadas a la *cláusula de redenominación por prefijo*, como sigue: la *cadena de nombre* **nueva** se obtiene prefijando la *cadena de nombre* posfijo con el *prefijo nuevo*; la *cadena de nombre* **antigua** se obtiene prefijando la *cadena de nombre* posfijo con el *prefijo antiguo*.

Cuando una *cadena de nombre* **nueva** y una *cadena de nombre* **antigua** se obtienen a partir de la misma *cadena de nombre* posfijo, se dice que la *cadena de nombre* **antigua** es la fuente de la *cadena de nombre* **nueva**.

reglas de visibilidad: Las *cadenas de nombre* **nuevas** asociadas a una *cláusula de redenominación por prefijo* son **visibles** en sus dominios de **destino**, y están **directamente enlazadas** en esos dominios a sus fuentes en los dominios de **origen**. Si la *cláusula de redenominación por prefijo* es parte de una *sentencia de toma* (*otorgamiento*), esas *cadenas de nombre* se toman (otorgan) en su dominio (dominios) de **destino**.

Se dice que una *cadena de nombre* NS es **tomable** por el modulión M directamente encerrado en el dominio R si y sólo si es **visible** en R y no está **enlazada** en R a ninguna *cadena de nombre* en el dominio de M ni **directamente enlazada** a la *ocurrencia de definición* de una *cadena de nombre* predefinida.

Se dice que una *cadena de nombre* NS es **otorgable** por el modulión M directamente encerrado en el dominio R si y sólo si es **visible** en el dominio de M y no está **enlazada** en el mismo a ninguna *cadena de nombre* en R ni **directamente enlazada** en el mismo a la *ocurrencia de definición* de una *cadena de nombre* predefinida.

condiciones estáticas: Si una *cláusula de redenominación por prefijo* está en una *sentencia de toma* directamente encerrada en el dominio de modulión M, cada una de sus *cadenas de nombre* **antiguas** debe ser:

- **ligada** en el dominio R que encierra directamente al dominio de M; y
- **tomable** por M.

Si una *cláusula de redenominación por prefijo* está en una *sentencia de otorgamiento* directamente encerrada en el dominio del modulión M, cada una de sus *cadenas de nombre* **antiguas** debe ser:

- **ligada** en el dominio de M, y
- **otorgable** por M.

Una cláusula de *redenominación por prefijo* que aparece en una *sentencia de otorgamiento* (*toma*) debe tener un *posfijo* que sea un *posfijo* de *otorgamiento* (*toma*).

ejemplo:

25.35 (stack ! int –> stack) ! **ALL** (1.1)

12.2.3.4 Sentencia de otorgamiento

sintaxis:

<*sentencia de otorgamiento*> ::= *(1)*
GRANT <*cláusula de redenominación por prefijo*>
{ , <*cláusula de redenominación por prefijo*> }* ; *(1.1)*
| **GRANT** <*ventana de otorgamiento*> [<*cláusula de prefijo*>] [<*cláusula amigo*>]; *(1.2)*

<*ventana de otorgamiento*> ::= *(2)*
<*posfijo de otorgamiento*> { , <*posfijo de otorgamiento*> }* *(2.1)*

<*posfijo de otorgamiento*> ::= *(3)*
<*cadena de nombre*> [(<*lista de parámetros formales*>) [[**RETURNS**]
(<*espec de resultado*>)]] *(3.1)*
| <*cadena de nombre de neomodo*> <*cláusula de prohibición*> *(3.2)*
| [<*prefijo*> !] **ALL** *(3.3)*

<*cláusula de prefijo*> ::= *(4)*
PREFIXED [<*prefijo*>] *(4.1)*

<cláusula de prohibición> ::= *(5)*
FORBID { *<lista de nombres de prohibición>* | **ALL** } *(5.1)*

<lista de nombres de prohibición> ::= *(6)*
(*<nombre de campo>* { , *<nombre de campo>* }*) *(6.1)*

<cláusula amigo> ::= *(7)*
TO *<lista de cláusulas amigo>* *(7.1)*

<lista de nombres amigo> ::= *(8)*
<nombre amigo> { , *<nombre amigo>*}* *(8.1)*

<nombre amigo> ::= *(9)*
<nombre de modulión o de modo moreta>
[! *<nombre de procedimiento o proceso amigo>*] *(9.1)*

<nombre de modo modulión o moreta> ::= *(10)*
<nombre de modulión> *(10.1)*
| *<nombre de modo moreta>* *(10.2)*

<nombre de procedimiento o proceso amigo> ::= *(11)*
<nombre de procedimiento> [(*<lista de parámetros>*)
[[**RETURNS**] (*<espec de resultado>*)]] *(11.1)*
| *<nombre de proceso>* *(11.2)*

semántica: Las sentencias de otorgamiento son un medio de extender la visibilidad de cadenas de nombres que están en un dominio de modulión a los dominios que lo circundan directamente. **FORBID** puede especificarse solamente para nombres de **neomodo** que son modos estructura. Esto significa que todas las localizaciones y valores de ese modo tienen campos que pueden ser seleccionados solamente dentro del modulión otorgante, y no fuera.

Se aplican las siguientes reglas de visibilidad:

- Si la *sentencia de otorgamiento* contiene *cláusula*(*s*) *de redenominación por prefijo*, la *sentencia de otorgamiento* tiene el efecto de su(*s*) *cláusula*(*s*) *de redenominación por prefijo* (véase 12.2.3.3).
- Si la *sentencia de otorgamiento* contiene *ventanas de otorgamiento*, es una notación abreviada para un conjunto de *sentencias de otorgamiento* con *cláusulas de redenominación por prefijo* construidas como sigue:
  - Para cada *posfijo de otorgamiento* en la *ventana de otorgamiento*, hay una *sentencia de otorgamiento* correspondiente.
  - El *prefijo antiguo* en su *cláusula de redenominación por prefijo* es vacío.
  - El *prefijo nuevo* en su *cláusula de redenominación por prefijo* es el *prefijo* asociado a la *cláusula de prefijo* en la *sentencia de otorgamiento*, o es vacío si no hay cláusula de prefijo en la *sentencia de otorgamiento* original.
  - El *posfijo* en la *cláusula de redenominación* por prefijo es el *posfijo* correspondiente en la *ventana de otorgamiento*.
- La notación **FORBID ALL** es una notación abreviada que prohíbe todos los *nombres de campo* del nombre de **neomodo** (véase 12.2.5).
- Si la *cláusula de redenominación por prefijo* en una *sentencia de otorgamiento* tiene un *posfijo de otorgamiento* que contiene un *prefijo* y **ALL**, entonces es de la forma:

  (OP–> NP) ! P ! **ALL**

  donde *OP* y *NP* son respectivamente el *prefijo antiguo* y el *prefijo nuevo*, posiblemente vacíos, y *P* es el *prefijo* en el *posfijo de otorgamiento*. La *cláusula de redenominación por prefijo* es entonces equivalente a una cláusula de la forma:

  (*OP ! P –> NP ! P*) ! **ALL**
- Si se da una cláusula amigo, la visibilidad de los objetos GRANTed se extiende solamente a los grupos que están mencionados en la *lista de nombres amigo*.

propiedades estáticas: Una *cláusula de prefijo* tiene asociado un *prefijo*, definido como sigue:

- Si la *cláusula de prefijo* contiene un *prefijo*, entonces ese *prefijo* está asociado.
- En otro caso, el *prefijo* asociado es un *prefijo simple* cuya *cadena de nombre* se determina como sigue:
  - Si el dominio que circunda directamente al *prefijo* es un *módulo* o *región*, la *cadena de nombre* es la misma que la del nombre de **módulo** o el nombre de **región** de ese modulión.
  - Si el dominio que circunda directamente al *prefijo* es una *espec de región* o *espec de módulo*, la cadena de nombre es la cadena de nombre que precede a **SPEC**.

Un *posfijo de otorgamiento* tiene asociado un conjunto de *cadenas de nombre*, definido como sigue:

- Si es una *cadena de nombre*, o contiene una *cadena de nombre de neomodo*, es el conjunto que contiene solamente esa *cadena de nombre*.
- En otro caso, designando por *OP* el *prefijo antiguo* (posiblemente vacío) de la *cláusula de redenominación por prefijo* en la que está situado el *posfijo de otorgamiento*, el conjunto contiene todas las *cadenas de nombre* de la forma *OP ! N* (es decir, obtenidas prefijando *N* con *OP*) para cualquier *cadena de nombre N* tal que *OP ! N* es **visible** en el dominio del modulión en que está situado el *posfijo de otorgamiento* y es **otorgable** por este modulión.

condiciones estáticas: La *cadena de nombre de neomodo con cláusula de prohibición* debe ser **visible** en el dominio *R* del modulión en que está situada la *sentencia de otorgamiento*. La *cadena de nombre de neomodo* debe estar **ligada** en *R* a la *ocurrencia de definición* de un neomodo, que debe ser un modo estructura, y cada *nombre de campo* en la *lista de nombres de prohibición* debe ser un nombre de **campo** de ese modo. La *ocurrencia de definición* de neomodo debe estar directamente encerrada en *R*. Todos los *nombres de campo* en una *lista de nombres de prohibición* deben tener cadenas de nombre diferentes.

Si la *sentencia de otorgamiento* está situada en el dominio de una *región* o *región de espec*, no debe otorgar una *cadena de nombre* que está **ligada** en ese dominio a la *ocurrencia de definición* de:

- un nombre de **localización**, o
- un nombre de **identidad-loc**, en el cual la localización en su declaración es **intrarregional**; o
- un nombre de **sinónimo** cuyo valor es **intrarregional**.

La *cláusula de redenominación por prefijo* en una *sentencia de otorgamiento* debe tener un *posfijo de otorgamiento*.

Si una *sentencia de otorgamiento* contiene una *cláusula de prefijo* que no contiene un *prefijo*, entonces su modulión directamente circundante no debe ser un *contexto* y:

- si su modulión directamente circundante es un *módulo* o *región*, entonces debe ser denominado (es decir, debe ser encabezado por una *ocurrencia de definición* seguida de un signo dos puntos);
- si su modulión directamente circundante es una *espec de módulo* o una *espec de región*, debe ser encabezado por una *cadena de nombre simple*.

Un nombre N contenido en una *lista de nombres amigo* de una *sentencia de otorgamiento*, que está colocada inmediatamente en el dominio de un grupo G, debe estar definida inmediatamente en el dominio del grupo que circunda directamente a G.

Si la sentencia de otorgamiento aparece inmediatamente dentro de una especificación moreta, no debe producirse el prefijado.

ejemplos:

25.7 **GRANT** (–> *stack* ! *char*) ! **ALL**; *(1.1)*

6.44 *gregorian_date*, *julian_day_number* *(2.1)*

12.2.3.5 Sentencia de toma

sintaxis:

<sentencia de toma> ::= *(1)*
SEIZE *<cláusula de redenominación por prefijo>*
{ ,*<cláusula de redenominación por prefijo>* }* ; *(1.1)*
| **SEIZE** *<ventana de toma>* [*<cláusula de prefijo>*] ; *(1.2)*

<ventana de toma> ::= *(2)*
<posfijo de toma> { , *<posfijo de toma>* }* *(2.1)*

<posfijo de toma> ::= *(3)*
<cadena de nombre> [(*<lista de parámetros formales>*)
[[**RETURNS**] (*<espec de resultado>*)]] *(3.1)*
| [*<prefijo>* !] **ALL** *(3.2)*

semántica: Las sentencias de toma son un medio de extender la visibilidad de cadenas de nombre en dominios de grupo para llevarla a los dominios de moduliones directamente encerrados.

Se aplican las siguientes reglas de visibilidad:

- Si la *sentencia de toma* contiene *cláusula(s) de redenominación por prefijo*, la *sentencia de toma* tiene el efecto de su(s) *cláusula(s) de redenominación por prefijo* (véase 12.2.3.3).

- Si la *sentencia de toma* contiene una *ventana de toma*, es una notación abreviada para un conjunto de *sentencias de toma* con *cláusulas de redenominación por prefijo* construidas como sigue:
  - Para cada *posfijo de toma* en la *ventana de toma*, hay una *sentencia de toma* correspondiente.
  - El *prefijo antiguo* en la *cláusula de redenominación por prefijo* es el prefijo asociado a la *cláusula de prefijo* en la sentencia de toma, o es vacío si no hay *cláusula de prefijo* en la *sentencia de toma* original.
  - El *prefijo nuevo* en su *cláusula de redenominación por prefijo* es vacío.
  - El *posfijo* en su *cláusula de redenominación por prefijo* es el *posfijo* correspondiente de la *ventana de toma*.
- Si una *cláusula de redenominación por prefijo* en una *sentencia de toma* tiene un *posfijo de toma* que contiene un *prefijo* y **ALL**, entonces es de la forma:

  (OP –> NP) ! P ! **ALL**

  donde *OP* y *NP* son respectivamente el *prefijo antiguo* y el *prefijo nuevo*, posiblemente vacíos, y *P* es el *prefijo* en el *posfijo de toma*. La *cláusula de redenominación por prefijo* es entonces equivalente a una cláusula de la forma:

  (OP ! P –> NP ! P) ! **ALL**

propiedades estáticas: Un *posfijo de toma* tiene asociado un conjunto de *cadenas de nombre*, definido como sigue:

- Si el *posfijo de toma* es una *cadena de nombre*, el conjunto contiene solamente la *cadena de nombre*.
- En otro caso, si el *posfijo de toma* es **ALL**, designando por *OP* el *prefijo antiguo* (posiblemente vacío) de la *cláusula de redenominación por prefijo* de la cual forma parte el *posfijo de toma*, el conjunto contiene todas las *cadenas de nombre* de la forma *OP ! S*, para cualquier *cadena de nombre S* tal que:
  - *OP ! S* es **visible** en el dominio que circunda directamente al modulión en el que está situada la *sentencia de toma;* y
  - es **tomable** por este modulión; y
  - está **ligada** a una **cuasi** *ocurrencia de definición* si este modulión tiene un *contexto* que lo precede.

condiciones estáticas: La *cláusula de redenominación por prefijo* en una *sentencia de toma* debe tener un *posfijo de toma*.

Si una *sentencia de toma* contiene una *cláusula de prefijo* que no contiene un *prefijo*, su modulión directamente circundante no debe ser un *contexto*, y

- si su modulión directamente circundante es un *módulo o región*, debe ser denominado (es decir, debe ir encabezado por una *ocurrencia de definición* seguida de un signo dos puntos);
- si su modulión directamente circundante es una *espec de módulo* o una *espec de región*, debe ir encabezado por una *cadena de nombre simple*.

ejemplos:

25.35 SEIZE (*stack ! int –> stack*) *!* ALL; *(1.1)*

12.2.4 Visibilidad de nombres de elemento de conjunto

Un *nombre de elemento de conjunto* sólo puede ocurrir en el contexto de un *literal de conjunto*.

Si se especifica un *nombre de modo conjunto* en el *literal de conjunto*, la *cadena de nombre* de un *nombre de elemento de conjunto* puede estar l**igada** a una *ocurrencia de definición de nombre de elemento de conjunto* en el modo de la clase del *literal de conjunto*.

En otro caso, un *nombre de modo conjunto*, y entonces la *cadena de nombre* puede estar **ligada** a una *ocurrencia de definición de nombre de elemento de conjunto* si no es **visible** en el dominio en el que el *literal de conjunto* está colocado.

12.2.5 Visibilidad de nombres de campo

Los *nombres de campo* sólo pueden aparecer en los siguientes contextos:

- *campos de estructura* y *campos de estructura valor*;
- *tuplas de estructura etiquetada*;
- *cláusulas de prohibición* en *campos de otorgamiento*.

Obsérvese que un *nombre de campo* no puede ocurrir en un *posfijo de otorgamiento* ni en un *posfijo de toma*.

En cada uno de estos casos, la *cadena de nombre* del *nombre de campo* puede estar **ligada** a una *ocurrencia de definición de nombre de campo* en el modo M o en el modo **definidor** de M, obtenido como sigue:

- M es el modo de la *localización estructura* o *valor primitivo estructura* (**fuerte)**.
- M es el modo de la *tupla de estructura*.
- M es el modo de la *ocurrencia de definición* a la cual está **ligada** la *cadena de nombre de neomodo* en el dominio en el que está situada la *cláusula de prohibición*.

Sin embargo, si la **novedad** de M es una *ocurrencia de definición* que define un nombre de **neomodo** que ha sido otorgado por una *sentencia de otorgamiento* en un modulión como *posfijo de otorgamiento* con una *cláusula de prohibición*, entonces los nombres de campo mencionados en la lista de nombres a prohibir son solamente **visibles**:

- en el grupo del modulión de otorgamiento;
- si la **novedad** de M está **ligada por novedad** a una **cuasi novedad** N, en el grupo del dominio en que N está directamente encerrado;
- si el modulión es una *espec de módulo* o una *espec de región*, en el dominio del modulión **correspondiente**.

Fuera de estos dominios los *nombres de campo* mencionados en la *lista de nombres a prohibir* son **invisibles** y no pueden utilizarse.

12.2.6 Dependencia de localizaciones

Un ejemplar LI de una localización declarada directamente L depende de la ejecución de su grupo inmediatamente circundante que creó LI.

ejemplo:

```
SYNMODE TM = TASK SPEC ....
SYNMODE MM = MODULE SPEC
             DCL T1 TM;
END MM;
DCL M1 MM;
```

El ejemplar vigente M1-I o M1 contiene un ejemplar M1-I.T1 de MM.T1. M1-I.T1 ha sido creado durante la ejecución de "DCL M1 MM". Por tanto, M1-I.T1 depende de M1-I.

Dependencia de localizaciones en la memoria heap: *GETSTACK* y *ALLOCATE* crean una nueva localización L y entregan un valor de referencia R o L. Hay dos casos:

a) se sabe que el modo de R es RM. En este caso L depende del creador del ejemplar pertinente de RM;

b) el modo de R no es conocido (IF ALLOCATE(...) = ALLOCATE(...)...). En este caso L depende del creador del ejemplar pertinente de LM, siendo LM es el modo de L.

Una localización Lc que es un subcomponente de una localización L depende de L.

12.3 Selección de caso

sintaxis:

```
<especificación de etiqueta de caso> ::=                                (1)
        <lista de etiquetas de caso> { , <lista de etiquetas de caso> }*  (1.1)

<lista de etiquetas de caso> ::=                                        (2)
        ( etiqueta de caso> { , <etiqueta de caso> }* )                 (2.1)
      | <irrelevante >                                                  (2.2)

<etiqueta de caso> ::=                                                  (3)
        <expresión literal discreta>                                    (3.1)
      | <intervalo literal>                                             (3.2)
      | <nombre de modo discreto>                                       (3.3)
      | ELSE                                                            (3.4)

< irrelevante> ::=                                                      (4)
        (*)                                                             (4.1)
```

semántica: La selección de caso es un modo de seleccionar una alternativa entre las de una lista de alternativas. Se basa en una lista especificada de valores de selector. La selección de caso puede aplicarse a:

- campos de alternativa (véase 3.13.4), en cuyo caso se selecciona una lista de campos variables,
- tuplas de matriz etiquetada (véase 5.2.5), en cuyo caso se selecciona un valor elemento de matriz,

- expresiones condicionales (véase 5.3.2), en cuyo caso se selecciona una expresión,
- acción de caso (véase 6.4), en cuyo caso se selecciona una lista de sentencias de acción.

En las situaciones primera, tercera y cuarta, se etiqueta cada alternativa con una especificación de etiqueta de caso; en la tupla de matriz etiquetada, se etiqueta cada valor con una lista de etiquetas de caso. Para facilidad de descripción, la lista de etiquetas de caso de la tupla de matriz etiquetada se considerará aquí como una especificación de etiqueta de caso con sólo una ocurrencia de lista de etiquetas de caso.

La selección de caso selecciona aquella alternativa que está etiquetada por la especificación de la etiqueta de caso que concuerda con la lista de valores de selector. (El número de valores de selector será siempre el mismo que el número de ocurrencias de lista de etiquetas de caso en la especificación de etiqueta de caso.) Se dice que una lista de valores concuerda con una especificación de etiqueta de caso si y sólo si cada valor concuerda con la lista de etiquetas de caso correspondiente (en posición) en la especificación de etiqueta de caso.

Se dice que un valor concuerda con una lista de etiquetas de caso si y sólo si:

- la lista de etiquetas de caso consta de etiquetas de caso y el valor es uno de los valores indicados explícitamente por una de las etiquetas de caso o está implícitamente indicado en el caso de **ELSE**;
- la lista de etiquetas de caso consiste en *irrelevante*.

Los valores indicados explícitamente por una etiqueta de caso son los valores entregados por cualquier *expresión literal discreta* o definidos por un *intervalo literal* o *nombre de modo discreto*. Los valores indicados implícitamente por **ELSE** son todos los posibles valores de selector no indicados explícitamente por ninguna lista de etiquetas de caso asociada (es decir, perteneciente al mismo valor de selector) en alguna especificación de etiqueta de caso.

propiedades estáticas:

- Un *campo de alternativa* con *especificación de etiqueta de caso*, una *tupla de matriz etiquetada*, una *expresión condicional* o una *acción de caso* tiene asociada una lista de especificaciones de etiqueta de caso, formada tomando la *especificación de etiqueta de caso* que precede a cada *alternativa variable*, *valor* o *alternativa de caso*, respectivamente.
- Una *etiqueta de caso* tiene asociada una clase que es, si se trata de una *expresión literal discreta*, la clase de la *expresión literal discreta*; si es un *intervalo literal*, la **clase resultante** de las clases de cada *expresión literal discreta* en el *intervalo literal*; si es un *nombre de modo discreto*, la **clase resultante** de la clase M-valuada, donde M es un *nombre de modo discreto*; si es **ELSE**, la clase **general**.
- Una *lista de etiquetas de caso* tiene asociada una clase que es, si se trata de *irrelevante*, la clase **general**; en otro caso, es la **clase resultante** de las clases de cada *etiqueta de caso*.
- Una *especificación de etiqueta de caso* tiene asociada una lista de clases, que son las clases de las listas de etiquetas de caso.
- Una lista de especificaciones de etiquetas de caso tiene asociada una **lista resultante de clases**. Esta **lista resultante de clases** se forma determinando, para cada posición de la lista, la **clase resultante** de todas las clases que tienen esta posición.

Una lista de especificaciones de etiquetas de caso está **completa** si y sólo si para todas las listas de posibles valores de selector está presente una especificación de etiqueta de caso que concuerda con la lista de valores de selector. El conjunto de todos los posibles valores de selector se determina según el contexto como sigue:

- Para un modo estructura **variable con marcadores**, es el conjunto de valores definidos por el modo del campo **marcador** correspondiente.
- Para un modo estructura **variable sin marcadores**, es el conjunto de valores definidos por el modo **raíz** de la **clase resultante** correspondiente (esta clase no es nunca la clase **general**, véase 3.13.4).
- Para una tupla de matriz, es el conjunto de valores definidos por el modo **índice** del modo de la tupla de matriz.
- Para una acción de caso con lista de intervalos, es el conjunto de valores definidos por el modo discreto correspondiente en la lista de intervalos.
- Para una acción de caso sin lista de intervalos, o una expresión condicional, es el conjunto de valores definidos por M, donde la clase del selector correspondiente es la clase de M-valuada o la clase M-derivada.

condiciones estáticas: Para cada *especificación de etiqueta de caso*, el número de ocurrencias *de lista de etiquetas de caso* debe ser igual.

Para cualesquiera dos ocurrencias de *especificación de etiqueta de caso*, sus listas de clases deben ser **compatibles**.

La lista de ocurrencias de *especificación de etiqueta de caso* debe ser **consistente**, es decir, que cada lista de posibles valores de selector concuerda, como máximo, con una especificación de etiqueta de caso.

Si el modo **raíz** de la clase de una *lista de etiquetas de caso* es un modo entero, debe existir un modo entero **predefinido** que contiene todos los valores entregados por cada etiqueta de caso.

ejemplos:

11.9 (*occupied*) *(2.1)*

11.58 (*rook*),(∗) *(1.1)*

8.26 (**ELSE**) *(2.1)*

12.4 Definición y resumen de las categorías semánticas

En esta subcláusula se ofrece una resumen de las categorías semánticas que se indican en la descripción de la sintaxis mediante una parte subrayada. Si estas categorías no se han definido en las subcláusulas apropiadas, se da la definición aquí; si se han definido, se hace referencia a la subcláusula adecuada.

12.4.1 Nombres

Nombres de modo

nombre de modo tiempo absoluto: *nombre* que es, por definición, un modo tiempo absoluto.

nombre de modo acceso: *nombre* que es, por definición, un modo acceso.

nombre de modo matriz: *nombre* que es, por definición, un modo matriz.

nombre de modo asociación: *nombre* que es, por definición, un modo asociación.

nombre de modo booleano: *nombre* que es, por definición, un modo booleano.

nombre de modo referencia ligada: *nombre* que es, por definición, un modo referencia ligada.

nombre de modo tampón: *nombre* que es, por definición, un modo tampón.

nombre de modo carácter: *nombre* que es, por definición, un modo carácter.

nombre de modo discreto: *nombre* que es, por definición, un modo discreto.

nombre de modo intervalo discreto: *nombre* que es, por definición, un modo intervalo discreto

nombre de modo duración: *nombre* que es, por definición, un modo duración.

nombre de modo evento: *nombre* que es, por definición, un modo evento.

nombre de modo coma flotante: *nombre* que es, por definición, un modo coma flotante.

nombre de modo intervalo coma flotante: *nombre* que es, por definición, un modo intervalo coma flotante.

nombre de modo referencia libre: *nombre* que es, por definición, un modo referencia libre.

nombre de modo moreta genérico: *nombre* que es, por definición, un modo moreta genérico.

nombre de modo interfaz: *nombre* que es, por definición, un modo interfaz.

nombre de modo ejemplar: *nombre* que es, por definición, un modo ejemplar.

nombre de modo entero: *nombre* que es, por definición, un modo entero.

nombre de modo: véase 3.2.1.

nombre de modo módulo: *nombre* que es, por definición, un modo módulo.

nombre de modo modulión o moreta: *nombre* que es, por definición, un modo modulión o un modo moreta.

nombre de modo moreta: *nombre* que es, por definición, un modo moreta.

nombre de modo matriz parametrizado: *nombre* que es, por definición, un modo matriz **parametrizado**.

nombre de modo cadena parametrizado: *nombre* que es, por definición, un modo cadena **parametrizado**.

nombre de modo estructura parametrizado: *nombre* que es, por definición, un modo estructura **parametrizado**.

nombre de modo conjuntista: *nombre* que es, por definición, un modo conjuntista.

nombre de modo procedimiento: *nombre* que es, por definición, un modo procedimiento.

nombre de modo región:	*nombre* que es, por definición, un modo región.
nombre de modo descriptor:	*nombre* que es, por definición, un modo descriptor.
nombre de modo conjunto:	*nombre* que es, por definición, un modo conjunto.
nombre de modo cadena:	*nombre* que es, por definición, un modo cadena.
nombre de modo estructura:	*nombre* que es, por definición, un modo estructura.
nombre de modo tarea:	*nombre* que es, por definición, un modo tarea.
nombre de modo texto:	*nombre* que es, por definición, un modo texto.
nombre de modo estructura variable:	*nombre* que es, por definición, un modo estructura **variable**.

Nombres de acceso

nombre de localización:	véase 4.1.2.
nombre de localización hacer-con:	véase 6.5.4.
nombre de enumeración de localización:	véase 6.5.2.
nombre de identidad-loc:	véase 4.1.3.

Nombres de valor

nombre de literal booleano:	véase 5.2.4.4.
nombre de literal de vacío:	véase 5.2.4.7.
nombre de sinónimo:	véase 5.1.
nombre de valor hacer-con:	véase 6.5.4.
nombre de enumeración de valor:	véase 6.5.2.
nombre de valor a recibir:	véanse 6.19.2, 6.19.3.

Nombres varios

nombre de rutina incorporada:	todo nombre definido en CHILL, o en la implementación, que designa una rutina incorporada.
nombre de procedimiento o proceso amigo:	véase 12.2.3.4.
nombre de procedimiento general:	*nombre de procedimiento* cuya generalidad es **general**.
nombre de módulo genérico:	véase 10.11
nombre de procedimiento genérico:	véase 10.11
nombre de proceso genérico:	véase 10.11
nombre de región genérico:	véase 10.11
nombre de etiqueta:	véanse 6.1, 10.6.
nombre de modulión:	véase 12.2.3.4
cadena de nombre de neomodo:	una *cadena de nombre* **ligada** a la *ocurrencia de definición* de un nombre de **neomodo**.
nombre no reservado:	*nombre* que no es ninguno de los nombres reservados mencionados en el apéndice III.
nombre de procedimiento:	véase 10.4.
nombre de proceso:	véase 10.5.
nombre de elemento de conjunto:	véase 3.4.5.
nombre de señal:	véase 11.5.
nombre de campo marcador:	véase 3.13.4.
nombre de sinónimo indefinido:	véase 5.1.

12.4.2 Localizaciones

localización acceso:	*localización* con un modo acceso.
localización matriz:	*localización* con un modo matriz.
localización asociación:	*localización* con un modo asociación.
localización tampón:	*localización* con un modo tampón.
localización cadena de caracteres:	*localización* con un modo cadena de **caracteres**.
localización discreta:	*localización* con un modo discreto.
localización evento:	*localización* con un modo evento.
localización coma flotante:	*localización* con un modo coma flotante.
localización ejemplar:	*localización* con un modo ejemplar.
localización entero:	*localización* con un modo entero.
localización moreta:	*localización* con un modo moreta.
localización modo estático:	*localización* con un modo estático.
localización cadena:	*localización* con un modo cadena.
localización estructura:	*localización* con un modo estructura.
localización texto:	*localización* con un modo texto.

12.4.3 Expresiones y valores

valor primitivo de tiempo absoluto:	*valor primitivo* de clase **compatible** con un modo tiempo absoluto.
expresión matriz:	*expresión* de clase **compatible** con un modo matriz.
valor primitivo de matriz:	*valor primitivo* de clase **compatible** con un modo matriz.
expresión booleana:	*expresión* de clase **compatible** con un modo booleano.
valor primitivo de localización moreta de referencia ligada:	véase 6.7.
valor primitivo de referencia ligada:	*valor primitivo* de clase **compatible** con un modo de referencia ligada.
expresión cadena de caracteres:	expresión de clase **compatible** con un modo cadena de **caracteres**.
valor constante:	*valor* que es **constante**.
expresión discreta:	*expresión* de clase **compatible** con un modo discreto.
expresión literal discreta:	*expresión discreta* que es **literal**.
valor primitivo de duración:	*valor primitivo* de clase **compatible** con un modo duración.
expresión de coma flotante:	*expresión* cuya clase es **compatible** con un modo coma flotante.
expresión literal de coma flotante:	*expresión de coma flotante* que es **literal.**
valor primitivo de referencia libre:	*valor primitivo* de clase **compatible** con un modo de referencia libre.
valor primitivo de ejemplar:	*valor primitivo* de clase **compatible** con un modo ejemplar.
expresión entera:	*expresión* de clase **compatible** con un modo entero.
expresión literal entera:	*expresión entera* que es **literal**.

expresión literal:	*expresión* que es **literal.**
expresión conjuntista:	*expresión* de clase **compatible** con un modo conjuntista.
valor primitivo de procedimiento:	*valor primitivo* de clase **compatible** con un modo procedimiento.
valor primitivo de referencia:	*valor primitivo* de clase **compatible** con un modo referencia ligada, un modo referencia libre o un modo descriptor.
valor primitivo de descriptor:	*valor primitivo* de clase **compatible** con un modo descriptor.
expresión cadena:	*expresión* de clase **compatible** con un modo cadena.
valor primitivo de cadena:	*valor primitivo* de clase **compatible** con un modo cadena.
valor primitivo estructura:	*valor primitivo* de clase **compatible** con un modo estructura.

12.4.4 Categorías semánticas varias

modo matriz:	*modo* en el que el *modo compuesto* es un *modo matriz*.
lista de parámetros efectivos constructor:	véase 4.1.2.
modo discreto:	*modo* en el que el *modo no compuesto* es un *modo discreto*.
sentencia de definición de procedimiento guardado en línea:	véase 10.4.
llamada a rutina incorporada que entrega una localización:	véase 6.7.
llamada a procedimiento que entrega una localización:	véase 6.7.
llamada a procedimiento componente moreta:	véase 2.7.
sentencia de declaración moreta:	véase 3.15.
sentencia de definición de neomodo moreta:	véase 3.15.
sentencia de definición de sínmodo moreta:	véase 3.15.
carácter no tanto por ciento:	*carácter* que no es el de tanto por ciento (%).
carácter no reservado:	*carácter* que no es comillas (") ni circunflejo (^).
carácter ancho no reservado:	*carácter ancho* que no es comillas (") ni circunflejo (^).
carácter no especial:	*carácter* que no es circunflejo (^) ni paréntesis izquierdo (*(*).
sentencia de definición de procedimiento guardado simple:	véase 10.4.
sentencia de firma de procedimiento guardado simple:	véase 10.4.
modo cadena:	*modo* en el que el *modo compuesto* es un *modo cadena*.
llamada a rutina incorporada que entrega un valor:	véase 6.7.
llamada a procedimiento que entrega un valor:	véase 6.7.
modo estructura variable:	*modo* en el que el *modo no compuesto* es un modo estructura **variable**.

13 Opciones de implementación

13.1 Rutinas incorporadas definidas por la implementación

semántica0 Una implementación puede proporcionar un conjunto de rutinas incorporadas definidas por la implementación, además del conjunto de rutinas incorporadas definidas por el lenguaje.

El mecanismo de paso de parámetros se define por la implementación.

nombres predefinidos: El nombre de una rutina incorporada definida por la implementación es predefinido como un nombre de **rutina incorporada**.

propiedades estáticas: Un nombre de **rutina incorporada** puede tener asociado un conjunto de nombres de excepción definidos por la implementación. Una *llamada a rutina incorporada* es una *llamada a rutina incorporada* **que entrega un valor (localización)** si y sólo si la implementación especifica que para una selección dada de propiedades estáticas de los parámetros y el contexto estático dado de la llamada, la llamada a rutina incorporada entrega un valor (localización).

La implementación especifica también la **regionalidad** del valor (localización).

13.2 Modos enteros definidos por la implementación

Una implementación define el **límite superior** y el **límite inferior** del modo entero *INT*. Una implementación puede definir modos enteros diferentes de los definidos por *INT*; por ejemplo, enteros cortos, enteros largos, enteros sin signo. Estos modos enteros deben ser denotados por nombres de **modo** entero definidos por la implementación. Se considera que estos nombres son nombres de **neomodo**, **similares** a *INT*. Sus intervalos de valores se definen por la implementación. Estos modos enteros pueden definirse como modos **raíz** de clases apropiadas.

13.3 Modos coma flotante definidos por la implementación

Una implementación define el **límite superior** y el **límite inferior**, el **límite superior negativo** y el **límite inferior positivo**, y la **precisión** del modo coma flotante *FLOAT*. Una implementación puede definir modos coma flotante diferentes de los definidos por *FLOAT*; por ejemplo, coma flotante cortos, coma flotante largos. Estos modos coma flotante deben ser denotados por nombres de **modo** coma flotante definidos por la implementación. Se considera que estos nombres son nombres de **neomodo**, **similares** a *FLOAT*. Sus intervalos de valores, límites inferiores y precisión se definen por la implementación. Estos modos coma flotante pueden definirse como modos **raíz** de clases apropiadas.

13.4 Nombres de proceso definidos por la implementación

Una implementación puede definir un conjunto de nombres de **proceso** definidos por la implementación, es decir, nombres de **proceso** cuya definición no se especifica en CHILL. Se considera que la definición está situada en el dominio del proceso imaginario más externo, o en cualquier contexto. Se pueden iniciar procesos de este nombre, y se pueden manipular valores de ejemplar que denotan tales procesos.

13.5 Manejadores definidos por la implementación

Una implementación puede especificar que un manejador definido por la implementación se agregue a una definición de proceso; tal manejador puede manejar cualquier excepción.

13.6 Nombres de excepción definidos por la implementación

Una implementación puede definir un conjunto de nombres de excepción.

13.7 Otras características definidas por la implementación

- verificación estática de condiciones dinámicas (véase 2.1.2),
- *directiva de implementación* (véase 2.6),
- caso de cadenas de nombre simple **especial**,

- *nombre de referencia de texto* (véanse 2.7 y 10.10.1),
- **generalidad** por defecto (véase 10.4),
- conjunto de valores de modos duración (véase 3.12.2),
- conjunto de valores de modos tiempo absoluto (véase 3.12.3),
- **organización de elementos** por defecto (véase 3.13.3),
- comparación de valores de estructura **variable sin marcador** (véase 3.13.4),
- número de bits en una palabra (véase 3.13.5),
- ocupación mínima de bits (véase 3.13.5),
- (sub)localizaciones **referibles** adicionales (véase 4.2.1),
- semántica de un nombre de *localización hacer-con* y de un nombre de *valor hacer-con* que es un campo **variable** de una localización estructura **variable sin marcador** (véanse 4.2.2 y 5.2.3),
- semántica de campos **variables** de estructuras **variables sin marcador** (véanse 4.2.10, 5.2.14 y 6.2),
- semántica de *conversión de localización* (véase 4.2.13),
- semántica de *conversión de expresión* y condiciones adicionales (véase 5.2.11),
- *parámetros efectivos* adicionales en una *expresión arrancar* (véase 5.2.15),
- intervalos de valores para expresiones **literales** y **constantes** (véase 5.3.1),
- algoritmo de calendarización (o cronología) (véanse 6.15, 6.18.2, 6.18.3, 6.19.2, 6.19.3 y 11.2.1),
- liberación de espacio de almacenamiento en *TERMINATE* (véase 6.20.4),
- denotación para ficheros (véase 7.1),
- operaciones sobre asociaciones (véanse 7.1 y 7.2.1),
- asociaciones no exclusivas (véase 7.1),
- atributos adicionales de valores de asociación (véase 7.2.2),
- semántica de *parámetros asociar* (véase 7.4.2),
- excepción *ASSOCIATEFAIL* (véase 7.4.2),
- semántica de *parámetros modificar* (véase 7.4.5),
- excepciones *CREATEFAIL*, *DELETEFAIL* y *MODIFYFAIL* (véase 7.4.5),
- excepción *CONNECTFAIL* (véase 7.4.6),
- semántica de la lectura de registros que no son valores legales de acuerdo con el modo registro (véase 7.4.9),
- acciones **temporizables** adicionales (véase 9.2),
- excepción *TIMERFAIL* (véanse 9.3.1, 9.3.2 y 9.3.3),
- precisión de valores de duración (véanse 9.4.1 y 9.4.2),
- indicación de valor **constante** en *cuasi definiciones de sinónimo* (véase 10.10.3),
- **regionalidad** de rutinas incorporadas (véase 11.2.2).

Apéndice I

Juego de caracteres CHILL

El juego de caracteres CHILL es una ampliación del Alfabeto N.º 5 del CCITT, versión internacional de referencia, Recomendación V3. Para los valores cuyas representaciones son mayores que 127, no se define una representación gráfica.

La representación entera es el número binario formado por los bits b_8 a b_1, siendo b_1 el bit menos significativo.

	$b_7b_6b_5$	000	001	010	011	100	101	110	111
$b_4b_3b_2b_1$		0	1	2	3	4	5	6	7
0000	0	NUL	TC_7 (DLE)	SP	0	@	P	'	p
0001	1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0010	2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0011	3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0100	4	TC_4 (EOT)	DC_4	$	4	D	T	d	t
0101	5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0110	6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0111	7	BEL	TC_{10} (ETB)	′	7	G	W	g	w
1000	8	FE_0 (BS)	CAN	(	8	H	X	h	x
1001	9	FE_1 (HT)	EM	)	9	I	Y	i	y
1010	10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1011	11	FE_3 (VT)	ESC	+	;	K	[	k	{
1100	12	FE_4 (FF)	IS_4 (FS)	,	<	L	\	l	|
1101	13	FE_5 (CR)	IS_3 (GS)	–	=	M	]	m	}
1110	14	SO	IS_2 (RS)	.	>	N	^	n	~
1111	15	SI	IS_1 (US)	/	?	O	_	o	DEL

Apéndice II

Símbolos especiales

	Nombre	Utilización
;	punto y coma	terminador de sentencias, etc.
,	coma	separador en diversas construcciones
(	paréntesis izquierdo	paréntesis de apertura de diversas construcciones
)	paréntesis derecho	paréntesis de cierre de diversas construcciones
[	corchete izquierdo	corchete de apertura de una tupla
]	corchete derecho	corchete de cierre de una tupla
(:	corchete izquierdo de tupla	corchete de apertura de una tupla
:)	corchete derecho de tupla	corchete de cierre de una tupla
:	dos puntos	indicador de etiqueta, indicador de intervalo
.	punto	símbolo de selección de campo
:=	símbolo de asignación	asignación, inicialización
<	menor que	operador relacional
<=	menor o igual	operador relacional
=	igual a	operador relacional, asignación, inicialización, indicador de definición
/=	no igual	operador relacional
>=	mayor o igual	operador relacional
>	mayor que	operador relacional
+	más	operador de adición
–	menos	operador de sustracción
*	asterisco	operador de multiplicación, valor indefinido, valor no denominado, símbolo de indiferente
/	barra oblicua	operador de división
//	doble barra oblicua	operador de concatenación
–>	flecha	referenciación y desreferenciación, redenominación por prefijo
<>	diamante	comienzo o fin de una cláusula directiva
/*	apertura de comentario	corchete inicial de un comentario
*/	cierre de comentario	corchete final de un comentario
'	apóstrofo	símbolo de comienzo o fin en diversos literales
#	signo de número	conversión de localización y expresión
"	comillas	símbolo de comienzo o fin en literales de cadena de caracteres
!	operador de prefijación	prefijación de nombres
B'	calificación de literal	base binaria para literal
b'	calificación de literal	base binaria para literal
D'	calificación de literal	base decimal para literal
d'	calificación de literal	base decimal para literal
H'	calificación de literal	base hexadecimal para literal
h'	calificación de literal	base hexadecimal para literal
O'	calificación de literal	base octal para literal
o'	calificación de literal	base octal para literal
W'	calificación de literal	carácter ancho o literal de cadena de caracteres
w'	calificación de literal	carácter ancho o literal de cadena de caracteres
– –	fin de línea	delimitador de fin de línea de comentarios en línea

Apéndice III

Cadenas de nombre simple especiales

III.1 Cadenas de nombre simple reservadas

ABSTRACT
ACCESS
AFTER
ALL
AND
ANDIF
ANY
ANY_ASSIGN
ANY_DISCRETE
ANY_INT
ANY_REAL
ARRAY
ASSIGNABLE
ASSERT
AT
BASED_ON
BEGIN
BIN
BODY
BOOLS
BUFFER
BY
CASE
CAUSE
CHARS
CONSTR
CONTEXT
CONTINUE
CYCLE
DCL
DELAY
DESTR
DO
DOWN
DYNAMIC
ELSE
ELSIF
END
ESAC
EVENT
EVER
EXCEPTIONS
EXIT
FI
FINAL
FOR
FORBID
GENERAL
GENERIC
GOTO
GRANT
IF
IMPLEMENTS
IN
INCOMPLETE
INIT
INLINE
INOUT
INTERFACE
INVARIANT
LOC
MOD
MODE
MODULE
NEW
NEWMODE
NONREF
NOT_ASSIGNABLE
NOPACK
NOT
OD
OF
ON
OR
ORIF
OUT
PACK
POS
POST
POWERSET
PRE
PREFIXED
PRIORITY
PROC
PROCESS
RANGE
READ
RECEIVE
REF
REGION
REIMPLEMENT
REM
REMOTE
RESULT
RETURN
RETURNS
ROW
SEIZE
SELF
SEND
SET
SIGNAL
SIMPLE
SPEC
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE
TASK
TEXT
THEN
THIS
TIMEOUT
TO
UP
VARYING
WCHARS
WHILE
WITH
WTEXT
XOR

III.2 Cadenas de nombre simple predefinidas

ABS
ABSTIME
ALLOCATE
ARCCOS
ARCSIN
ARCTAN
ASSOCIATE
ASSOCIATION
BOOL
CARD
CHAR
CONNECT
COS
CREATE
DAYS
DELETE
DISCONNECT
DISSOCIATE
DURATION
EOLN
EXISTING
EXP
EXPIRED
FALSE
FIRST
FLOAT
GETASSOCIATION
GETSTACK
GETTEXTACCESS
GETTEXTINDEX
GETTEXTRECORD
GETUSAGE
HOURS
INDEXABLE
INSTANCE
INT
INTTIME
ISASSOCIATED
LAST
LENGTH
LN
LOG
LOWER
MAX
MILLISECS
MIN
MINUTES
MODIFY
NULL
NUM
OUTOFFILE
PRED
PTR
READABLE
READONLY
READRECORD
READTEXT
READWRITE
SAME
SECS
SEQUENCIBLE
SETTEXTACCESS
SETTEXTINDEX
SETTEXTRECORD
SIN
SIZE
SUCC
SQRT
TAN
TERMINATE
TIME
TRUE
UPPER
USAGE
VARIABLE
WAIT
WCHAR
WHERE
WRITEABLE
WRITEONLY
WRITERECORD
WRITETEXT

III.3 Nombres de excepción

ALLOCATEFAIL
ASSERTFAIL
ASSOCIATEFAIL
CONNECTFAIL
CREATEFAIL
DELAYFAIL
DELETEFAIL
EMPTY
INVFAIL
MODIFYFAIL
NOTASSOCIATED
NOTCONNECTED
OVERFLOW
POSTFAIL
PREFAIL
RANGEFAIL
READFAIL
SENDFAIL
SPACEFAIL
TAGFAIL
TEXTFAIL
THIS_FAIL
TIMERFAIL
UNDERFLOW
WRITEFAIL

Apéndice IV

Ejemplos de programas

IV.1 Operaciones sobre enteros

```
integer_operations:
MODULE

    add:
    PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
        RESULT i+j;
    END add;

    mult:
    PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
        RESULT i*j;
    END mult;

    GRANT add, mult;
    SYNMODE operand_mode=INT;
    GRANT operand_mode;
    SYN  neutral_for_add=0,
         neutral_for_mult=1;
    GRANT  neutral_for_add,
           neutral_for_mult;

END integer_operations;
```

IV.2 Las mismas operaciones sobre fracciones

```
fraction_operations:
MODULE
    NEWMODE fraction=STRUCT (num,denum INT);

    add:
    PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
        RETURN [f1.num*f2.denum+f2.num*f1.denum,f1.denum*f2.denum];
    END add;

    mult:
    PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
        RETURN [f1.num*f2.num,f2.denum*f1.denum];
    END mult;

    GRANT add, mult;
    SYNMODE operand_mode=fraction;
    GRANT operand_mode;
    SYN  neutral_for_add fraction=[ 0,1 ],
         neutral_for_mult fraction=[ 1,1 ];
    GRANT  neutral_for_add,
           neutral_for_mult;

END fraction_operations;
```

IV.3 Las mismas operaciones sobre números complejos

```
1       complex_operations:
2       MODULE
3           NEWMODE complex=STRUCT (re,im FLOAT);
4
5           add:
6           PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
7                   RETURN [c1.re+c2.re,c1.im+c2.im];
8           END add;
9
10          mult:
11          PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
12                  RETURN [c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re];
13          END mult;
14
15          GRANT add, mult;
16          SYNMODE operand_mode=complex;
17          GRANT operand_mode;
18          SYN   neutral_for_add=complex [ 0.0,0.0 ],
19                  neutral_for_mult=complex [ 1.0,0.0 ];
20          GRANT  neutral_for_add,
21                     neutral_for_mult;
22
23      END complex_operations;
```

IV.4 Aritmética de orden general

```
1       general_order_arithmetic:  /* from collected algorithms from CACM no. 93 */
2       MODULE
3           op:
4           PROC (a INT INOUT, b,c,order INT)
5                   EXCEPTIONS (wrong_input);
6               DCL d INT;
7               ASSERT b>0 AND c>0 AND order>0
8                   ON (ASSERTFAIL):
9                       CAUSE wrong_input;
10                  END;
11              CASE order OF
12                  (1):        a := b+c;
13                              RETURN;
14                  (2):        d := 0;
15                  (ELSE):     d := 1;
16              ESAC;
17              DO FOR i := 1 TO c;
18                  op (a,b,d,order–1);
19                  d := a;
20              OD;
21              RETURN;
22          END op;
23
24          GRANT op;
25
26      END general_order_arithmetic;
```

IV.5 Adición bit por bit y verificación del resultado

```
1       add_bit_by_bit:
2       MODULE
3           adder:
4           PROC (a STRUCT (a2,a1 BOOL) IN, b STRUCT (b2,b1 BOOL) IN)
5                   RETURNS (STRUCT (c4,c2,c1 BOOL));
```

```
6               DCL c STRUCT (c4,c2,c1 BOOL);
7               DCL k2,x,w,t,s,r BOOL;
8               DO WITH a,b,c;
9                   k2 := a1 AND b1;
10                  c1 := NOT k2 AND (a1 OR b1);
11                  x := a2 AND b2 AND k2;
12                  w := a2 OR b2 OR k2;
13                  t := b2 AND k2;
14                  s := a2 AND k2;
15                  r := a2 AND b2;
16                  c4 := r OR s OR t;
17                  c2 := x OR (w AND NOT c4);
18              OD;
19              RETURN c;
20          END adder;
21          GRANT adder;
22      END add_bit_by_bit;
23
24      exhaustive_checker:
25      MODULE
26          SEIZE adder;
27          SYNMODE res=ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
28          DCL r INT, results res;
29          r := 0;
30          DO FOR a2 IN BOOL;
31              DO FOR a1 IN BOOL;
32                  DO FOR b2 IN BOOL;
33                      DO FOR b1 IN BOOL;
34                          r+ := 1;
35                          results (r) := adder ([a2,a1], [b2,b1]);
36                      OD;
37                  OD;
38              OD;
39          OD;
40          ASSERT
41              results=res [  [FALSE,FALSE,FALSE ],[FALSE,FALSE,TRUE ],
42                             [FALSE,TRUE,FALSE ] ,[FALSE,TRUE,TRUE ],
43                             [FALSE,FALSE,TRUE ] ,[FALSE,TRUE,FALSE ],
44                             [FALSE,TRUE,TRUE ],[TRUE,FALSE,FALSE ],
45                             [FALSE,TRUE,FALSE ] ,[FALSE,TRUE,TRUE ],
46                             [TRUE,FALSE,FALSE ] ,[TRUE,FALSE,TRUE ],
47                             [FALSE,TRUE,TRUE ],[TRUE,FALSE,FALSE ],
48                             [TRUE,FALSE,TRUE ],[TRUE,TRUE,FALSE ]];
49      END exhaustive_checker;
```

IV.6 Operaciones con fechas

```
1       playing_with_dates:
2       MODULE /* from collected algorithms from CACM no. 199 */
3           SYNMODE  month=SET (jan,feb,mar,apr,may,jun,
4                       jul,aug,sep,oct,nov,dec);
5           NEWMODE date=STRUCT (day INT (1:31), mo month, year INT);
6
7           gregorian_date:
8           PROC (julian_day_number INT) RETURNS (date);
9               DCL  j INT:= julian_day_number,
10                   d,m,y INT;
11              j– := 1_721_119;
12              y := (4 * j – 1) / 146_097;
13              j := 4 * j – 1 – 146_097 * y;
14              d := j / 4;
15              j := (4 * d + 3) / 1_461;
```

```
16              d := 4 * d + 3 – 1_461 * j;
17              d := (d + 4) / 4;
18              m := (5 * d – 3) / 153;
19              d := 5 * d – 3 – 153 * m;
20              d := (d + 5) / 5;
21              y := 100 * y + j;
22              IF m<10  THEN     m + := 3;
23                       ELSE     m – := 9;
24                                y + := 1;
25              FI;
26              RETURN [d,month (m-1), y];
27          END gregorian_date;
28
29          julian_day_number:
30          PROC (d date) RETURNS (INT);
31              DCL c,y,m INT;
32              DO WITH d;
33                  m := NUM (mo)+1;
34                  IF m>2   THEN     m – := 3;
35                           ELSE     m + := 9;
36                                    year – := 1;
37                  FI;
38                  c := year/100;
39                  y := year-100*c;
40                  RETURN(146_097*c)/4+(1_461*y)/4
41                                +(153*m+2)/5+day+1_721_119;
42              OD;
43          END julian_day_number;
44          GRANT gregorian_date, julian_day_number;
45      END playing_with_dates;
46
47      test:
48      MODULE
49          SEIZE gregorian_date, julian_day_number;
50          ASSERT julian_day_number ([ 10,dec,1979 ])= julian_day_number
51                      (gregorian_date(julian_day_number([ 10,dec,1979 ])));
52      END test;
```

IV.7 Numerales romanos

```
1       Roman:
2       MODULE
3           SEIZE n,rn;
4           GRANT convert;
5           convert:
6           PROC () EXCEPTIONS (string_too_small);
7               DCL r INT:= 0;
8               DO WHILE n>=1_000;
9                   rn(r) := 'M';
10                  n – := 1_000;
11                  r + := 1;
12              OD;
13              IF n>500 THEN     rn(r) := 'D';
14                                n – := 500;
15                                r + := 1;
16              FI;
17              DO WHILE n>=100;
18                  rn(r) := 'C';
19                  n – := 100;
20                  r + := 1;
21              OD;
22              IF n>=50 THEN     rn(r) := 'L';
```

```
23                                  n – := 50;
24                                  r + := 1;
25              FI;
26              DO WHILE n>=10;
27                  rn(r) := 'X';
28                  n – := 10;
29                  r + := 1;
30              OD;
31              IF n>=5 THEN rn(r) := 'V';
32                           n – := 5;
33                           r + := 1;
34              FI;
35              DO WHILE n>=1;
36                  rn(r) := 'I';
37                  n – := 1;
38                  r + := 1;
39              OD;
40              RETURN;
41          END ON (RANGEFAIL):  DO FOR i := 0 TO UPPER (rn);
42                                   rn(i) := '.';
43                               OD;
44                               CAUSE string_too_small;
45          END convert;
46      END Roman;
47      test:
48      MODULE
49          SEIZE convert;
50          DCL n INT INIT:= 1979;
51          DCL rn CHARS (20) INIT:= (20)" ";
52          GRANT n,rn;
53          convert ();
54          ASSERT rn="MDCCCCLXXVIIII"//(6)" ";
55      END test;
```

IV.8 Cuenta de letras en una cadena de caracteres de longitud arbitraria

```
1       letter_count:
2       MODULE
3           SEIZE max;
4           DCL letter POWERSET CHAR INIT:= ['A' : 'Z'];
5           count:
6           PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT);
7               output := [(ELSE) : 0];
8               DO FOR i := 0 TO UPPER (input –>);
9                   IF input –> (i) IN letter
10                      THEN
11                          output (input –> (i)) + := 1;
12                  FI;
13              OD;
14          END count;
15          GRANT count;
16      END letter_count;
17      test:
18      MODULE
19          SYNMODE results=ARRAY ('A':'Z')INT;
20          DCL c CHARS (10) INIT:= "A-B<ZAA9K' ";
21          DCL output results;
22          SYN max=10_000;
23          GRANT max;
24          SEIZE count;
25          count (–> c,output);
26          ASSERT output=results [('A') : 3,('B','K','Z') : 1, (ELSE) : 0];
27      END test;
```

IV.9 Números primos

```
1       prime:
2       MODULE
3
4           SYN max = H'7FFF;
5           NEWMODE number_list =POWERSET INT (2:max);
6           SYN empty = number_list [ ];
7           DCL sieve number_list INIT:= [ 2:max ],
8               primes number_list INIT:= empty;
9           GRANT primes;
10          DO WHILE sieve/=empty;
11              primes OR:= [MIN (sieve)];
12              DO FOR j := MIN (sieve) BY MIN (sieve) TO max;
13                  sieve – := [j];
14              OD;
15          OD;
16      END prime;
```

IV.10 Implementación de pilas de dos formas distintas, transparentes para el usuario

```
1       stack: MODULE
2           NEWMODE element =STRUCT (a INT, b BOOL);
3           stacks_1:
4           MODULE
5               SEIZE element;
6               SYN max=10_000,min=1;
7               DCL   stack ARRAY (min : max) element,
8                     stackindex INT INIT:= min;
9
10              push:
11              PROC (e element) EXCEPTIONS (overflow);
12                  IF stackindex=max
13                      THEN CAUSE overflow;
14                  FI;
15                  stackindex + := 1;
16                  stack (stackindex) := e;
17                  RETURN;
18              END push;
19
20              pop:
21              PROC () EXCEPTIONS (underflow);
22                  IF stackindex=min
23                      THEN CAUSE underflow;
24                  FI;
25                  stackindex – := 1;
26                  RETURN;
27              END pop;
28
29              elem:
30              PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
31                  IF i<min OR i>max
32                      THEN CAUSE bounds;
33                  FI;
34                  RETURN stack (i);
35              END elem;
36
37              GRANT push,pop,elem;
38          END stacks_1;
39          stacks_2:
40          MODULE
```

```
41              SEIZE element;
42              NEWMODE cell=STRUCT (pred,succ REF cell,info element);
43              DCL p,last,first REF cell INIT:= NULL;
44
45              push:
46              PROC (e element) EXCEPTIONS (overflow);
47                  p := ALLOCATE (cell)  ON
48                                        (ALLOCATEFAIL) : CAUSE overflow;
49                  END;
50                  IF last=NULL
51                      THENfirst := p;
52                           last := p;
53                      ELSE last –>. succ := p;
54                           p –>. pred := last;
55                           last := p;
56                  FI;
57                  last –>. info := e;
58                  RETURN;
59              END push;
60
61              pop:
62              PROC () EXCEPTIONS (underflow);
63                  IF last=NULL
64                      THEN CAUSE underflow;
65                  FI;
66                  p := last;
67                  last := last –>. pred;
68                  IF last = NULL
69                      THEN first := NULL;
70                      ELSE last –>. succ := NULL;
71                  FI;
72                  TERMINATE(p);
73                  RETURN;
74              END pop;
75
76              elem:
77              PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
78                  IF first=NULL
79                      THEN CAUSE bounds;
80                  FI;
81                  p := first;
82                  DO FOR j := 2 TO i;
83                      IF p –>. succ=NULL
84                          THEN CAUSE bounds;
85                      FI;
86                      p := p –>. succ;
87                  OD;
88                  RETURN p –>. info;
89              END elem;
90
91              /* GRANT push,pop,elem; */
92          END stacks_2;
93      END stack;
```

IV.11 Fragmento para jugar al ajedrez

```
1       chess_fragments:
2       MODULE
3           NEWMODE piece=STRUCT (color SET (white,black),
4                                 kind SET (pawn,rook,knight,bishop,queen,king));
5           NEWMODE column=SET (a,b,c,d,e,f,g,h);
6           NEWMODE line=INT (1 : 8);
```

```
7       NEWMODE square=STRUCT   (status SET (occupied,free),
8                               CASE status OF
9                               (occupied) : p piece,
10                              (free) :
11                              ESAC);
12      NEWMODE board=ARRAY (line) ARRAY (column) square;
13      NEWMODE move=STRUCT (lin_1,lin_2 line,
14                           col_1,col_2 column);
15
16      initialize:
17      PROC (bd board INOUT);
18          bd := [  (1):   [  (a,h):    [.status: occupied, .p : [white,rook]],
19                             (b,g):    [.status: occupied, .p : [white,knight]],
20                             (c,f):    [.status: occupied, .p : [white,bishop]],
21                             (d):      [.status: occupied, .p : [white,queen]],
22                             (e):      [.status: occupied, .p : [white,king]]],
23                   (2):   [  (ELSE):   [.status: occupied, .p : [white,pawn]]],
24                   (3:6): [  (ELSE):   [.status: free]],
25                   (7):   [  (ELSE):   [.status: occupied, .p : [black,pawn]]],
26                   (8):   [  (a,h):    [.status: occupied, .p : [black,rook]],
27                             (b,g):    [.status: occupied, .p : [black,knight]],
28                             (c,f):    [.status: occupied, .p : [black,bishop]],
29                             (d):      [.status: occupied, .p : [black,queen]],
30                             (e):      [.status: occupied, .p : [black,king]]]
31                ];
32          RETURN;
33      END initialize;
34      register_move:
35      PROC (b board LOC,m move) EXCEPTIONS (illegal);
36          DCL  starting square LOC:= b (m.lin_1)(m.col_1),
37               arriving square LOC:= b (m.lin_2)(m.col_2);
38          DO WITH m;
39              IF starting.status=free THEN CAUSE illegal; FI;
40              IF arriving.status/=free THEN
41                  IF arriving.p.kind=king THEN CAUSE illegal; FI;
42              FI;
43              CASE starting.p.kind, starting.p.color OF
44                  (pawn),(white):
45                  IF col_1 = col_2 AND (arriving.status/=free
46                      OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
47                      OR (col_2=PRED (col_1) OR col_2=SUCC (col_1))
48                      AND arriving.status=free THEN CAUSE illegal; FI;
49                      IF arriving.status/=free THEN
50                          IF arriving.p.color=white THEN CAUSE illegal; FI; FI;
51                  (pawn),(black):
52                  IF col_1=col_2 AND (arriving.status/=free
53                      OR NOT (lin_2=lin_1–1 OR lin_2=lin_1–2 AND lin_1=7))
54                      OR (col_2=PRED (col_1) OR col_2=SUCC (col_1))
55                      AND arriving.status=free THEN CAUSE illegal; FI;
56                      IF arriving.status/=free THEN
57                          IF arriving.p.color=black THEN CAUSE illegal; FI; FI;
58                  (rook),(*):
59                  IF NOT ok_rook (b,m)
60                      THEN CAUSE illegal;
61                  FI;
62                  (bishop),(*):
63                  IF NOT ok_bishop (b,m)
64                      THEN CAUSE illegal;
65                  FI;
66                  (queen),(*):
67                  IF NOT ok_rook (b,m) AND NOT ok_bishop (b,m)
68                      THEN CAUSE illegal;
```

```
69                  FI;
70                  (knight),(*):
71                  IF ABS (  ABS (NUM (col_2)-NUM (col_1))
72                          –ABS (lin_2- lin_1)) /= 1
73                      OR    ABS (NUM (col_2)-NUM (col_1))
74                          +ABS (lin_2- lin_1) =/ 3 THEN CAUSE illegal; FI;
75                  IF arriving.status/=free THEN
76                      IF arriving.p.color=starting.p.color THEN
77                      CAUSE illegal; FI; FI;
78                  (king),(*):
79                  IF ABS (NUM (col_2)-NUM (col_1)) > 1
80                      OR ABS (lin_2- lin_1) > 1
81                      OR lin_2=lin_1 AND col_2=col_1 THEN CAUSE illegal; FI;
82                  IF arriving.status/=free THEN
83                      IF arriving.p.color=starting.p.color THEN
84                      CAUSE illegal; FI; FI;/* checking king moving to check not implemented */
85              ESAC;
86          OD;
87          arriving := starting;
88          starting := [.status:free];
89          RETURN;
90      END register_move;
91      ok_rook:
92      PROC (b board,m move) RETURNS (BOOL);
93          DCL  starting square := b (m.lin_1)(m.col_1),
94               arriving square := b (m.lin_2)(m.col_2);
95
96          DO WITH m;
97              IF NOT (col_2=col_1 OR lin_1=lin_2) THEN RETURN FALSE; FI;
98              IF arriving.status/=free THEN
99                  IF arriving.p.color=starting.p.color THEN;
100                 RETURN FALSE; FI; FI;
101             IF col_1=col_2
102                 THEN IF lin_1<lin_2
103                         THEN DO FOR lin := lin_1+1 TO lin_2–1;
104                                 IF  b (lin)(col_1).status/=free
105                                     THEN RETURN FALSE;
106                                 FI;
107                             OD;
108                         ELSE DO FOR lin := lin_1–1 DOWN TO lin_2+1;
109                                 IF b (lin)(col_1).status/=free
110                                     THEN RETURN FALSE;
111                                 FI;
112                             OD;
113                         FI;
114             ELSIF col_1<col_2
115                 THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
116                         IF b (lin_1)(col).status/=free
117                             THEN RETURN FALSE;
118                         FI;
119                     OD;
120                 ELSE DO FOR col := SUCC (col_2) DOWN TO PRED (col_1);
121                         IF b (lin_1)(col).status/=free
122                             THEN RETURN FALSE;
123                         FI;
124                     OD;
125             FI;
126             RETURN TRUE;
127         OD;
128     END ok_rook;
129     ok_bishop:
130     PROC (b board,m move) RETURNS (BOOL);
```

```
131             DCL  starting square := b (m.lin_1)(m.col_1),
132                  arriving square := b (m.lin_2)(m.col_2),
133                  col column;
134
135             DO WITH m;
136                CASE lin_2>lin_1,col_2>col_1 OF
137                   (TRUE),(TRUE): col := col_1;
138                                  DO FOR lin := lin_1+1 TO lin_2–1;
139                                       col := SUCC (col);
140                                       IF b (lin)(col).status/=free
141                                          THEN RETURN FALSE;
142                                       FI;
143                                  OD;
144                                  IF SUCC (col)/=col_2
145                                       THEN RETURN FALSE;
146                                  FI;
147                   (TRUE),(FALSE): col := col_1;
148                                  DO FOR lin := lin_1+1 TO lin_2–1;
149                                       col := PRED (col);%
150                                       IF b (lin)(col).status/=free
151                                          THEN RETURN FALSE;
152                                       FI;
153                                  OD;
154                                  IF PRED (col)/=col_2
155                                       THEN RETURN FALSE;
156                                  FI;
157                   (FALSE),(TRUE): col := col_1;
158                                  DO FOR lin := lin_1–1 DOWN TO lin_2+1;
159                                       col := SUCC (col);
160                                       IF b (lin)(col).status/=free
161                                          THEN RETURN FALSE;
162                                       FI;
163                                  OD;
164                                  IF SUCC (col)/=col_2
165                                       THEN RETURN FALSE;
166                                  FI;
167                   (FALSE),(FALSE): col := col_1;
168                                  DO FOR lin := lin_1–1 DOWN TO lin_2+1;
169                                       col := PRED (col);
170                                       IF b (lin)(col).status/=free
171                                          THEN RETURN FALSE;
172                                       FI;
173                                  OD;
174                                  IF PRED (col)/=col_2
175                                       THEN RETURN FALSE;
176                                  FI;
177             ESAC;
178             IF arriving.status=free THEN RETURN TRUE;
179             ELSE RETURN arriving.p.color/=starting.p.color; FI;
180             OD;
181          END ok_bishop;
182          END chess_fragments;
```

IV.12 Construcción y manejo de una lista circularmente enlazada

```
1          circular_list:
2          MODULE
3             handle_list:
4             MODULE
5                GRANT insert, remove, node;
6                NEWMODE node=STRUCT (pred, suc REF node, value INT);
7                DCL pool ARRAY (1:1000)node;
8                DCL head node := (: NULL,NULL,0 :);
```

```
9
10              insert: PROC (new node);
11                  /* insert actions */
12              END insert;
13
14              remove: PROC ();
15                  /* remove actions */
16              END remove;
17
18              initialize_list:
19              BEGIN
20                  DCL last REF node := ->head;
21                  DO FOR new IN pool;
22                      new.pred := last;
23                      last->.suc := ->new;
24                      last := ->new;
25                      new.value := 0;
26                  OD;
27                  head.pred := last;
28                  last->.suc := ->head;
29              END initialize_list;
30
31          END handle_list;
32          manipulate:
33          MODULE
34              SEIZE node, remove, insert;
35              DCL node_a node := (: NULL,NULL,536 :);
36              remove();
37              remove();
38              insert(node_a);
39          END manipulate;
40      END circular_list;
```

IV.13 Una región para manejar accesos competitivos a un recurso

```
1       allocate_resources:
2       REGION
3           GRANT allocate, deallocate;
4           NEWMODE resource_set = INT (0:9);
5           DCL allocated ARRAY (resource_set)BOOL:= (: (resource_set): FALSE:);
6           DCL resource_freed EVENT;
7
8           allocate:
9           PROC () RETURNS (resource_set);
10              DO FOR EVER;
11                  DO FOR i IN resource_set;
12                      IF NOT allocated(i)
13                          THEN
14                              allocated(i) := TRUE;
15                              RETURN i;
16                      FI;
17                  OD;
18                  DELAY resource_freed;
19              OD;
20          END allocate;
21
22          deallocate:
23          PROC (i resource_set);
24              allocated(i) := FALSE;
25              CONTINUE resource_freed;
26          END deallocate;
27
28      END allocate_resources;
```

IV.14 Cola de espera para las llamadas que llegan a una central

```
1       switchboard:
2       MODULE
3           /* This example illustrates a switchboard which queues incoming calls
4              and feeds them to the operator at an even rate. Every time the
5              operator is ready one and only one call is let through. This is
6              handled by a call distributor which lets calls through at fixed
7              intervals. If the operator is not ready or there are other calls
8              waiting, a new call must queue up to wait for its turn. */
9           DCL operator_is_ready,
10              switch_is_closed EVENT;
11
12          call_distributor:
13          PROCESS ();
14              wait:
15              PROC (x INT);
16                  /*some wait action*/
17              END wait;
18              DO FOR EVER;
19                  wait(10 /*seconds*/);
20                  CONTINUE operator_is_ready;
21              OD;
22          END call_distributor;
23
24          call_process:
25          PROCESS ();
26              DELAY CASE
27              (operator_is_ready): /* some actions */ ;
28              (switch_is_closed): DO FOR i IN INT (1:100);
29                                     CONTINUE operator_is_ready;
30                                     /* empty the queue*/
31                                  OD;
32              ESAC;
33          END call_process;
34
35          operator:
36          PROCESS ();
37              DCL time INT;
38              DO FOR EVER;
39                  IF time = 1700
40                      THEN CONTINUE switch_is_closed;
41                  FI;
42              OD;
43          END operator;
44
45          START call_distributor();
46          START operator();
47          DO FOR i IN INT (1:100);
48              START call_process();
49          OD;
50      END switchboard;
```

IV.15 Asignar y desasignar un conjunto de recursos

```
1       definitions:
2       MODULE
3       SIGNAL
4           acquire,
5           release=(INSTANCE),
6           congested,
```

```
7           ready,
8           advance,
9           readout=(INT);
10          GRANT ALL;
11      END definitions;
12      counter_manager:
13      MODULE
14      /*  To illustrate the use of signals and the receive case, (buffers
15          might have been used instead) we will look at an example where an
16          allocator manages a set of resources, in this case a set of
17          counters. The module is part of a larger system where there are
18          users, that can request the services of the counter_manager. The
19          module is made to consist of two process definitions, one for the
20          allocation and one for the counters. Initiate and terminate
21          are internal signals sent from the allocator
22          to the counters. All the other signals are external, being sent
23          from or to the users. */
24
25          SEIZE/* external signals */
26              acquire, release, congested,ready,advance,readout;
27          SIGNAL initiate = (INSTANCE),
28                     terminate;
29          allocator:
30          PROCESS ();
31              NEWMODE no_of_counters = INT (1:100);
32          DCL counters ARRAY (no_of_counters)
33                                  STRUCT (counter INSTANCE, status SET (busy,idle));
34          DO FOR each IN counters;
35              each := (: START counter(), idle :);
36          OD;
37          DO FOR EVER;
38          BEGIN
39              DCL user INSTANCE;
40              await_signals:
41              RECEIVE CASE SET user;
42              (acquire):
43                  DO FOR each IN counters;
44                      DO WITH each;
45                          IF status = idle
46                              THEN
47                                  status := busy;
48                                  SEND initiate (user) TO counter;
49                                  EXIT await_signals;
50                          FI;
51                      OD;
52                  OD;
53                  SEND congested TO user;
54              (release IN this_counter):
55                  SEND terminate TO this_counter;
56                  find_counter:
57                  DO FOR each IN counters;
58                      DO WITH each;
59                          IF this_counter = counter
60                              THEN
61                                  status := idle;
62                                  EXIT find_counter;
63                          FI;
64                      OD;
65                  OD find_counter;
66              ESAC await_signals;
67          END;
68          OD;
```

```
69          END allocator;
70          counter:
71          PROCESS ();
72              DO FOR EVER;
73              BEGIN
74                  DCL  user INSTANCE,
75                       count INT:= 0;
76                  RECEIVE CASE
77                      (initiate IN received_user):
78                          SEND ready TO received_user;
79                          user := received_user;
80                  ESAC;
81                  work_loop:
82                  DO FOR EVER;
83                      RECEIVE CASE
84                          (advance): count + := 1;
85                          (terminate):
86                              SEND readout(count) TO user;
87                              EXIT work_loop;
88                      ESAC;
89                  OD work_loop;
90              END;
91              OD;
92          END counter;
93          START allocator();
94      END counter_manager;
```

IV.16 Asignar y desasignar un conjunto de recursos empleando tampones

```
1
2
3       user_world:
4       MODULE
5       /* This example is the same as no.15 except that buffers are
6           used for communication instead of signals.
7           The main difference is that processes are now identified
8           by means of references to local message buffers rather than
9           by instance values. There is one message buffer declared
10          local to each process. There is one set of message types
11          for each process definition. When started each process must
12          identify its buffer address to the starting process.
13          The user_world module sketches some of the environment in
14          which the counter_manager is used. */
15
16      SEIZE allocator;
17      GRANT  user_buffers,user_messages,
18             allocator_messages, allocator_buffers,
19             counter_messages, counters_buffers;
20      NEWMODE
21          user_messages =
22              STRUCT (type SET (   congested, ready,
23                       readout, allocator_id),
24                           CASE type OF
25                               (congested) : ,
26                               (ready) : counter REF counters_buffers,
27                               (readout) : count INT,
28                               (allocator_id): allocator REF allocator_buffers
29                           ESAC),
30          user_buffers = BUFFER (1) user_messages,
31          allocator_messages =
32              STRUCT (type SET (acquire, release, counter_id),
33                           CASE type OF
34                               (acquire) : user REF user_buffers,
```

```
35                              (release,
36                              counter_id): counter REF counters_buffers
37                          ESAC),
38          allocator_buffers = BUFFER (1) allocator_messages,
39          counter_messages =
40              STRUCT (type SET (initiate, advance, terminate),
41                          CASE type OF
42                              (initiate) : user REF user_buffers,
43                              (advance,
44                              terminate):
45                          ESAC),
46          counters_buffers = BUFFER (1) counter_messages;
47      DCL  user_buffer user_buffers,
48          allocator_buf REF allocator_buffers,
49          counter_buf REF counters_buffers;
50      START allocator(–>user_buffer);
51      RECEIVE CASE
52          (user_buffer IN u_msg): allocator_buf := u_msg.allocator;
53      ESAC;
54      END user_world;
55      counter_manager:
56      MODULE
57      SEIZE    user_buffers,user_messages,
58          allocator_messages, allocator_buffers,
59          counter_messages, counters_buffers;
60      GRANT allocator;
61
62      allocator:
63      PROCESS (starter REF user_buffers);
64          DCL allocator_buffer allocator_buffers;
65          NEWMODE no_of_counters = INT (1:10);
66          DCL  counters ARRAY    (no_of_counters)
67                                  STRUCT (counter REF counters_buffers,
68                                          status SET (busy, idle)),
69              message allocator_messages;
70          SEND starter–>([allocator_id, –>allocator_buffer]);
71          DO FOR each IN counters;
72              START counter(–>allocator_buffer);
73              RECEIVE CASE
74                  (allocator_buffer IN a_msg): each := [a_msg.counter, idle];
75              ESAC;
76          OD;
77          DO FOR EVER;
78              BEGIN
79              DCL user REF user_buffers;
80              RECEIVE (allocator_buffer IN message);
81              handle_messages:
82              CASE message.type OF
83              (acquire):
84                  user := message.user;
85                  DO FOR each IN counters;
86                      DO WITH each;
87                          IF status= idle
88                              THEN status := busy;
89                                  SEND counter–>([initiate, user]);
90                                  EXIT handle_messages;
91                          FI;
92                      OD;
93                  OD;
94                  SEND user–>([congested]);
95              (release):
96                  SEND message.counter–>([terminate]);
97                  find_counter:
```

```
98                  DO FOR each IN counters;
99                      DO WITH each;
100                         IF message.counter = counter
101                             THENstatus := idle;
102                                 EXIT find_counter;
103                         FI;
104                     OD;
105                 OD find_counter;
106             (counter_id): ;
107             ESAC handle_messages;
108             END;
109         OD;
110     END allocator;
111     counter:
112     PROCESS (starter REF allocator_buffers);
113         DCL counter_buffer counters_buffers;
114         SEND starter–>([counter_id, –>counter_buffer]);
115         DO FOR EVER;
116             BEGIN
117                 DCL  user REF user_buffers,
118                      count INT:= 0,
119                      message counter_messages;
120                 RECEIVE (counter_buffer IN message);
121                 CASE message.type OF
122                     (initiate):     user := message.user;
123                                     SEND user–>([ready, –>counter_buffer]);
124                     ELSE /* some error action */
125                 ESAC;
126             work_loop:
127             DO FOR EVER;
128                 RECEIVE (counter_buffer IN message);
129                 CASE message.type OF
130                     (advance):    count + := 1;
131                     (terminate):  SEND user–>([readout, count]);
132                                   EXIT work_loop;
133                     ELSE /* some error action */
134                 ESAC;
135             OD work_loop;
136         END;
137         OD;
138     END counter;
139     END counter_manager;
```

IV.17 Explorador de cadena 1

```
1       string_scanner1: /*     This program implements strings by means
2                               of packed arrays of characters. */
3       MODULE
4           SYN
5               blanks ARRAY (0:9)CHAR PACK = [(*):' ' ], linelength = 132;
6           SYNMODE
7               stringptr = ROW ARRAY (lineindex)CHAR PACK,
8               lineindex = INT (0:linelength–1);
9
10      scanner:
11      PROC(string stringptr, scanstart lineindex INOUT,
12              scanstop lineindex, stopset POWERSET CHAR)
13              RETURNS (ARRAY (0:9)CHAR PACK);
14          DCL  count INT:= 0,
15               res ARRAY (0:9)CHAR PACK:= blanks;
16          DO
17              FOR c IN string–>(scanstart:scanstop)
18              WHILE NOT (c IN stopset);
```

```
19              count + := 1;
20          OD;
21          IF count>0
22              THEN
23                  IF count>10
24                      THEN
25                          count := 10;
26                  FI;
27                  res(0:count–1) := string–>(scanstart:scanstart+count–1);
28          FI;
29          RESULT res;
30          IF scanstart+count < scanstop
31              THEN
32                  scanstart := scanstart+count+1;
33          FI;
34      END scanner;
35
36      GRANT scanner;
37
38      END string_scanner1;
```

IV.18 Explorador de cadena 2

```
1       string_scanner2: /*     This example is the same as no.17 but it uses
2                               character string instead of packed arrays */
3       MODULE
4           SYN
5               blanks = (10)" ", linelength = 132;
6           SYNMODE
7               stringptr = ROW CHARS (linelength),
8               lineindex = INT (0:linelength–1);
9
10          scanner:
11              PROC(string stringptr, scanstart lineindex INOUT,
12                          scanstop lineindex, stopset POWERSET CHAR)
13                          RETURNS (CHARS (10));
14                  DCL count INT:= 0;
15                  DO FOR i := scanstart TO scanstop
16                      WHILE NOT (string–>(i) IN stopset);
17                          count + := 1;
18                  OD;
19                  IF count>0
20                      THEN
21                          IF count>=10
22                              THEN
23                                  RESULT string–>(scanstart UP 10);
24                              ELSE
25                                  RESULT string–>(scanstart:scanstart+count–1)
26                                          //blanks(count:9);
27                          FI;
28                      ELSE
29                          RESULT blanks;
30                  FI;
31                  IF scanstart+count < scanstop
32                      THEN
33                          scanstart := scanstart+count+1;
34                  FI;
35              END scanner;
36
37          GRANT scanner;
38
39      END string_scanner2;
```

IV.19 Supresión de un ítem en una lista doblemente enlazada

```
1       queue: MODULE
2           SYNMODE info=INT;
3           queue_removal:
4           MODULE
5               SEIZE info;
6               GRANT remove;
7               remove:
8               PROC(p PTR) RETURNS (info) EXCEPTIONS (EMPTY);
9                       /*  This procedure removes the item referred to
10                          by p from a queue and returns the information
11                          contents of that queue element */
12                      SYNMODE  element = STRUCT (
13                                      i info POS (0,8:31),
14                                      prev PTR POS (1,0:15),
15                                      next PTR POS (1,16:31));
16                      DCL x REF element LOC:= element(p), prev, next PTR;
17                      prev := x–>.prev;
18                      next := x–>.next;
19                      x–>.prev, x–>.next := NULL;
20                      RESULT x–>.i;
21                      p := prev;
22                      x–>.next := next;
23                      p := next;
24                      x–>.prev := prev;
25              END remove;
26          END queue_removal;
27      END queue;
```

IV.20 Actualizar un registro de un fichero

```
1       read_modify_write:
2       MODULE
3
4           /*  this example indicates how the CHILL i/o concepts can be used     */
5           /*  to write an application where a record of a random accessible     */
6           /*  file can be updated or added if not yet in use                    */
7
8           NEWMODE
9               index_set = INT (1:1000),
10              record_type = STRUCT (
11                                      free        BOOL,
12                                      count       INT,
13                                      name        CHARS (20));
14
15          DCL
16              curindex            index_set,
17              file_association    ASSOCIATION,
18              record_file         ACCESS (index_set) record_type,
19              record_buffer       record_type;
20
21          ASSOCIATE (file_association,"DSK:RECORDS.DAT");     /* create association     */
22          CONNECT (record_file,file_association,READWRITE);   /* connect to file        */
23          curindex := 123;                                    /* position record        */
24          READRECORD (record_file,curindex,record_buffer);    /* read the record        */
25              IF record_buffer.free                           /* if record is free      */
26              THEN                                        /* the claim and          */
27                  record_buffer.free := FALSE             /* initialize it          */
28                  record_buffer.count := 0;
29                  record_buffer.name := "CHILL I/O concept  ";
```

```
30              FI;
31          record_buffer.count + := 1;                          /* increment its count    */
32          WRITERECORD (record_file, curindex, record_buffer);  /* write the record       */
33          DISSOCIATE (file_association);                       /* end the association    */
34
35      END read_modify_write;
```

IV.21 Fusión de dos ficheros clasificados

```
1       merge_sorted_files:
2       MODULE
3
4           /* this example shows how two sorted files can be merged into one      */
5           /* new sorted file, where the field 'key' is used for sorting          */
6           /* the old sorted files are deleted after the merging has been done    */
7
8           NEWMODE
9               record_type = STRUCT (
10                                      key     INT,
11                                      name    CHARS (50));
12
13          DCL
14              flag            BOOL,
15              infiles         ARRAY (BOOL) ACCESS record_type,
16              outfile         ACCESS record_type,
17              buffers         ARRAY (BOOL) record_type,
18              innames         ARRAY (BOOL) CHARS (10) INIT:= ["FILE.IN.1 ","FILE.IN.2 "],
19              outname         CHARS (10) INIT:= "FILE.OUT  ",
20              inassocs        ARRAY (BOOL) ASSOCIATION,
21              outassoc        ASSOCIATION;
22
23          /* associate both sorted input files, connect an access to them for input   */
24          /* and read their first record into a buffer                                */
25
26          DO
27              FOR  curfile IN infiles,
28                   curbuffer IN buffers,
29                   curassoc IN inassocs,
30                   curname IN innames;
31                  CONNECT (curfile, ASSOCIATE (curassoc,curname), READONLY);
32                  READRECORD (curfile, curbuffer);
33          OD;
34
35          /* associate the output file, create a file for the association     */
36          /* and connect an access to it for output                           */
37
38          ASSOCIATE (outassoc,outname);
39          CREATE (outassoc);
40          CONNECT (outfile, outassoc, WRITEONLY);
41          merge_files:
42          DO FOR EVER
43
44              /* determine which file, if any at all, to process next    */
45              /* 'flag' indicates the file                               */
46
47              CASE OUTOFFILE (infiles(FALSE)),OUTOFFILE (infiles(TRUE)) OF
48                  (TRUE),  (TRUE):                    /* both files are empty   */
49                           EXIT merge_files;
50                  (TRUE),  (FALSE):                   /* one file is empty      */
51                           flag := TRUE;
52                  (FALSE), (TRUE):                    /* one file is empty      */
53                           flag := FALSE;
```

```
54              (FALSE), (FALSE):                       /* no file is empty        */
55                      flag := buffers(FALSE).key>buffers(TRUE).key;
56          ESAC;
57
58              /* output the buffer which currently contains a record with the    */
59              /* smallest value for 'key', fill the buffer with a new record     */
60
61              WRITERECORD (outfile,buffers(flag));
62              READRECORD (infiles(flag), buffers(flag));
63      OD merge_files;
64
65      /* delete the input files and close the output file */
66      DO
67      FOR curassoc IN inassocs;
68          DELETE (curassoc);                      /* delete the file          */
69          DISSOCIATE (curassoc);                  /* and terminate association*/
70      OD;
71      DISSOCIATE (outassoc);                      /* disconnect and terminate */
72
73  END merge_sorted_files;
```

IV.22 Lectura de un fichero con registros de longitud variable

```
1   variable_length_records:
2   MODULE
3
4       /* This example shows how a file which consists of variable length     */
5       /* records can be treated.                                             */
6       /* The file consists of a number of strings of varying length; the     */
7       /* algorithm will read a string, allocate an appropriate location      */
8       /* for it, and put the reference to this location into a push down list */
9
10      NEWMODE
11          string = CHARS (80),
12          link_record = STRUCT (
13                                  next_record   REF link_record,
14                                  string_row    ROW string);
15
16      DCL
17          pushdownlist    REF link_record INIT:= NULL,
18          length          INT (1:80),
19          temporaryrow    ROW string,
20          fileaccess      string DYNAMIC,
21          association     ASSOCIATION;
22          filename        CHARS (20) VARYING INIT := "INPUT.DATA";
23      ASSOCIATE (association,filename);           /* associate the input file */
24      CONNECT (fileaccess, association, READONLY); /* connect access for input */
25      temporaryrow := READRECORD (fileaccess);    /* read the first record    */
26      DO                                          /* while not end-of-file    */
27          WHILE NOT(OUTOFFILE(fileaccess));
28              pushdownlist := ALLOCATE (link_record,  /* get a new link record */
29                                  [pushdownlist,NULL ]); /* and initialize it */
30              length := 1 + UPPER (temporaryrow->);   /* determine length of string */
31          DO
32              WITH pushdowlist->;                 /* add new string to list   */
33              string_row := ALLOCATE (CHARS (length),  /* allocate space for string */
34                                      temporaryrow->); /* and fill it          */
35          OD;
36          temporaryrow := READRECORD (fileaccess); /* get next record in file */
37      OD;
38      DISSOCIATE (association);                   /* end the association      */
39
40  END variable_length_records;
```

IV.23 Utilización de módulos de spec

```
1       /* The examples 23 and 24 are example 8 divided in two pieces. */
2       letter_count:
3       SPEC MODULE
4           /* This is a spec module for the corresponding module in example 8. */
5           SEIZE max;
6           count:
7           PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT) END;
8           GRANT count;
9       END letter_count;
10      letter_count: REMOTE "example 24";
11      test:
12      MODULE
13          /* This is the module 'test' from example 8.          */
14          /* It can now be piecewise compiled together with     */
15          /* the above spec module                              */
16          SYNMODE results = ARRAY ('A':'Z') INT;
17          DCL c CHARS (10) INIT:= "A-B<ZAA9K' ";
18          DCL output results;
19          SYN max = 10_000;
20          GRANT max;
21          SEIZE count;
22          count (–> c, output);
23          ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, (ELSE) : 0 ];
24      END test;
```

IV.24 Ejemplo de un contexto

```
1       CONTEXT
2           /* This is a context for the module "letter_count"    */
3           /* as used in example 23, allowing the piecewise      */
4           /* compilation of "letter_count"                      */
5           SYN max = 10_000;
6       FOR
7       letter_count:
8       MODULE
9           SEIZE max;
10          DCL letter POWERSET CHAR INIT:= ['A' : 'Z'];
11          count:
12          PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT);
13              output := [(ELSE) : 0];
14              DO FOR i := 0 TO UPPER (input –>);
15                  IF input –> (i) IN letter THEN
16                      output (input –> (i)) + := 1;
17                  FI;
18              OD;
19          END count;
20          GRANT count;
21      END letter_count;
```

IV.25 Utilización de prefijación y módulos remotos

```
1           /* This example uses the module 'stack' from example 27 or 28.   */
2           /* It shows how prefixes can be used to prevent name clashes.    */
3           /* It uses the remote construct to share the source code.        */
4       char_stack:
5       MODULE
6           SYNMODE element = CHAR;
7           GRANT (–> stack ! char) ! ALL;
8           stack: SPEC REMOTE"example 29";
```

```
9           stack: REMOTE       "example 27 or 28 for CHAR";
10      END char_stack;
11
12      int_stack:
13      MODULE
14          SYNMODE element = INT;
15          GRANT (–> stack ! int) ! ALL;
16          stack: SPEC REMOTE"example 29";
17          stack: REMOTE       "example 27 or 28 for CHAR";
18      END int_stack;
19          /* Here 'push', 'pop' and 'element' are visible but      */
20          /* with prefixes 'stack ! char' and 'stack ! int' for    */
21          /* the implementations with element = CHAR and           */
22          /* element = INT, respectively.                          */
23          /* Below are some possibilities of using the granted     */
24          /* names inside modules.                                 */
25      MODULE
26          SEIZE ALL PREFIXED stack ;
27          DCL c CHAR;
28          int ! push (123) ;
29          char ! push ('a') ;
30          int ! pop ( ) ;
31          c := char ! elem (1) ;
32      END;
33
34      MODULE
35          SEIZE (stack ! int –> stack) ! ALL;
36          stack ! push (345) ;
37          stack ! pop ( ) ;
38      END;
```

IV.26 Utilización de entrada/salida de texto

```
1       textio:
2       MODULE
3
4           /* This example shows the use of the text i/o features. */
5
6           DCL
7               outfile     ASSOCIATION,
8               output      TEXT (80) DYNAMIC,
9               size        INT:= 12345,
10              flag        BOOL:= FALSE,
11              set         SET (a,b,c) := b,
12              s1          CHARS (5) := "CHILL",
13              s2          CHARS (5) VARYING:= "text";
14
15          ASSOCIATE (outfile,"OUTPUT.DATA");          - - associate the output file
16          CREATE (outfile);                           - - create it
17          CONNECT (output,outfile,WRITEONLY);         - - then connect text location
18          WRITETEXT (output,"%B%/",10);               - - 1010
19          WRITETEXT (output,"%C%/",set);              - - b
20          WRITETEXT (output,"size = %C%/",size);      - - size = 12345
21          WRITETEXT (output,"%CL6%C i/o%/",s1,s2);    - - CHILL text i/o
22          WRITETEXT (output,"flag =%X%C",flag);       - - flag = FALSE
23          size := GETTEXTINDEX (output);              - - 12
24          DISSOCIATE (outfile);
25      END textio;
```

IV.27 Una pila genérica

```
1       /* This example implements a generic stack. Please      */
2       /* note that the element mode has been left out.        */
3       /* The element mode is defined in the surroundings.     */
4       /* The context is a virtually introduced context,       */
5       /* and it has no source.                                */
6     CONTEXT REMOTE FOR
7     stack:
8     MODULE
9       SEIZE element;
10      NEWMODE cell = STRUCT (pred,succ REF cell,info element);
11      DCL p,last,first REF cell INIT:= NULL;
12
13      push:
14      PROC (e element) EXCEPTIONS (overflow)
15        p := ALLOCATE (cell) ON (ALLOCATEFAIL): CAUSE overflow; END;
16        IF last = NULL THEN
17          first := p;
18          last := p;
19        ELSE
20          last –> .succ := p;
21          p –> .pred := last;
22          last := p;
23        FI;
24        last –> .info := e;
25        RETURN;
26      END push;
27
28      pop:
29      PROC () EXCEPTIONS (underflow)
30        IF last = NULL THEN
31          CAUSE underflow;
32        FI;
33        p:= last;
34        last := last –> .pred;
35        IF last = NULL THEN
36          first := NULL;
37        ELSE
38          last –> .succ := NULL;
39        FI;
40        TERMINATE (p);
41        RETURN;
42      END pop;
43
44      elem:
45      PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
46        IF first = NULL THEN
47          CAUSE bounds;
48        FI;
49        p := first;
50        DO FOR j := 2 TO i;
51          IF p –> .succ = NULL THEN
52            CAUSE bounds;
53          FI;
54          p := p –> .succ;
55        OD;
56        RETURN p –> .info;
57      END elem;
58
59      GRANT push,pop,elem;
60    END stack;
```

IV.28 Un tipo de datos abstracto

```
1       /* This example implements a stack with the same functionality      */
2       /* of example 27, demonstrating how an abstract data type           */
3       /* can be implemented in two different ways in CHILL.               */
4     CONTEXT REMOTE FOR
5     stack:
6     MODULE
7       SEIZE element;
8       SYN max = 10_000, min = 1;
9       DCL   stack ARRAY (min : max) element,
10
11              stackindex INT INIT:= min-1;
12      push:
13      PROC (e element) EXCEPTIONS (overflow)
14        IF stackindex = max THEN
15          CAUSE overflow;
16        FI;
17        stackindex +:= 1;
18        stack(stackindex) := e;
19        RETURN;
20      END push;
21      pop:
22      PROC () EXCEPTIONS (underflow)
23        IF stackindex = min THEN
24          CAUSE underflow;
25        FI;
26        stackindex–:= 1;
27        RETURN;
28      END pop;
29
30      elem:
31      PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
32        IF i < min OR i > max THEN
33          CAUSE bounds;
34        FI;
35        RETURN stack(i);
36      END elem;
37
38      GRANT push,pop,elem;
39    END stacks;
```

IV.29 Ejemplo de un módulo de spec

```
1       /* This SPEC MODULE defines the interface of examples 27 and 28. */
2     stack: SPEC MODULE
3       SEIZE: element;
4       push: PROC (e element) EXCEPTIONS (overflow) END;
5       pop: PROC () EXCEPTIONS (underflow) END;
6       elem: PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds) END;
7       GRANT push,pop,elem;
8     END stack;
```

IV.30 Orientación a objetos: modos para pilas simples, secuenciales

```
1     /*  The examples show the application of object-orientation to the well known stack data structure.
2         Two different implementations of stack modes with identical interfaces are realized (Example 30).
3         Based on these modes extended modes with an additional operation (e.g. Top; Example 31) or
4         with other properties (e.g. mutual exclusive access to stacks (Example 32)) are realized.      */
5
6     SYNMODE   StackMode1 = MODULE SPEC      /* ---------------- Definition of the interface   */
```

```
7           GRANT ElementMode, Push, Pop;                          /* Simple, sequential stack  */
8           NEWMODE ElementMode = STRUCT (a INT, b BOOL);
9           Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow) END Push;
10          Pop:  PROC( ) RETURNS(ElementMode) EXCEPTIONS(Underflow) END Pop;
11          SYN  Length = 10_000;
12          DCL  StackData ARRAY (1:Length) ElementMode,           /* Array implementation*/
13                TopOfStack RANGE(0:Length) INIT := 0;            /* of the stack*/
14      END StackMode1;
15
16      SYNMODE  StackMode1 = MODULE BODY    /* ------------------Definition of the body */
17          Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow)
18              IF TopOfStack = Length THEN
19                  CAUSE Overflow;
20              ELSE
21                  TopOfStack +:= 1;
22                  StackData(TopOfStack) := Elem;
23          FI;
24          END Push;
25          Pop: PROC( ) RETURNS(ElementMode) EXCEPTIONS(Underflow)
26              IF TopOfStack = 0 THEN
27                  CAUSE Underflow;
28              ELSE
29                  RESULT(StackData(TopOfStack));
30                  TopOfStack –:= 1;
31              FI;
32          END Pop;
33      END StackMode1;
34
35      MainProgram1: MODULE
36          SEIZE StackMode1;
37          DCL Stack1 StackMode1;
38          DCL Elem1 StackMode1!ElementMode;
39          Elem1 := [10, TRUE];
40          Stack1.Push(Elem1);
41          Stack1.Push( [20, FALSE] );
42      END MainProgram1;
43
44      SYNMODE StackMode2 = MODULE SPEC     /* ----------------- Definition of the interface  */
45          GRANT ElementMode, Push, Pop;                    /* Same interface as StackMode1  */
46          NEWMODE ElementMode = STRUCT (a INT, b BOOL);
47          Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow) END Push;
48          Pop:  PROC( ) RETURNS(ElementMode) EXCEPTIONS(Underflow) END Pop;
49
50          NEWMODE ListElem = STRUCT    (next REF ListElem,     /* List implementation  */
51                                        info ElementMode);          /* of the stack  */
52          DCL Stack REF ListElem INIT := NULL;
53      END StackMode2;
54
55      SYNMODE StackMode2 = MODULE BODY      /* ---------------------- Definition of the body */
56          Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow)
57              Stack := ALLOCATE (ListElem, [Stack, Elem])
58                  ON (ALLOCATEFAIL) : CAUSE Overflow; END;
59          END Push;
60          Pop:  PROC( ) RETURNS(ElementMode) EXCEPTIONS(Underflow)
61              DCL Temp REF ListElem;
62              IF Stack = NULL THEN
63                  CAUSE Underflow;
64              ELSE
65                  RESULT (Stack–>.info);
66                  Temp := Stack;
67                  Stack := Stack–>.next;
68                  TERMINATE (Temp);
```

```
69              FI;
70          END Pop;
71      END StackMode2;
72
73      MainProgram2: MODULE   /* ------------------------ Essentially the same as MainProgram1 */
74          SEIZE StackMode2;
75          DCL Stack1 StackMode2;
76          DCL Elem1 StackMode2!ElementMode;
77          Elem1 := [10, TRUE];
78          Stack1.Push(Elem1);
79          Stack1.Push( [20, FALSE] );
80      END MainProgram2;
```

IV.31 Orientación a objetos: extensión de modo: pila simple, secuencial con operación "top"

```
1
2       SYNMODE StackWithTopMode2 =     MODULE SPEC          /* BASED_ON indicates   */
3                                       BASED_ON StackMode2     /* mode derivation or   */
4           GRANT Top;                                                /* inheritance   */
5           Top: PROC( ) RETURNS (ElementMode)              /* Top is an additional operation */
6                       EXCEPTIONS (EmptyStack) END Top;
7       END StackWithTopMode2 ;
8
9       SYNMODE StackWithTopMode2 = MODULE BODY BASED_ON StackMode2
10          Top: PROC( ) RETURNS (ElementMode) EXCEPTIONS (EmptyStack)
11              IF Stack = NULL THEN
12                  CAUSE EmptyStack;
13              ELSE
14                  RETURN (Stack–>.info);
15              FI;
16          END Top;
17      END StackWithTopMode2 ;
18
19      MainProgram3: MODULE   /* ---------------------------------- Very similar to MainProgram2 */
20          SEIZE StackWithTopMode2;
21          DCL Stack1 StackWithTopMode2;
22          DCL Elem1 StackWithTopMode2!ElementMode;
23          Elem1 := [10, TRUE];
24          Stack1.Push(Elem1);
25          Stack1.Push( [20, FALSE] );
26          Elem1 := Stack1.Top( );
27      END MainProgram3;
```

IV.32 Orientación a objetos: modos para pilas con sincronización de acceso

```
1       /* Based on the mode StackWithTopMode2 defined in example 31 the mode
2          RegionStackWithTopMode1 is defined whose objects behave like regions:
3          at any point in time at most one of the public procedures may be in execution.
4          Apart from this the behavior is essentially the same as for StackWithTopMode2:
5          erroneous use of an object causes an exception. The second mode
6          RegionStackWithTopMode2 uses the CHILL event mechanism to deal with
7          erroneous use of a stack object.                                          */
8
9       SYNMODE RegionStackWithTopMode1 = REGION SPEC BASED_ON StackWithTopMode2
10              /* Just put the base mode into a "region envelope" */
11              /* In case of an erroneous use same behaviour as StackWithTopMode2: cause an exception */
12      END RegionStackWithTopMode1 ;
13
14      SYNMODE RegionStackWithTopMode1 = REGION BODY BASED_ON StackWithTopMode2
15      END RegionStackWithTopMode1 ;
16
```

```
17      MainProgram4: MODULE
18          SEIZE RegionStackWithTopMode1;
19          DCL Stack1 RegionStackWithTopMode1;
20          Producer: PROCESS ( ) ;
21              DCL Elem1 RegionStackWithTopMode1!ElementMode;
22              DO FOR EVER
23                  /* compute Elem1 */
24                  Stack1.Push(Elem1);
25              OD;
26          END Producer;
27          Consumer: PROCESS ( ) ;
28              DCL Elem1 RegionStackWithTopMode1!ElementMode;
29              DO FOR EVER
30                  Elem1 := Stack1.Pop ( );
31                  /* process Elem1 */
32              OD;
33          END Consumer;
34          START Producer ( );
35          START Consumer ( );
36      END MainProgram4;
37
38      SYNMODE RegionStackWithTopMode2 = REGION SPEC BASED_ON StackWithTopMode2
39              /* In case of an erroneous use different behaviour as StackWithTopMode2:
40                 use the event mechanism */
41          GRANT Push, Pop, Top;
42          Push: PROC(Elem ElementMode IN) REIMPLEMENT END Push;
43          Pop:  PROC( ) RETURNS (ElementMode) REIMPLEMENT END Pop;
44          Top:  PROC( ) RETURNS (ElementMode) REIMPLEMENT END Top;
45          DCL NotEmpty, NotFull EVENT;
46      END RegionStackWithTopMode2 ;
47
48      SYNMODE RegionStackWithTopMode2 = REGION BODY BASED_ON StackWithTopMode2
49          Push: PROC(Elem ElementMode IN) REIMPLEMENT
50              PushLoop: DO
51                  BEGIN
52                      StackWithTopMode2!Push(Elem);
53                      EXIT PushLoop;
54                  END
55                  ON (Overflow): DELAY NotFull; END;
56              OD PushLoop;
57              CONTINUE NotEmpty;
58          END Push;
59          Pop:  PROC( ) RETURNS(ElementMode) REIMPLEMENT
60              PopLoop: DO
61                  BEGIN
62                      RESULT StackWithTopMode2!Pop( );
63                      EXIT PopLoop;
64                  END
65                  ON (Underflow): DELAY NotEmpty; END;
66              OD PopLoop;
67              CONTINUE NotFull;
68          END Pop;
69          Top: PROC( ) RETURNS (ElementMode) REIMPLEMENT
70              TopLoop: DO
71                  BEGIN
72                      RESULT StackWithTopMode2!Top( );
73                      EXIT TopLoop;
74                  END
75                  ON (EmptyStack): DELAY NotEmpty; END;
76              OD TopLoop;
77              CONTINUE NotFull;
78          END Top;
```

```
79      END RegionStackWithTopMode2 ;
80
81      MainProgram5: MODULE   /* -------------------------- Essentially the same as MainProgram4  */
82          SEIZE RegionStackWithTopMode2;
83          DCL Stack1  RegionStackWithTopMode2;
84          Producer: PROCESS ( ) ;
85              DCL Elem1  RegionStackWithTopMode2!ElementMode;
86              DO FOR EVER
87                  /* compute Elem1 */
88                  Stack1.Push(Elem1);
89              OD;
90          END Producer;
91          Consumer: PROCESS ( ) ;
92              DCL Elem1  RegionStackWithTopMode2!ElementMode;
93              DO FOR EVER
94                  Elem1 := Stack1.Pop (Elem1);
95                  /* process  Elem1 */
96              OD;
97          END Consumer;
98          START Producer ( );
99          START Consumer ( );
100     END MainProgram5;
```

Apéndice V

Características suprimidas

Los aspectos descritos en este apéndice no están incluidos en la presente Recomendación pero si estaban contenidos en la versión de la Recomendación Z.200 (1984), *Libro Rojo*, Tomo VI – Fascículo VI.12 y en la versión de la Recomendación Z.200 (1988), *Libro Azul*, Tomo X, Fascículo X.6. A continuación se da una breve descripción de esos aspectos; para una definición completa de las mismas, véanse las subcláusulas correspondientes de la Recomendación Z.200 (1984), que se mencionan a continuación. Estos aspectos pueden ser soportados por una implementación. Si no se da una indicación, las referencias se hacen a la Recomendación Z.200 (1984).

V.1 Directiva de liberación (véase 2.6)

Una directiva de liberación liberaba las cadenas de nombre simple **reservadas** especificadas en la *lista de cadenas de nombre simple reservadas*, de modo que podían ser redefinidas.

V.2 Sintaxis de los modos enteros (véase 3.4.2)

BIN era sintaxis derivada para *INT*.

V.3 Modos conjunto con huecos (véase 3.4.5)

Un modo conjunto definía un conjunto de valores nominados o innominados. Un modo conjunto era un modo conjunto **con huecos** si y sólo si el número de sus nombres de **elemento de conjunto** era inferior al **número de valores** del modo conjunto.

V.4 Sintaxis de los modos procedimiento (véase 3.7)

Una *espec de resultado* sin la cadena de nombre simple **reservada** facultativa **RETURNS** era una sintaxis derivada para la *espec de resultado* con **RETURNS**.

V.5 Sintaxis de modo cadena (véase 3.11.2)

La notación **CHAR** *(n)* y **BIT** *(n)* denotaban cadenas de caracteres y cadenas de bits respectivamente.

V.6 Sintaxis de los modos matriz (véase 3.11.3)

La cadena de nombre simple **reservada ARRAY** era facultativa.

V.7 Notación de la estructura de niveles (véase 3.11.5)

El *modo estructura de niveles* era sintaxis derivada para un *modo estructura anidada* único. En la notación de la estructura de niveles los campos iban precedidos por un número de nivel. Si una estructura contenía campos que a su vez eran estructuras o matrices de estructuras, se formaba una jerarquía de estructuras y se podía asociar a cada campo un número de nivel. En vez de escribir modos de estructura anidada, se permitía en el *modo estructura de niveles* escribir el número de nivel delante del nombre de campo.

V.8 Nombres de referencia de correspondencia (véase 3.11.6)

Los nombres de referencia de correspondencia podían utilizarse para especificar la relación de correspondencia en un modo definido por la implementación.

V.9 Declaraciones basadas (véase 4.1.4)

Una declaración basada sin un *nombre de localización de referencia libre o ligada* era sintaxis derivada para una sentencia de definición de sínmodo. Una declaración basada con un *nombre de localización de referencia libre o ligada* definía uno o más nombres de acceso. Estos nombres constituían una forma alternativa de acceder a una localización, desreferenciando el valor de referencia contenido en la localización de referencia especificada. Esta operación de desreferenciación se efectuaba única y exclusivamente cuando se establecía un acceso mediante un nombre **basado** declarado.

V.10 Literales de cadena de caracteres (véase 5.2.4.6)

Los literales de cadena de caracteres eran delimitados por caracteres de apóstrofo. Además de la representación imprimible, podía emplearse la representación hexadecimal. Los literales de cadena de caracteres de longitud uno servían como literales de carácter.

V.11 Expresiones de recibir [véase 5.3.9/Z.200 (1988)]

Se utilizaban expresiones de recibir para recibir valores de localizaciones de tampón. El proceso ejecutante podía ser demorado y podía reactivar otro proceso, que había sido demorado al enviar un valor a la localización de tampón especificada.

V.12 Notación addr (véase 5.3.8)

ADDR (<*localización*>) era sintaxis derivada para –> <*localización*>.

V.13 Sintaxis de asignación (véase 6.2)

El símbolo = era sintaxis derivada para el símbolo := .

V.14 Sintaxis de la acción de caso (véase 6.4)

La *lista de intervalos* de una *acción de caso* podía especificarse más generalmente por un *modo discreto*, y no solamente por un *nombre de modo discreto*.

V.15 Sintaxis de la acción hacer para (véase 6.5.2)

El intervalo de la *enumeración por intervalo* de la acción hacer para podía especificarse más generalmente por un modo *discreto* y no solamente por un *nombre de modo discreto*.

V.16 Contadores de bucle explícitos (véase 6.5.2)

Si un nombre de acceso estaba visible en el dominio en que estaba situada la *acción hacer*, y ese nombre era igual a uno de los nombres definidos por un *contador de bucle*, el *contador de bucle* era **explícito**; en otro caso, era **implícito**. En el primer caso, el valor del contador de bucle se almacenaba en la localización denotada, justamente antes de una terminación anormal. Se distinguía entre terminación **normal** y **anormal**. La terminación normal se producía si la evaluación de por lo menos uno de los contadores de bucle indicaba terminación. Se producía una terminación anormal si la evaluación de la condición mientras entregaba *FALSE* o si se abandonaba la acción hacer por haberse transferido el control fuera de la misma.

V.17 Sintaxis de la acción llamar (véase 6.7)

La cadena de nombre simple reservada **CALL** era facultativa. Una *acción llamar* con **CALL** era sintaxis derivada de una *acción llamar* sin **CALL**.

V.18 Excepción RECURSEFAIL (véase 6.7)

La excepción *RECURSEFAIL* era causada cuando un procedimiento **no recursivo** se llamaba a sí mismo recursivamente.

V.19 Sintaxis de la acción arrancar (véase 6.13)

La *acción arrancar* con la opción **SET** era sintaxis derivada para la acción de asignación simple:

<localización ejemplar> : := <expresión arrancar>.

V.20 Nombres explícitos de valor a recibir (véase 6.19)

Una acción recibir señal y elegir y una acción recibir tampón y elegir podían introducir nombres de **valor a recibir**. Si un nombre era visible en el dominio donde se situaba una *acción recibir señal y elegir*, y era igual a uno de los nombres introducidos después de **IN**, el nombre de **valor a recibir** era **explícito**, y si no era **implícito**. En el primer caso, el valor a recibir era almacenado en la localización denotada inmediatamente antes de la ejecución de la lista de sentencias de acción.

V.21 Bloques (véase 8.1)

La *acción condicional*, la *acción de caso*, la *acción hacer* y la *acción demorar y elegir* no estaban definidas como bloques.

V.22 Sentencia de entrada (véase 8.4)

Un procedimiento podía tener múltiples puntos de entrada establecidos mediante sentencias de entrada. Dichas sentencias se consideraban definiciones de procedimiento adicionales. La ocurrencia de definición en la sentencia de entrada definía el nombre del punto de entrada en el procedimiento en cuyo dominio estaba situado. El punto de entrada se determinaba por la posición textual de la sentencia de entrada.

V.23 Nombres de registro (véase 8.4)

La especificación de registro podía facilitarse en el parámetro formal del procedimiento y en la espec de resultado. En el caso de transferencia por valor, ello significaba que el valor efectivo estaba contenido en el registro especificado; en el caso de transferencia por localización indicaba que el puntero (oculto) de la localización efectiva estaba contenido en el registro especificado. Si la especificación estaba en la espec de resultado, ello indicaba que el valor proporcionado o el puntero (oculto) que señalaba la localización suministrada estaba contenido en el registro especificado.

V.24 Atributo recursivo [véase 10.4/Z.200 (1988)]

La **recursividad** de procedimientos era un valor por defecto en la implementación, a menos que el atributo **RECURSIVE** estuviese especificado en una lista de atributos de procedimiento.

V.25 Cuasi sentencias de causa y cuasi manejadores (véase 8.10.3)

Cuasi sentencias de causa indicaban la presencia de sentencias de causa en módulos remotos o regiones remotas directamente encerrados en el dominio que circunda directamente el dominio del módulo de espec o región de espec en que se colocó la cuasi sentencia de causa. Cuasi manejadores indicaban la presencia, en el programa, de un manejador alcanzable desde el módulo, región o contexto directamente encerrados en el contexto a que se agregaba el cuasi manejador.

V.26 Sintaxis de cuasi sentencias [véase 10.10.3/Z.200 (1988)]

Las sentencias de cuasi procedimiento y de definición de proceso eran terminadas por un **END** <*cadena de nombre simple*>.

V.27 Nombres débilmente visibles y sentencias de visibilidad [véase 12.2.1/Z.200 (1988)]

Una *cadena de nombre* que no era **fuertemente visible** en un dominio se decía que era **débilmente visible** en el mismo si estaba **implicada** por una *cadena de nombre* que era **fuertemente visible** en el dominio. La *cadena de nombre* en el dominio se **enlazaba** a *ocurrencias de definición* **implicadas**. Si éstas no definían el mismo elemento de conjunto de modos conjunto **similares** se producía un **enfrentamiento débil**; en otro caso, la *cadena de nombre* se ligaba a ellas. En 12.2.4 se definían las *ocurrencias de definición* **implicadas**.

V.28 Nombres débilmente visibles y sentencias de visibilidad (véase 10.2.4.3)

Se decía que una *cadena de nombre* NS débilmente visible en el dominio R era tomable por el modulión M directamente encerrado en R, si NS estaba vinculada en R a una *ocurrencia de definición* no circundada por el dominio de M. Se decía que una *cadena de nombre* NS débilmente visible en el dominio R de modulión M era otorgable por M, si NS estaba vinculada en R a una *ocurrencia de definición* circundada por R.

V.29 Infiltrabilidad (véase 10.2.4.4)

Cuando una *sentencia de otorgamiento* contenía (**DIRECTLY**) **PERVASIVE**, todas las cadenas de nombre otorgadas por la misma tenían la propiedad (directamente) infiltrable en los dominios circundantes del modulión M que encerraba directamente a la *sentencia de otorgamiento*. Las cadena de nombre :

- eran **fuertemente visibles** en un dominio directamente circundante S de M;
- cuando las cadenas de nombre tenían la propiedad **directamente infiltrable** en S, tenían también la propiedad **directamente infiltrable** en M;
- si no eran **fuertemente visibles directamente** en un dominio R y eran **fuertemente visibles** en un dominio que encerraba directamente a R y en el cual tenían la propiedad de infiltrable, entonces eran **fuertemente visibles indirectamente** en R y tenían también la propiedad de infiltrable en R.

V.30 Toma mediante nombre modulión (véase 10.2.4.5)

Si una *cláusula de redenominación por prefijo* en una *sentencia de toma* tenía un *posfijo de toma* que contenía una *cadena de nombre* modulión y **ALL**, entonces la *cláusula de redenominación por prefijo* era equivalente a un conjunto de *sentencias de toma*, para toda cadena de nombre que fuese fuertemente visible en el dominio que encerraba directamente el modulión en que estaba situada la *sentencia de toma* y era tomable por este modulión, y era otorgada por el modulión asociado al *nombre de modulión* en el dominio que circundaba directamente al modulión en que estaba situada la *sentencia de toma*.

V.31 Cadenas de nombre simple predefinidas (véase C.2)

AND, NOT, OR, REM, MOD, THIS y *XOR* eran cadenas de nombre simple predefinidas.

Apéndice VI

Índice de las reglas de producción

Los números escritos en negrita indican las páginas en que se encuentran las definiciones de los distintos términos; los números en tipo de escritura corriente indican las páginas en que se utilizan los términos.

	Subcláusula de definición utilizada en subcláusula	página(s)
A		
acción		79
acción afirmar	6.10	79, **91**
acción arrancar	6.13	79, **91**
acción causar	6.12	79, **91**
acción condicional	6.3	79, **81**
acción continuar	6.15	79, **92**
acción de asignación		79
acción de asignación múltiple		79
acción de asignación simple		79
acción de caso	6.4	79, **81**
acción de temporización	9.3	79, **122**
acción de temporización absoluta	9.3.2	122, **123**
acción de temporización cíclica	9.3.3	122, **123**
acción de temporización relativa		122
acción demorar	6.16	79, **92**
acción demorar y elegir	6.17	79, **92**
acción encorchetada		79
acción enviar	6.18.1	79, **93**
acción enviar señal		93
acción enviar tampón	6.18.3	93, **94**
acción hacer	6.5.1	79, **83**
acción ir a	6.9	79, **90**
acción llamar	6.7	79, **87**
acción parar	6.14	79, **91**
acción recibir señal y elegir	6.19.2	94, **95**
acción recibir tampón y elegir	6.19.3	94, **96**
acción recibir y elegir	6.19.1	79, **94**
acción resultar	6.8	79, **90**
acción retornar	6.8	79, **90**
acción salir	6.6	79, **86**
acción vacía	6.11	79, **91**
alternativa a elegir		121
alternativa de caso	6.4	81, **82**
alternativa de demora		92
alternativa de elegir valor		71
alternativa entonces		71
alternativa recibir señal		95
alternativa recibir tampón		96
alternativa si no		71
alternativa variable		32
anchura de cláusula		115, 118
anchura de exponente		115
anchura fraccional		115
argumento de formato	7.5.3	112, **113**
argumento de localización		113
argumento de longitud	6.20.3	97, **98**

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
argumento de modo	6.20.3	97, **98**, 101
argumento de texto	7.5.3	112,**113**
argumento de upper lower	6.20.3	97,**98**
argumento de valor		113
atributo de parámetro		24
atributo de resultado	3.8	**24**
atributo resultado		24
B		
bit final		35
bit inicial		35
bloque principio-fin	10.3	79,**129**
C		
cadena de caracteres		9
cadena de control de formato	7.5.4	**114**, 115
cadena de nombre		10, 163, 165
cadena de nombre de neomodo		163
cadena de nombre prefijada		10, 11
cadena de nombre simple	2.2	**8**, 10, 11, 39, 41, 43, 44, 79, 129, 130, 134, 135, 136, 137, 138, 139, 141, 142
cadena de nombre simple		130, 142
calificador de conversión		115
campo		32
campo alternativo		32
campo de estructura	4.2.10	47, **53**
campo fijo		32
campo variable		32
carácter		9, 59, 115
carácter ancho		9
carácter ancho no reservado		60
carácter no especial		59
carácter no reservado		60
carácter no-por ciento		114
cifras significativas		21
cláusula amigo	12.2.3.4	163, **164**
cláusula de conversión		115
cláusula de e/s		115
cláusula de edición	7.5.6	115, **118**
cláusula de formato		114
cláusula de herencia de módulo		39
cláusula de herencia de región		41
cláusula de herencia de tarea		43
cláusula de implementación		39, 41, 43
cláusula de prefijo		163, 165
cláusula de prohibición	12.2.3.4	163, **164**
cláusula de redenominación por prefijo	12.2.3.3	**162**, 163, 165
cláusula directiva	2.6	**10**
cláusula e/s	7.5.7	**119**
cláusula entonces		81
cláusula parentizada	7.5.4	114, **115**
cláusula si no		81
código de control	7.5.4	114, **115**
código de conversión		115
código de edición		118
código e/s		119
comentario	2.4	**9**
comentario de fin de lína		9
comentario de fin de línea	2.4	**9**
comentario encorchetado		9

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
comillas		60
componente cuerpo de región	3.15.3	**41**, 43
componente cuerpo de tarea		43
componente de cuerpo de módulo		39
componente de especificación de módulo		39
componente de módulo común		39
componente especificación de región		41, 43
componente especificación de tarea		43
componente interfaz	3.15.5	44, **45**
componente módulo común		41, 45
comunicación cuerpo de región		41
contador de bucle		83, 84
contenido de localización		55
contexto		138
contexto remoto	10.10.1	**136**, 138
control con		86
control mientras	6.5.3	83, **86**
control para		83
conversión de expresión	5.2.11	55, **68**
conversión de localización	4.2.13	47, **54**
conversión de representación		55
cuasi declaración		139
cuasi declaración de identidad-loc		139
cuasi declaración de localización		139
cuasi definición de señal		140
cuasi definición de sinónimo		139
cuasi lista de parámetros formales		139
cuasi parámetro formal		139
cuasi sentencia de datos	10.10.3	127, **139**
cuasi sentencia de declaración		139
cuasi sentencia de definición		139
cuasi sentencia de definición de procedimiento		139
cuasi sentencia de definición de proceso		139
cuasi sentencia de definición de señal	10.10.3	139, **140**
cuasi sentencia de definición de sinónimo		139
cuerpo de contexto	10.2	**127**, 136, 138
cuerpo de modo módulo		39
cuerpo de modo región		41
cuerpo de modo tarea		43
cuerpo de módulo	10.2	**127**, 134, 141
cuerpo de módulo de espec	10.2	**127**, 138
cuerpo de principio-fin	10.2	**127**, 129
cuerpo de procedimiento	10.2	**127**, 129, 130
cuerpo de proceso	10.2	**127**, 134
cuerpo de región	10.2	**127**, 135, 141
cuerpo de región de espec	10.2	**127**, 138
D		
declaración		45
declaración de identidad-loc	4.1.3	45, **47**
declaración de localización		45
definición de modo	3.2.1	**13**, 14, 15
definición de procedimiento		129, 141
definición de procedimiento guardado		130
definición de procesamiento guardado	10.3	**130**
definición de proceso		134, 141
definición de señal		148
definición de sinónimo		54
descriptor desreferenciado	4.2.5	47, **49**
designador de pieza		136, 137

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
digit	2.2	**8**
dígito		8, 57, 114, 115
dígito hexadecimal	5.2.4.2	**57**, 61
dígito octal		57, 61
directiva		10
directiva de implementación		10
E		
ejemplificación de modo moreta genérico	10.11	**142**
ejemplificación de modo moreta genérico		38
ejemplificación de módulo genérico	10.11	134, **142**
ejemplificación de procedimiento genérico	10.11	**142**
ejemplificación de procedimiento genérico		129
ejemplificación de proceso genérico	10.11	**142**
ejemplificación de proceso genérico		134
ejemplificación de región genérica	10.11	**142**
elemento de arranque		50, 65
elemento de cadena	4.2.6	47, **50**
elemento de conjunto		18
elemento de conjunto numerado		18
elemento de formato		114
elemento de la derecha		50, 65
elemento de la izquierda	4.2.7	**50**, 65
elemento de lista e/s		113
elemento de matriz	4.2.8	47, **51**
elemento inferior		52, 66
elemento lista e/s		113
elemento superior		52, 66
enumeración conjuntista	6.5.2	83, **84**
enumeración de localización	6.5.2	83, **84**
enumeración de valor		83
enumeración por intervalo	6.5.2	83, **84**
enumeración por paso		83
espec de módulo		138
espec de parámetro		24, 130, 139
espec de procedimiento formal genérico		142
espec de región		138
espec de resultado	3.8	23, **24**, 129, 130, 139, 142, 163, 164, 165
espec remota	10.10.1	**136**, 138
especificación de etiqueta de caso	12.3	32, 71, 82, **167**
especificación de formato		114
especificación de modo módulo		39, 141
especificación de modo región		41, 141
especificación de modo tarea		43, 142
etiqueta de caso		167
exponente		58
expresión	5.3.2	51, 52, 61, 68, 70, **71**, 79, 98, 107, 110
expresión año		124
expresión arrancar	5.2.15	55, **69**, 91
expresión booleana		39, 71, 81, 86, 91, 130
expresión cadena		84, 98, 113
expresión cadena de caracteres		113
expresión coma flotante		97, 98, 113
expresión coma flotante		98
expresión condicional		71
expresión conjuntista		84, 97
expresión día		124
expresión discreta		82, 83, 84, 97, 98, 113
expresión dónde		107

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
expresión entera		50, 83, 97, 98, 120, 124
expresión escribir		110
expresión hora		124
expresión índice		107, 110, 112
expresión literal		139
expresión literal de coma flotante		21
expresión literal de entero		21
expresión literal discreta		19, 30, 32, 167
expresión literal entera		59
expresión literal entera		18, 19, 25, 27, 29, 35, 59, 77, 92
expresión matriz		84, 98
expresión mes		124
expresión minuto		124
expresión numérica	6.20.3	**98**
expresión parentizada	5.2.17	55, **70**
expresión segundo		124
expresión utilización		107
F		
factor de repetición		114
fin de línea		9
firma de procedimiento guardado		130
G		
generalidad		130
H		
herencia de interfaz		44
herencia de módulo		39
herencia de región		41
herencia de tarea		43
hexadecimal digit		57
I		
indicación de modo formal genérico		142
indicación de modo genérico formal		16
índice superior		30
inicialización		45
inicialización ligada a moreta	4.1.2	45, **46**
inicialización ligada al dominio	4.1.2	45, **46**
inicialización ligada al tiempo de vida	4.1.2	45, **46**
instanciación de región genérica		135
intervalo		61
intervalo de literal		19, 26
intervalo de valor float		21
intervalo literal	3.4.6	**19**, 167
irrelevante	12.3	**167**
irrelevante		167
iteración		83
L		
letra		8
límite float inferior		21
límite float superior		21
límite inferior		19
límite superior		19
lista de argumentos de e/s texto		112
lista de atributos de procedimiento	10.4	129, **130**, 139
lista de atributos de procedimientos componentes en línea		130

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
lista de atributos de procedimientos componentes simples		130
lista de atributos de procedimientos guardados		130
lista de atributos de procedimientos guardados		130
lista de cláusulas amigo		164
lista de conjunto		18
lista de conjunto no numerada		18
lista de conjunto numerada		18
lista de contextos		134, 135, 138, 141
lista de e/s	7.5.3	**113**
lista de etiquetas de caso		62, 167
lista de eventos		92
lista de excepciones	3.8	23, **24**, 121, 129, 130, 139, 142
lista de excepciones		130
lista de expresiones		51, 66, 98
lista de expresiones literales		32
lista de intervalos	6.4	81, **82**
lista de localizaciones		95
lista de marcadores		32
lista de modos formales genéricos		142
lista de nombres amigo	12.2.3.4	**164**
lista de nombres de campo		62
lista de nombres de prohibición		164
lista de ocurrencia de definición de nombre de campo		32
lista de ocurrencias de definición	2.7	**10**, 13, 45, 47, 54, 95, 130, 139, 142, 143
lista de ocurrencias de definición de nombre de campo	2.7	**10**, 32
lista de parámetros		130
lista de parámetros	3.8	23, **24**, 164
lista de parámetros asociar		105
lista de parámetros de rutina incorporada		87
lista de parámetros efectivos		38, 69, 87
lista de parámetros efectivos constructor		46, 101
lista de parámetros efectivos genéricos		142
lista de parámetros efectivos genéricos		142
lista de parámetros formales	10.4	129, **130**, 134, 142, 163, 165
lista de parámetros formales genéricos		142
lista de parámetros modificar	7.4.5	106, **107**
lista de selectores de caso	6.4	71, 81, **82**
lista de sentencias de acción		81 82, 83, 92, 95, 96, 121, 122, 123, 127
lista de sentencias de datos		127
lista de sinónimos formales genéricos		142
lista e/s		112
literal	5.2.4.1	55, **56**
literal binario de cadena de bits		61
literal booleano	5.2.4.4	56, **58**
literal coma flotante	5.2.4.3	56, **58**
literal coma flotante con signo		58, 77
literal coma flotante sin signo		58
literal de cadena de bits	5.2.4.9	56, **61**
literal de cadena de caracteres	5.2.4.8	56, **60**, 137
literal de cadena de caracteres anchos		60
literal de cadena de caracteres estrechos		60
literal de carácter	5.2.4.5	56, **59**
literal de carácter ancho		59
literal de carácter estrecho		59
literal de conjunto	5.2.4.6	56, **59**

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
literal de vacío	5.2.4.7	56, **60**
literal entero	5.2.4.2	56, **57**
literal entero binario		57
literal entero con signo		57, 77
literal entero decimal		57
literal entero hexadecimal		57
literal entero octal		57
literal entero sin signo		57
literal hexadecimal de cadena de bits		61
literal octal de cadena de bits		61
llamada a procedimiento		87
llamada a procedimiento componente moreta		87
llamada a procedimiento que entrega un valor	5.2.13	55, **69**
llamada a procedimiento que entrega una localización		47, **53**
llamada a rutina e/s asociado		105
llamada a rutina incorporada		87
llamada a rutina incorporada asociar		105
llamada a rutina incorporada atribuir		97, **101**
llamada a rutina incorporada CHILL	6.20	**97**
llamada a rutina incorporada CHILL que entrega un valor		97
llamada a rutina incorporada CHILL que entrega una localización		97
llamada a rutina incorporada CHILL simple		97
llamada a rutina incorporada conectar	7.4.6	105, **107**
llamada a rutina incorporada de e/s que entrega un valor	7.4.1	97, **105**
llamada a rutina incorporada de e/s que entrega una localización	7.4.1	97, **105**
llamada a rutina incorporada desconectar	7.4.7	105, **109**
llamada a rutina incorporada disociar	7.4.3	105, **106**
llamada a rutina incorporada e/s asociado	7.4.2	**105**
llamada a rutina incorporada escribir registro	7.4.9	105, **110**
llamada a rutina incorporada establecer texto	7.5.8	105, **120**
llamada a rutina incorporada leer registro	7.4.9	105, **110**
llamada a rutina incorporada modificación	7.4.5	105, **106**
llamada a rutina incorporada obtener texto	7.5.8	105, **120**
llamada a rutina incorporada que entrega un atributo de acceso	7.4.8	105, **109**
llamada a rutina incorporada que entrega un atributo de asociación	7.4.4	105, **106**
llamada a rutina incorporada que entrega un tiempo absoluto		124
llamada a rutina incorporada que entrega un valor	5.2.14	55, **69**
llamada a rutina incorporada que entrega un valor de tiempo	9.4	97, **124**
llamada a rutina incorporada que entrega una duración		124
llamada a rutina incorporada que entrega una localización	4.2.12	47, **53**
llamada a rutina incorporada simple de e/s	7.4.1	97, **105**
llamada a rutina incorporada simple que entrega una temporización	9.4.3	97, **125**
llamada a rutina incorporada terminar	6.20.4	97, **101**
llamada a rutina incorporada texto	7.5.3	105, **112**
llamada a una rutina incorporada CHILL que entrega un valor	6.20.2	**97**
llamada a una rutina incorporada que entrega un		69

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
valor		
localización		47, 55, 78, 79, 87, 90, 95, 96, 97, 105, 107
localización texto		120
localización acceso		110, 120
localización acceso		98
localización acesso		107
localización almacenamiento		110
localización año		125
localización asociación		105, 106, 107
localización cadena		50, 84, 98, 113
localización cadena de caracteres		113, 120
localización coma flotante		113
localización coma flotante		98
localización día		125
localización discreta		98, 113
localización ejemplar		92, 95, 96
localización entera		125
localización estructura		53, 86
localización evento		92, 98
localización hora		125
localización matriz		51, 52, 84, 98
localización mes		125
localización minuto		125
localización modo estático		54, 110
localización moreta		11, 87
localización moreta predefinida	4.2.14	**54**
localización moreta predefinida		47
localización referenciada		78
localización segundo		125
localización suceso		92
localización tampón		94, 96, 98
localización texto		98, 107, 113, 120
localización texto		98
localización transferencia		107, 109
longitud		35
longitud de cadena		29
longitud de evento		25
longitud de tampón		25
longitud de texto		27
M		
manejador	8.2	39, 41, 43, 46, 47, 79, **121**, 129, 130, 134, 135, 141
manejador de temporización		122, 123
modo	3.3	13, **16**, 22, 24, 25, 26, 30, 32, 45, 47, 54, 139, 140, 142, 148
modo acceso		26
modo asociación		26
modo booleano		17
modo cadena		23, 29
modo cadena parametrizado		29
modo carácter	3.4.4	17, **18**
modo coma flotante		20
modo compuesto	3.13.1	16, **29**
modo conjuntista	3.6	16, **22**
modo conjunto	3.4.5	17, **18**
modo definidor		13
modo descriptor	3.7.4	22, **23**
modo discreto	3.4.1	16, **17**, 22, 26

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
modo duración		28
modo ejemplar	3.9	16, **24**
modo elemento		30
modo elemento tampón		25
modo entero		17
modo entrada-salida	3.11.1	16, **26**
modo estructura	3.13.4	29, **32**
modo estructura parametrizada		32
modo estructura variable		23
modo evento		25
modo formal genérico		142
modo índice		26, 27, 30
modo interfaz	3.15.5	38, **44**, 142
modo intervalo de coma flotante	3.5.2	20, **21**
modo intervalo discreto	3.4.6	17, **19**
modo matriz	3.13.3	23, 29, **30**
modo matriz parametrizado		30
modo miembro		22
modo módulo	3.15.2	38, **39**
modo moreta	3.15.1	29, **38**
modo no compuesto		16
modo procedimiento	3.8	16, **23**
modo real	3.5	16, **20**
modo referencia	3.7.1	16, **22**
modo referencia libre	3.7.3	22, **23**
modo referencia ligada		22
modo referenciado		22
modo región	3.15.3	38, **41**
modo registro		26
modo sincronización	3.10.1	16, **25**
modo tampón		25
modo tarea	3.15.4	38, **43**
modo temporización	3.12.1	16, **28**
modo texto	3.11.4	26, **27**
modo texto ancho		27
modo texto estrecho		27
modo tiempo absoluto		28
modulión remoto	10.10.1	134, 135, **136**
módulo	10.6	79, **134**, 135
módulo de contexto	10.10.1	79, **137**
módulo de espec	10.10.2	79, 127, 135, **138**
módulo de espec simple		138
módulo de herencia	3.15.2	**39**
N		
nombre	2.7	**10**
nombre amigo		164
nombre calificado	2.7	10, **11**
nombre de acceso	4.2.2	47, **48**
nombre de cadena simple		43
nombre de campo	2.7	**10**, 53, 62, 67, 164
nombre de campo marcador		32
nombre de componente	2.7	**11**
nombre de componente moreta	2.7	10, **11**
nombre de elemento de conjunto	2.7	**10**, 59
nombre de enumeración de localización		48
nombre de enumeración de valor		56
nombre de etiqueta		86, 90
nombre de excepción	2.7	**11**, 24, 91
nombre de identidad-loc		48

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
nombre de literal booleano		58
nombre de literal de vacío	5.2.4.7	**60**
nombre de localización		48
nombre de localización hacer-con		48
nombre de modo		48, 49, 54, 59, 61, 68, 98
nombre de modo acceso		26
nombre de modo acceso		98
nombre de modo asociación		26
nombre de modo booleano		17
nombre de modo cadena		29, 98
nombre de modo cadena origen		29
nombre de modo cadena parametrizado		29
nombre de modo carácter		18
nombre de modo coma flotante		20, 21
nombre de modo coma flotante		98
nombre de modo conjuntista		22
nombre de modo conjunto		18
nombre de modo descriptor		23
nombre de modo discreto		19, 82, 84, 98, 167
nombre de modo duración		28
nombre de modo entero		17
nombre de modo estructura		32
nombre de modo estructura parametrizada		32
nombre de modo estructura variable		32, 98
nombre de modo estructura variable origen		32
nombre de modo evento		25, 98
nombre de modo instancia		24
nombre de modo interfaz		39, 44
nombre de modo intervalo de coma flotante		21
nombre de modo intervalo discreto		19
nombre de modo matriz		30, 98
nombre de modo matriz origen		30
nombre de modo matriz parametrizado		30
nombre de modo modulión o moreta	12.2.3.4	**164**
nombre de modo módulo		39, 41, 43
nombre de modo moreta		38, 142, 164
nombre de modo moreta genérico		142
nombre de modo procedimiento		23
nombre de modo referencia libre		23
nombre de modo referencia ligada		22
nombre de modo región		41
nombre de modo tampón		25, 98
nombre de modo tarea		43
nombre de modo texto		98
nombre de modo texto		98
nombre de modo tiempo absoluto		28
nombre de modulión		164
nombre de modulión o de modo moreta		164
nombre de módulo genérico		142
nombre de procedimiento		87, 143, 164
nombre de procedimiento general		56
nombre de procedimiento genérico		142
nombre de procedimiento o proceso amigo		164
nombre de proceso		69, 148, 164
nombre de proceso genérico		142
nombre de referencia de texto	2.7	**11**, 137
nombre de región genérica		142
nombre de rutina incorporada		87
nombre de señal		93, 95
nombre de sinónimo		56

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
nombre de sinónimo indefinido		70
nombre de valor	5.2.3	55, **56**
nombre de valor a recibir		56
nombre de valor hacer-con		56
nombre no reservado		87
O		
objeto compuesto		84
ocurrencia de definición	2.7	**10**, 79, 83, 96, 129, 130, 134, 135, 139, 140, 141, 148
ocurrencia de definición de nombre de campo	2.7	**10**
ocurrencia de definición de nombre de componente	2.7	**11**
ocurrencia de definición de nombre de elemento de conjunto	2.7	**10**, 18
operador aritmético aditivo		74, 80
operador aritmético multiplicativo	5.3.7	75, **76**, 80
operador cero-ádico	5.2.16	55, **70**
operador de asignación		79
operador de concatenación de cadena		74, 80
operador de diferencia conjuntista		74, 80
operador de inclusión conjuntista		73
operador de pertenencia		73
operador de repetición de cadena		77
operador diádico cerrado	6.2	79, **80**
operador exponenciación	5.3.9	76, **77**
operador monádico		77
operador relacional		73
operador–3		73
operador–4		74
operando–0	5.3.3	71, **72**
operando–1		72
operando–2	5.3.5	72, **73**
operando–3	5.3.6	73, **74**
operando–4	5.3.7	74, **75**
operando–5	5.3.9	75, **76**
operando–6	5.3.9	76, **77**
operando–7	5.3.10	77, **78**
organización de campo	3.13.5	32, **35**
organización de elementos	3.13.5	30, **35**
P		
palabra		35
parámetro asociar		105
parámetro de rutina incorporada		87
parámetro efectivo		87
parámetro efectivo genérico	10.11	142, **143**
parámetro formal		130
parámetro formal genérico		142
parámetro modificar		107
parte afirmación		130
parte afirmación	10.3	**130**
parte con	6.5.4	83, **86**
parte de control		83
parte genérica		141, 142
parte invariable		39, 41, 43
paso		35
plantilla	10.11	127, 135, **141**
plantilla de modo de interfaz genérico		141
plantilla de modo de módulo genérico		141

	Subcláusula de definición utilizada en subcláusula	página(s)
plantilla de modo de región genérico		141
plantilla de modo de tarea genérico		141
plantilla de modo interfaz genérico	10.11	**142**
plantilla de modo módulo genérico	10.11	**141**
plantilla de modo región genérico	10.11	**141**
plantilla de modo tarea genérico	10.11	**142**
plantilla de módulo genérico		141
plantilla de procedimiento genérico		141
plantilla de procedimiento genérico	10.11	**141**
plantilla de proceso genérico		141
plantilla de proceso genérico	10.11	**141**
plantilla de región genérica		141
plantilla de región genérica	10.11	**141**
por ciento		114
pos		35
posfijo		162
posfijo de otorgamiento		162, 163
posfijo de toma		162, 165
prefijo		10, 162, 163, 165
prefijo antiguo		162
prefijo nuevo		162
prefijo simple		10
primer elemento		52, 66
prioridad		87, 92, 93, 94
procedimiento efectivo genérico		143
programa	10.8	**135**
R		
referencia libre desreferenciada	4.2.4	47, **49**
referencia ligada desreferenciada	4.2.3	47, **48**
región	10.7	127, **135**
región de espec	10.10.2	127, 135, **138**
región de espec simple		138
representación de conversión	5.2.12	**68**
resultado		90
S		
secuencia de control		59, 60
secuencia de dígitos		57, 58
segmento de cadena	4.2.7	47, **50**
segmento de matriz	4.2.9	47, **52**
sentencia de acción	6.1	**79**, 127
sentencia de concesión		41
sentencia de datos		127
sentencia de declaración		39, 41, 45, 127
sentencia de declaración moreta		135
sentencia de definición		127
sentencia de definición de neomodo	3.2.3	**15**, 39, 127, 139, 143
sentencia de definición de neomodo moreta		135
sentencia de definición de procedimiento	10.4	127, **129**
sentencia de definición de procedimiento guardado	10.4	**130**
sentencia de definición de procedimiento guardado en línea		39
sentencia de definición de procedimiento guardado simple		39, 41
sentencia de definición de proceso	10.5	39, 127, **134**
sentencia de definición de señal	11.5	39, 41, 45, 127, 139, **148**
sentencia de definición de sínmodo	3.2.2	**14**, 39, 127, 139, 143
sentencia de definición de sínmodo moreta		135

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
sentencia de definición de sinónimo	5.1	39, **54**, 127, 139, 143
sentencia de especificación de proceso		39, 45
sentencia de firma de procedimiento guardado	10.3	**130**
sentencia de firma de procedimiento guardado simple		39, 41, 45
sentencia de otorgamiento	12.2.3.4	39, 162, **163**
sentencia de toma	12.2.3.5	39, 142, 162, **165**
sentencia de visibilidad	12.2.3.2	127, **162**
símbolo de asignación	6.2	46, 47, 79, **80**, 83
sinónimo formal genérico		142
sinónimo formal genérico	10.11	**142**
subexpresión		71
suboperando–0		72
suboperando–1		72
suboperando–2		73
suboperando–3		74
suboperando–4		75
suboperando–5		76
T		
tamaño de paso		35
tamaño de segmento		50, 52, 65, 66
texto de formato		114
tipo de cadena		29
tupla	5.2.5	55, **61**
tupla conjuntista		61
tupla de estructura	5.2.5	61, **62**
tupla de estructura etiquetada		62
tupla de estructura no etiquetada		62
tupla de matriz		61
tupla de matriz etiquetada	5.2.5	61, **62**
tupla de matriz no etiquetada		61
U		
unidad de programa remota		15
unidad de programa remoto	10.10.1	14, **137**, 141
V		
vacía		91, 127, 139
vacío		137, 162
valor	5.3.1	46, 61, 62, **70**, 79, 87, 90, 93, 94, 101, 105, 107
valor campo de estructura	5.2.10	55, **67**
valor constante		46, 54, 139
valor de paso		83
valor elemento de cadena	5.2.6	55, **65**
valor elemento de matriz	5.2.8	55, **66**
valor final	6.5.2	83, **84**
valor indefinido		70
valor inicial		83
valor primitivo	5.2.1	**55**, 78
valor primitivo de cadena		65
valor primitivo de descriptor		49
valor primitivo de duración		122, 123
valor primitivo de ejemplar		93
valor primitivo de estructura		67, 86
valor primitivo de matriz		66
valor primitivo de procedimiento		87
valor primitivo de referencia		101
valor primitivo de referencia libre		49

	Subcláusula de definición utilizada en	
	subcláusula	**página(s)**
valor primitivo de referencia ligada		48
valor primitivo localización moreta de referencia ligada		87
valor primitivo tiempo absoluto		123, 125
valor segmento de cadena	5.2.7	55, **65**
valor segmento de matriz	5.2.9	55, **66**
ventana de otorgamiento		163
ventana de toma		165

SERIES DE RECOMENDACIONES DEL UIT-T

Serie A	Organización del trabajo del UIT-T
Serie B	Medios de expresión: definiciones, símbolos, clasificación
Serie C	Estadísticas generales de telecomunicaciones
Serie D	Principios generales de tarificación
Serie E	Explotación general de la red, servicio telefónico, explotación del servicio y factores humanos
Serie F	Servicios de telecomunicación no telefónicos
Serie G	Sistemas y medios de transmisión, sistemas y redes digitales
Serie H	Sistemas audiovisuales y multimedios
Serie I	Red digital de servicios integrados
Serie J	Transmisiones de señales radiofónicas, de televisión y de otras señales multimedios
Serie K	Protección contra las interferencias
Serie L	Construcción, instalación y protección de los cables y otros elementos de planta exterior
Serie M	RGT y mantenimiento de redes: sistemas de transmisión, circuitos telefónicos, telegrafía, facsímil y circuitos arrendados internacionales
Serie N	Mantenimiento: circuitos internacionales para transmisiones radiofónicas y de televisión
Serie O	Especificaciones de los aparatos de medida
Serie P	Calidad de transmisión telefónica, instalaciones telefónicas y redes locales
Serie Q	Conmutación y señalización
Serie R	Transmisión telegráfica
Serie S	Equipos terminales para servicios de telegrafía
Serie T	Terminales para servicios de telemática
Serie U	Conmutación telegráfica
Serie V	Comunicación de datos por la red telefónica
Serie X	Redes de datos y comunicación entre sistemas abiertos
Serie Y	Infraestructura mundial de la información y aspectos del protocolo Internet
Serie Z	**Lenguajes y aspectos generales de soporte lógico para sistemas de telecomunicación**

Ginebra, 2002