UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.200

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(11/99)

SÉRIE Z: LANGAGES ET ASPECTS INFORMATIQUES GÉNÉRAUX DES SYSTÈMES DE TÉLÉCOMMUNICATION

Langages de programmation – CHILL: le langage de haut niveau de l'UIT-T

CHILL – Le langage de programmation de l'UIT-T

Recommandation UIT-T Z.200

(Antérieurement Recommandation du CCITT)

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS INFORMATIQUES GÉNÉRAUX DES SYSTÈMES DE TÉLÉCOMMUNICATION

TECHNIQUES DE DESCRIPTION FORMELLE	
Langage de description et de spécification (SDL)	Z.100–Z.109
Application des techniques de description formelle	Z.110–Z.119
Diagrammes des séquences de messages	Z.120–Z.129
LANGAGES DE PROGRAMMATION	
CHILL: le langage de haut niveau de l'UIT-T	**Z.200–Z.209**
LANGAGE HOMME-MACHINE	
Principes généraux	Z.300–Z.309
Syntaxe de base et procédures de dialogue	Z.310–Z.319
LHM étendu pour terminaux à écrans de visualisation	Z.320–Z.329
Spécification de l'interface homme-machine	Z.330–Z.399
QUALITÉ DES LOGICIELS DE TÉLÉCOMMUNICATION	Z.400–Z.499
MÉTHODES DE VALIDATION ET D'ESSAI	Z.500–Z.599

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

NORME INTERNATIONALE 9496

RECOMMANDATION UIT-T Z.200

CHILL – LE LANGAGE DE PROGRAMMATION DE L'UIT-T

Résumé

La présente Recommandation | Norme internationale définit le langage de programmation de l'UIT-T appelé CHILL. Il s'agit d'un langage fortement typé, structuré en blocs et orienté objet, initialement destiné à l'implémentation des systèmes incorporés larges et complexes.

Ce langage a été conçu pour être fiable et rapide d'exécution, tout en étant suffisamment souple et puissant pour englober les applications nécessaires. Ses ressources favorisent également le développement modulaire de systèmes importants.

Source

La Recommandation UIT-T Z.200, élaborée par la Commission d'études 10 (1997-2000) de l'UIT-T, a été approuvée le 19 novembre 1999. Un texte identique est publié comme Norme internationale ISO/CEI 9496.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'UIT. Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes d'études à traiter par les Commissions d'études de l'UIT-T, lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution n° 1 de la CMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

NORME INTERNATIONALE
RECOMMANDATION UIT-T

CHILL – LE LANGAGE DE PROGRAMMATION DE L'UIT-T

1 Introduction

La présente Recommandation | Norme internationale définit le langage de programmation CHILL de l'UIT-T; le CHILL, qui est l'acronyme de CCITT High Level Language (langage de haut niveau du CCITT), a été introduit en 1980.

Les paragraphes suivants du présent article introduisent certaines motivations de la conception du langage et donnent une description de ses caractéristiques.

Pour de plus amples renseignements concernant le matériel d'introduction et d'entraînement sur ce sujet, le lecteur pourra consulter les manuels "Introduction to CHILL" et "CHILL user's manual".

Une autre définition du CHILL, de forme mathématique stricte (reposant sur la notation VDM) se trouve dans le manuel intitulé "Définition formelle du CHILL".

1.1 Généralités

Le CHILL est un langage fortement typé, structuré en blocs et conçu avant tout pour l'implémentation de grands systèmes complexes imbriqués.

Le CHILL est conçu de manière à:

- améliorer la fiabilité et augmenter l'efficacité des exécutables grâce à un grand nombre de contrôles effectués à la compilation;
- couvrir, être suffisamment souple et puissante pour couvrir la gamme d'applications nécessaire et à exploiter différents types de matériel;
- offrir les outils nécessaires à même de favoriser le développement progressif et modulaire des grands systèmes;
- répondre aux besoins des applications en temps réel grâce à des primitives intégrées de concomitance et de contrôle temporel;
- permettre la génération d'un code objet très efficace;
- être facile à apprendre et à utiliser.

La puissance d'expression qu'offre la conception du langage permet aux ingénieurs de choisir des constructions appropriées à partir d'une large gamme de facilités et de réaliser une implémentation pouvant correspondre plus précisément à la spécification d'origine.

Le CHILL faisant une nette distinction entre objets statiques et objets dynamiques, la quasi-totalité des contrôles sémantiques peuvent être faits lors de la compilation, ce qui présente des avantages évidents pour l'exécution. La violation des règles dynamiques du CHILL se traduit par des exceptions à l'exécution qui peuvent être interceptées par un système approprié de traitement des exceptions (toutefois, la génération de tels contrôles implicites est facultative, à moins qu'un tel système défini par l'utilisateur soit explicitement spécifiée).

Le CHILL permet d'écrire les programmes d'une façon indépendante de la machine. Le langage proprement dit est indépendant de la machine, mais certains systèmes de compilation peuvent exiger des objets définis spécifiquement pour l'implémentation. On notera que les programmes qui contiennent de tels objets ne sont en général pas portables.

1.2 Vue générale du langage

Un programme CHILL se compose essentiellement de trois parties:

- une description des objets;
- une description des actions à effectuer sur les objets;
- une description de la structure du programme.

Les objets sont décrits par des énoncés informatifs (énoncés déclaratifs et définissants), les actions sont décrites par des énoncés d'action et la structure du programme par des énoncés de structuration du programme.

Les objets manipulables du CHILL sont les valeurs et les locus où les valeurs peuvent être placées. Les actions définissent les opérations à effectuer sur les objets et l'ordre dans lequel les valeurs sont placées dans les locus et en sont extraites. La structure du programme détermine la durée de vie et la visibilité des objets.

Le CHILL prévoit un contrôle statique étendu sur l'emploi des objets dans un contexte donné.

Dans les paragraphes qui suivent, on récapitule les différents concepts du CHILL. Chaque paragraphe est une introduction à un article de même titre décrivant le concept en détail.

1.3 Modes et classes

A un locus est attaché un mode. Le mode d'un locus définit l'ensemble des valeurs que le locus peut contenir ainsi que d'autres propriétés associées au locus et aux valeurs qu'il peut contenir (à noter que toutes les propriétés d'un locus ne sont pas déterminées par son seul mode). Parmi les propriétés d'un locus, on trouve: taille, structure interne, protection, référenciation, etc. Parmi les propriétés d'une valeur, il y a: représentation interne, relation d'ordre, opérations permises, etc.

A une valeur est attachée une classe. La classe d'une valeur détermine les modes des locus qui peuvent contenir la valeur.

Le CHILL comporte les catégories de mode suivantes:

– modes discrets: modes entier, caractère, booléen, ensemble (symbolique) ainsi que leurs intervalles;

– modes réels: modes virgule flottante ainsi que leurs intervalles;

– modes ensemblistes: ensembles d'éléments d'un mode discret;

– modes référence: références liées, références libres et descripteurs utilisés comme références de locus;

– modes composites: modes chaîne, matrice et structure;

– modes procédure: procédures considérées comme objets manipulables;

– modes instance: identifications de processus;

– modes synchronisation: modes événement et tampon pour la synchronisation des processus et la communication;

– modes entrée-sortie: modes d'association d'accès et de texte pour les opérations d'entrée-sortie;

– modes temporisation: modes durée et temps absolu pour la temporisation

– modes moreta: modes module, région et tâche pour l'orientation objets avec héritage unique.

Le CHILL fournit des notations pour un ensemble de modes standard. Des modes définis par le programme peuvent être introduits au moyen de définitions de modes. Certaines constructions du langage ont ce qu'on appelle un mode dynamique. Il s'agit d'un mode dont certaines propriétés peuvent seulement être déterminées dynamiquement. Les modes dynamiques sont toujours des modes paramétrés avec des paramètres déterminés à l'exécution. Un mode non dynamique est un mode statique.

Les modes moreta permettent au CHILL d'assurer la programmation orientée objet d'une manière très polyvalente:

– modes module: les valeurs de ces modes se comportent dans une large mesure comme des modules, raison pour laquelle ils ressemblent beaucoup aux objets de la programmation classique orientée objet (p. ex. Smalltalk, C++, Eiffel ou Java);

– modes région: les valeurs de ces modes se comportent dans une large mesure comme des régions. Les objets de cette nature ne sont généralement pas utilisés dans la programmation orientée objet classique.

– modes tâche: les valeurs de ces modes ont essentiellement la même structure que les régions, mais elles ont leur propre sous procédure parallèle d'exécution de commande, et la communication entre elles et avec d'autres objets se fait de manière asynchrone.

Les classes n'ont pas de notations en CHILL. Elles sont introduites uniquement dans le métalangage pour décrire des conditions de contexte statiques et dynamiques.

1.4 Locus et leurs accès

Les locus sont des emplacements où des valeurs peuvent être placées et d'où elles peuvent être obtenues. Pour placer ou obtenir une valeur, il faut accéder au locus.

Les énoncés déclaratifs définissent les noms à employer pour accéder à un locus. Ce sont:

1) les déclarations de locus;

2) les déclarations d'identité de locus.

Les premiers créent des locus et établissent des noms d'accès aux locus nouvellement créés. Les seconds et les derniers établissent de nouveaux noms d'accès pour des locus créés ailleurs.

En dehors des déclarations de locus, de nouveaux locus peuvent être créés au moyen d'une opération prédéfinie *GETSTACK* ou *ALLOCATE*, qui rendra une valeur référence (voir ci-dessous) du locus nouvellement créé.

Un locus peut être **référençable**. Cela signifie qu'il correspond au locus une valeur référence de ce locus. Cette valeur référence est obtenue comme résultat de l'opération qui consiste à repérer le locus **référençable**. En déréférençant une valeur référence, on obtient le locus référencé. CHILL exige que certains locus soient toujours **référençables**; mais pour d'autres locus, on laisse l'implémentation décider s'ils sont **référençables** ou non. La propriété d'être ou non référençable doit, pour chaque locus, se déterminer statiquement.

Un locus peut être **protégé**, ce qui signifie qu'on ne peut y accéder que pour obtenir une valeur et non pour y placer de nouvelles valeurs (sauf à l'initialisation).

Un locus peut être composé, ce qui signifie qu'il est fait de sous-locus auxquels on peut accéder séparément. Un sous-locus n'est pas nécessairement **référençable**. Un locus contenant au moins un sous-locus **protégé** est dit posséder la **propriété de protection**. Les méthodes d'accès fournissant des sous-locus (ou sous-valeurs) sont: indexer et trancher pour les chaînes et les matrices, et sélectionner pour les structures.

A un locus est attaché un mode. Si ce mode est dynamique, le locus est appelé locus à mode dynamique.

Les propriétés suivantes des locus, bien qu'elles puissent être déterminées statiquement, ne font pas partie du mode:

référençabilité: une référence existe-t-elle ou non pour le locus?

classe de mémoire: est-il ou non alloué statiquement?

régionalité: est-il ou non déclaré à l'intérieur d'une région?

1.5 Valeurs et leurs opérations

Les valeurs sont des objets de base pour lesquels sont définies des opérations spécifiques. Une valeur est soit une valeur définie (au sens du CHILL), soit une **valeur non définie** (au sens du CHILL). L'utilisation d'une valeur non définie dans des contextes déterminés produit une situation indéfinie (au sens du CHILL) et le programme est considéré incorrect.

Le CHILL permet d'utiliser des locus dans des contextes où une valeur est requise; dans ce cas, un accès au locus est effectué pour obtenir la valeur qu'il contient.

A une valeur est attachée une classe. Les valeurs **fortes** sont les valeurs auxquelles, outre la classe, est attaché un mode. Dans ce cas, la valeur est toujours une des valeurs définies par ce mode. La classe est utilisée pour les contrôles de compatibilité et le mode pour la description des propriétés de la valeur. Certains contextes exigent que ces propriétés soient connues et une valeur **forte** est alors requise.

Une valeur peut être **littérale**, auquel cas elle dénote une valeur discrète, indépendante de l'implémentation et connue à la compilation. Une valeur peut être **constante**, auquel cas elle produit toujours la même valeur, c'est-à-dire qu'il n'est besoin de la calculer qu'une seule fois. Lorsque le contexte nécessite une valeur **constante** ou **littérale**, cette valeur est supposée être évaluée avant l'exécution et ne peut générer d'exceptions. Une valeur peut être **intrarégionale** auquel cas, elle peut référencer d'une façon ou d'une autre des locus déclarés dans une région. Une valeur peut être composée, c'est-à-dire contenir des sous-valeurs.

Les énoncés de définition de synonyme établissent de nouveaux noms dénotant des valeurs **constantes**.

1.6 Actions

Les actions constituent la partie algorithmique d'un programme CHILL.

L'action d'affectation place une valeur (calculée) dans un ou plusieurs locus. L'appel de procédure invoque une procédure, l'appel de routine prédéfinie invoque une routine prédéfinie (une routine prédéfinie est une procédure dont la définition n'est pas écrite en CHILL et qui a un mécanisme plus général de passage des paramètres et du résultat). Pour revenir d'un appel de procédure ou pour établir son résultat, les actions résulter et revenir sont utilisées.

Pour contrôler le déroulement en séquence des actions, CHILL fournit les actions de commande séquentielles suivantes:

– l'action conditionnelle: pour un branchement à deux voies;

– l'action à cas: pour un branchement multiple; le choix de la voie peut être basé sur plusieurs valeurs, comme pour une table de décision;

– l'action faire: pour une itération ou un parenthésage;

– l'action sortir: pour quitter une action parenthésée d'une façon structurée;

– l'action induire: pour induire une exception déterminée;

– l'action aller: pour un transfert inconditionnel à un point étiqueté d'un programme.

Les énoncés d'action et informatifs peuvent être groupés pour former un module ou un bloc début-fin, ce qui forme à nouveau une action (composée).

Pour contrôler les déroulements simultanés d'actions, le CHILL fournit les actions de cas démarrer, arrêter, mettre en attente, continuer, envoyer, attente, recevoir ainsi que les expressions et recevoir et démarrer.

1.7 Entrée et sortie

Les facilités d'entrée et de sortie du CHILL offrent le moyen de communiquer avec des dispositifs très divers du monde extérieur.

Le modèle référence entrée-sortie peut avoir trois états différents. A l'état libre, il n'y a pas d'interaction avec le monde extérieur.

L'opération *ASSOCIATE* permet d'entrer dans l'état de traitement de fichiers. Dans l'état de traitement de fichiers, il existe des locus de mode association, qui désignent des objets du monde extérieur. Il est possible, par des appels de routine prédéfinie, de lire et de modifier les attributs des associations définis par le langage, c'est-à-dire **existants**, **lisibles**, **écrivables**, **indexables**, **séquençables** et **variables**. La création et la suppression de fichiers sont aussi effectuées dans l'état traitement de fichiers.

L'opération *CONNECT* permet de connecter le locus de mode accès à un locus de mode association et d'entrer dans l'état de transfert de données. L'opération *CONNECT* permet de placer un indice de **base** dans un fichier. Dans l'état transfert de données, plusieurs attributs de locus de mode accès peuvent être inspectés et les opérations de transfert de données *READRECORD* et *WRITERECORD* peuvent être effectuées.

Pendant les opérations de transfert de texte, les valeurs sont représentées sous une forme assimilable par l'individu qui peut être transférée vers ou à partir d'un fichier ou d'un locus CHILL.

1.8 Traitement des exceptions

Les conditions sémantiques dynamiques du CHILL sont les conditions (liées au contexte) qui, en général, ne peuvent être vérifiées statiquement. (On laisse à l'implémentation le soin de décider d'engendrer ou non du code pour contrôler les conditions dynamiques à l'exécution, à moins qu'un filet approprié ne soit explicitement spécifié.) Le non-respect d'une règle sémantique dynamique cause une exception d'exécution; cependant, au cas où une implémentation peut déterminer statiquement qu'une condition dynamique va être non respectée, elle peut rejeter le programme.

Des exceptions peuvent également être causées par l'exécution d'une action causer ou, conditionnellement, par l'exécution d'une action affirmer. Quand, en un point donné du programme, une exception est causée, le contrôle est transmis au filet associé à cette exception, s'il est spécifié. On peut déterminer statiquement si un filet est ou non spécifié pour une exception en un point donné. Si aucun filet explicite n'est spécifié, le contrôle peut être transmis à un filet d'exception défini par l'implémentation.

Les exceptions ont un nom, qui est soit un nom d'exception prédéfini du CHILL, soit un nom d'exception défini par l'implémentation, soit un nom d'exception défini par le programme. Il faut noter que, lorsqu'un filet est spécifié pour un nom d'exception prédéfini par le CHILL, la condition dynamique associée doit être contrôlée.

1.9 Supervision temporelle

La supervision temporelle du CHILL permet de réagir au déroulement du temps dans le monde extérieur. Un processus devient **temporisable** quand il atteint un point clairement défini de l'exécution de certaines actions, et dès ce moment il peut être interrompu. Lorsque cela se produit, le contrôle est transféré à un filet approprié.

Les programmes peuvent détecter l'écoulement d'une période de temps ou peut se synchroniser sur un instant absolu ou à intervalles précis sans qu'il y ait cumul des dérives. Des routines prédéfinies sont prévues pour convertir le temps ou les durées en nombres entiers, pour mettre un processus en attente et pour détecter l'expiration d'une supervision temporelle.

1.10 Structure des programmes

Les énoncés de structuration de programme sont le bloc début-fin, le module, la procédure, le processus, la région et le mode moreta. Les énoncés de structuration de programme fournissent les moyens de contrôler la durée de vie des locus et la visibilité des noms.

La durée de vie d'un locus est le temps durant lequel un locus existe à l'intérieur du programme. Les locus peuvent être explicitement déclarés (dans une déclaration de locus) ou engendrés (appel aux routines prédéfinies *GETSTACK* ou *ALLOCATE*), ou ils peuvent être implicitement déclarés ou engendrés comme le résultat de l'utilisation de constructions du langage.

Un nom est dit **visible** en un certain point du programme s'il peut être utilisé en ce point. La portée d'un nom comprend tous les points où il est **visible**, c'est-à-dire où l'objet qu'il dénote est identifié par le nom.

Les blocs début-fin déterminent à la fois la visibilité des noms et la durée de vie des locus.

Les modules sont fournis pour restreindre la visibilité des noms afin de se protéger contre les utilisations non autorisées. Au moyen des énoncés de visibilité, il est possible d'exercer un contrôle sur la visibilité des noms dans diverses parties du programme.

Une procédure est un sous-programme (éventuellement paramétré) qui peut être invoqué (appelé) à différents endroits d'un programme. Elle peut rendre une valeur (procédure rendant valeur) ou un locus (procédure rendant locus), ou encore ne pas transmettre de résultat. Dans ce dernier cas, la procédure ne peut être appelée que dans une action d'appel de procédure.

Les processus, les locus des tâches, les régions et les locus des régions fournissent les moyens de réaliser une structure d'exécutions simultanées.

Des gabarits génériques donnent les moyens permettant de construire des modules génériques, des régions, des procédures, des processus et des modes moreta. Ils peuvent être paramétrés par des constantes, des modes et des procédures SYN. Les déclarations d'instanciation générique servent à obtenir des modules (non génériques), des régions, des procédures, des processus et des modes moreta qui sont appelés des instances génériques. Une instance générique est obtenue à partir d'un gabarit générique T par le remplacement, dans T, des paramètres génériques formels par les paramètres génériques réels correspondants.

Un programme CHILL complet est une liste d'unités de programme qui est considérée comme englobée dans une définition (imaginaire) de processus. Ce processus le plus externe est démarré par le système sous le contrôle duquel le programme est exécuté. Une unité de programme peut être un module, une région, une déclaration de définition synmode de moreta, une déclaration de définition "neomode" de moreta ou un gabarit générique.

Des constructions sont prévues pour faciliter différentes manières de développement de programme à partir de fragments. On utilise un module de spec et une région de spec pour définir les propriétés statiques d'un fragment de programme; un contexte sert à définir les propriétés statiques de noms saisis. De plus, il est possible de spécifier, par l'intermédiaire de la facilité éloignée, que le texte d'un fragment de programme se trouve ailleurs.

1.11 Exécution simultanée

Le CHILL prévoit l'exécution simultanée d'unités de programme. Le fil d'exécution (processus ou tâche) est l'unité d'exécution concomitante. L'évaluation d'une expression démarrer cause la création d'une nouvelle instance de la définition de processus indiquée. Ce processus est alors considéré comme exécuté en concomitance avec le fil d'exécution qui l'a démarré. CHILL prévoit qu'un ou plusieurs processus avec la même définition ou une définition différente peuvent être actifs en même temps. L'action arrêter, exécutée par un processus ou une tâche, termine ce processus.

Un fil d'exécution est toujours dans un des deux états suivants: il peut être soit actif soit en attente. La transition de l'état actif à l'état en attente est appelée mise en attente du fil d'exécution, la transition de l'état en attente à l'état actif est appelée la réactivation du fil d'exécution. L'exécution d'actions de mise en attente sur des événements, d'actions de réception sur des tampons ou signaux, d'actions envoyer sur des tampons, d'une action d'appel à une procédure de composante d'un locus de région, d'une action d'appel à une procédure de composante d'un locus de tâche au cas où il n'y a pas assez de stockage à faire peut mettre en attente le fil d'exécution qui les exécute. L'exécution d'une action continuer sur des événements, d'actions envoyer sur des tampons ou des signaux, d'actions recevoir sur des tampons, de libérer un locus de région ou au commencement de l'exécution d'une procédure de composante appelée de l'extérieur peut réactiver un fil d'exécution en attente.

Les tampons et les événements sont des locus à utilisation restreinte. Les opérations envoyer, recevoir et recevoir avec cas sont définies sur les tampons; les opérations mettre en attente, attente et continuer sont définies sur les événements. Les tampons sont des moyens de synchroniser les processus et de transmettre l'information entre eux. Les événements sont utilisés uniquement pour la synchronisation. Les signaux sont définis dans des énoncés de définitions de signaux. Ils dénotent des fonctions de composition et de décomposition de listes de valeurs transmises entre processus. Les actions envoyer et les actions recevoir avec cas prennent en charge la communication de la liste de valeurs ainsi que la synchronisation.

Une région est un module d'une espèce particulière. Elle fournit des moyens d'exclusion mutuelle pour les accès aux structures de données qui sont partagées par plusieurs fils d'exécution.

1.12 Propriétés sémantiques générales

Les conditions sémantiques (liées au contexte) du CHILL sont les conditions de compatibilité sur les modes et classes (vérification des modes) et les conditions de visibilité (vérification des portées). La vérification des modes détermine comment les noms peuvent être utilisés, la vérification des portées détermine où ils peuvent l'être.

Les règles de vérification des modes sont formulées en termes d'exigences de compatibilité entre modes, entre classes, et entre modes et classes. Les exigences de compatibilité entre modes et classes et entre classes elles-mêmes sont définies en termes de relations d'équivalence entre modes. Si des modes dynamiques sont impliqués, la vérification des modes est partiellement dynamique.

Les règles de portée définissent la visibilité des noms, déterminée par la structure du programme et par des énoncés explicites de visibilité. Ces derniers déterminent la visibilité des noms qui y sont mentionnés. Les noms introduits dans un programme ont un endroit où ils sont définis ou déclarés. Cet endroit est appelé l'occurrence de définition du nom. Les endroits où le nom est utilisé sont appelés occurrences d'utilisation du nom. Les règles d'identification associent une occurrence de définition unique à chaque occurrence d'utilisation d'un nom.

1.13 Options pour l'implémentation

Le CHILL permet des modes entier définis par l'implémentation, des routines prédéfinies par l'implémentation, des noms de **processus** définis par l'implémentation, des filets d'exceptions définis par l'implémentation et des noms d'exceptions définis par l'implémentation.

Un mode entier défini par l'implémentation doit être dénoté par un nom de **mode** défini par l'implémentation. Ce nom est considéré comme défini dans un énoncé de définition de néomode non spécifié en CHILL. Il est permis d'étendre aux modes entier définis par l'implémentation les opérations arithmétiques existantes prédéfinies par le CHILL, dans le cadre des règles syntaxiques et sémantiques du CHILL. Des exemples de modes entier définis par l'implémentation sont les entiers longs et les entiers courts.

Une opération prédéfinie est une procédure dont la définition n'est pas spécifiée en CHILL et qui peut avoir un système de passage de paramètres et de transmission du résultat plus général que les procédures CHILL.

Un nom de **processus** prédéfini est un nom de processus dont la définition n'est pas spécifiée en CHILL et qui peut avoir un système de passage de paramètres plus général que les processus CHILL. Un processus CHILL peut coopérer avec des processus définis par l'implémentation ou démarrer de tels processus.

Un filet d'exception défini par l'implémentation est un filet terminant la définition du processus. Si ce filet reçoit le contrôle après occurrence d'une exception, l'implémentation peut décider des actions à accomplir. Si une condition dynamique définie par l'implémentation est violée, il en résulte une exception définie par l'implémentation.

2 Préliminaires

2.1 Métalangage

La description du CHILL se compose de deux parties:

- la description de la syntaxe acontextuelle;
- la description des conditions sémantiques.

2.1.1 Description de la syntaxe acontextuelle

La syntaxe acontextuelle est décrite à l'aide du formalisme de Backus-Naur étendu (BNF, *Bekus-Maur form*). Les catégories syntaxiques sont indiquées par un ou plusieurs mots français, écrits en italiques, entre crochets angulaires (< et >). Cet indicateur est appelé symbole non terminal. Pour chaque symbole non terminal, une règle de production est donnée dans une section syntaxique correspondante. Une règle de production pour un symbole non terminal se compose du symbole non terminal à gauche du symbole ::=, et, à droite, d'une ou de plusieurs constructions composées de symboles terminaux ou non terminaux. Ces constructions sont séparées par une barre verticale (|) qui dénote différents choix de production pour le symbole non terminal.

Parfois, le symbole non terminal contient une partie soulignée. Cette dernière ne fait pas partie de la description acontextuelle, mais définit une catégorie sémantique (voir 2.1.2).

Les éléments syntaxiques peuvent être groupés entre accolades ({ et }). La répétition d'un groupe entre accolades est indiquée par un astérisque (*) ou un plus ($^+$). Un astérisque indique que le groupe est optionnel et peut être répété un nombre quelconque de fois; un signe plus indique que le groupe doit être présent et peut être répété un nombre quelconque de fois. Par exemple, $\{A\}^*$ remplace toute séquence de *A*, la séquence vide incluse, tandis que $\{A\}^+$ remplace toute séquence d'au moins un *A*. Si des éléments syntaxiques sont groupés entre crochets ([et]), le groupe est optionel. Un groupe entre accolades ou entre crochets peut contenir une ou plusieurs barres verticales, indiquant le choix entre des éléments syntaxiques.

Une distinction est faite entre syntaxe stricte, pour laquelle les conditions sémantiques sont données directement, et syntaxe dérivée. La syntaxe dérivée est considérée comme une extension de la syntaxe stricte et la sémantique pour la syntaxe dérivée est expliquée indirectement en termes de la syntaxe stricte associée.

Il est à noter que la description syntaxique acontextuelle est choisie de façon à convenir à la description sémantique dans la présente Recommendation | Norme internationale et non pour convenir à un algorithme d'analyse quelconque (par exemple, quelques ambiguïtés acontextuelles ont été introduites pour plus de clarté). Les ambiguïtés sont résolues par la référence à la catégorie sémantique des éléments syntaxiques.

2.1.2 Description sémantique

La description de chaque catégorie syntaxique (symbole non terminal), est organisée en sections intitulées **sémantique**, **propriétés statiques**, **propriétés dynamiques**, **conditions statiques** et **conditions dynamiques**.

La section **sémantique** décrit les concepts dénotés par les catégories syntaxiques (c'est-à-dire leur signification et leur comportement).

La section **propriétés statiques** définit les propriétés sémantiques de la catégorie syntaxique qui peuvent se déterminer statiquement. Ces propriétés sont utilisées dans la formulation des conditions statiques ou dynamiques dans les sections où la catégorie syntaxique est utilisée.

Si nécessaire, une section **propriétés dynamiques** définit les propriétés de la catégorie syntaxique qui ne sont connues que dynamiquement.

La section **conditions statiques** décrit les conditions dépendant du contexte contrôlables statiquement qui doivent être remplies lorsque la catégorie syntaxique est utilisée. Certaines conditions statiques sont exprimées dans la syntaxe au moyen d'une partie soulignée du symbole non terminal (voir 2.1.1). Cette utilisation exige que le non terminal soit d'une sous-catégorie sémantique spécifique. Par exemple, *expression booléenne* est identique à <*expression*> au sens acontextuel mais sémantiquement exige que l'*expression* soit d'une classe booléenne.

La section **conditions dynamiques** décrit les conditions qui dépendent du contexte et qui doivent être satisfaites au cours de l'exécution. Dans certains cas, les conditions sont statiques si des modes non dynamiques sont utilisés. Dans ces cas, la condition mentionnée à la rubrique **conditions statiques** est reprise sous la rubrique **conditions dynamiques**. Dans d'autres cas, les conditions dynamiques peuvent être vérifiées de manière statistique; une implémentation peut considérer qu'il s'agit là d'une violation d'une condition statique.

Dans la description sémantique, différentes polices de caractères sont utilisées: les caractères italiques (sans < et >) désignent des objets syntaxiques; les termes correspondants en caractères romains désignent les objets sémantiques correspondants (ex. *locus* désigne un locus). Les caractères gras sont utilisés pour indiquer des propriétés sémantiques; parfois, une propriété peut être exprimée à la fois syntactiquement et sémantiquement (ex. la phrase "l'*expression* est **constante**" a la même signification que "l'*expression* est une *expression* *constante*").

Sauf indication contraire, la sémantique, les propriétés et les conditions décrites dans la sous-section d'une catégorie syntaxique sont valables indépendamment du contexte dans lequel, dans les autres sections, cette catégorie syntaxique peut apparaître.

Les propriétés d'une catégorie syntaxique A qui a une règle de production de la forme *A::=B*, où *B* désigne une catégorie syntaxique, sont les mêmes que *B* sauf indication contraire.

Dans la présente Recommandation | Norme internationale, des noms virtuels ont été introduits pour décrire des modes, des locus ou des valeurs qui ne surviennent pas explicitement dans le texte du programme. Lorsque c'est le cas, le nom est précédé d'une esperluette (&). Ces noms ont une fonction purement descriptive.

2.1.3 Exemples

Pour la plupart des sections syntaxe, il y a une section intitulée **exemples** donnant un ou plusieurs exemples des catégories syntaxiques définies. Ces exemples font partie d'un ensemble d'exemples de programmes donnés à l'Appendice IV. Pour chaque exemple, on indique via quelle règle de syntaxe il est produit et dans quel exemple il a été pris.

Par exemple, *6.20 (d+5)/5* (*1.2*) montre un exemple de la chaîne terminale *(d+5)/5*, produite via la règle (*1.2*) de la section syntaxe correspondante, prise dans l'exemple de programme n° *6* ligne *20*.

2.1.4 Règles d'identification dans le métalangage

Parfois, la description sémantique mentionne des représentations textuelles de nom simple **spéciales** du CHILL (voir l'Appendice III). Ces chaînes de nom simple **spéciales** sont toujours utilisées avec leur signification CHILL et ne sont donc pas influencées par les règles d'identification d'un programme CHILL existant.

2.2 Vocabulaire

Les programmes sont représentés au moyen de l'ensemble de caractères CHILL (voir l'Appendice I), exception faite des littéraux à grands caractères, des littéraux à chaîne de grands caractères et des observations. La représentation d'un programme CHILL n'est pas spécifiée, ce qui signifie que l'on peut aussi utiliser une représentation à caractères multi-octets. L'alphabet CHILL est représenté par la catégorie syntaxique <*caractère*> à partir de laquelle tout caractère de l'ensemble de caractères CHILL peut être obtenu comme une production terminale. Les caractères du jeu UCS-2 niveau 1 sont représentés par la catégorie syntactique <*grand caractère*> à partir de laquelle tout caractère du jeu UCS-2 niveau 1 peut être obtenu comme une production terminale.

Les éléments lexicaux du CHILL sont:

- les symboles spéciaux;
- les noms;
- les littéraux.

Les symboles spéciaux sont énumérés dans l'Appendice II. Ils sont fournis d'un seul caractère ou d'une combinaison de caractères.

Les chaînes de noms simples sont formées d'après la syntaxe suivante:

syntaxe:

<*chaîne de nom simple*>*::=* *(1)*
<*lettre*> { <*lettre*> | <*chiffre*> | _ }* *(1.1)*

<*lettre*> *::=* *(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)*

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

sémantique: le caractère souligné (_) fait partie de la chaîne de nom simple, c'est-à-dire que la chaîne de nom simple *life_time* est différente de la chaîne de nom simple *lifetime*. Lettres majuscules et minuscules sont différentes, par exemple, *Status* et *status* sont deux chaînes de nom simple différentes.

Le langage possède un certain nombre de chaînes de nom simple **spéciales** à signification prédéterminée (voir l'Appendice III). Certaines d'entre elles sont **réservées**, c'est-à-dire qu'elles ne peuvent pas être utilisées pour d'autres usages.

Les chaînes de nom simple **spéciales** dans un fragment doivent être soit entièrement en représentation majuscule soit entièrement en représentation minuscule. Les chaînes de nom simple **réservées** le sont uniquement dans la représentation choisie (par exemple, si les minuscules sont choisies, **row** est réservé, **ROW** ne l'est pas).

conditions statiques: une *chaîne de nom simple* ne peut pas être une des chaînes de noms simples **réservées** (voir l'Appendice III.1).

2.3 Espacements

Il est permis d'insérer un ou plusieurs espaces avant et après chaque élément lexical. Une telle séquence s'appelle délimiteur. Les éléments lexicaux sont également terminés par le premier caractère qui ne peut pas en faire partie. Par exemple, *IFBTHEN* sera considéré comme *chaîne de nom simple* et non comme le début d'une action **IF** B **THEN**, //* sera considéré comme le symbole de concaténation (//) suivi d'un astérisque (*) et non comme un symbole de division (/) suivi du crochet ouvrant d'un commentaire (/*).

2.4 Commentaires

syntaxe:

<commentaire> ::= *(1)*
<commentaire parenthésé> *(1.1)*
| *<commentaire fin de ligne>* *(1.2)*

<commentaire parenthésé> ::= *(2)*
/* *<chaîne de caractères>* */ *(2.1)*

<commentaire fin de ligne> ::= *(3)*
–– *<chaîne de caractères> <fin de ligne>* *(3.1)*

<chaîne de caractères> ::= *(4)*
{ *<caractères>* }* *(4.1)*
| { *<grands caractères>* }* *(4.2)*

NOTE – *Fin de ligne* signifie la fin de la ligne dans laquelle intervient le commentaire.

sémantique: un *commentaire* donne de l'information au lecteur d'un programme. Il n'a pas d'influence sur la sémantique du programme.

Un *commentaire* peut être placé à tout endroit où des espaces peuvent servir à délimiter les éléments lexicaux.

Un *commentaire parenthésé* est terminé par la première occurrence de la séquence spéciale */. Un *commentaire de fin de ligne* se termine par la première occurrence de fin de ligne.

exemple:

4.1 /* *des algorithmes repris du CACM n° 93* */ *(2.1)*

2.5 Commandes de mise en page

Les commandes de mise en page (retour arrière) (BS, *backspace*), (retour de chariot) (CR, *carriage return*), (présentation de forme) (FF, *form feed*), (tabulation horizontale) (HT, *horizontal tabulation*), (interligne) (LF, *line feed*) et (tabulation verticale) (VT, *vertical tabulation*) de l'ensemble de caractères (voir l'Appendice I, FE_0 à FE_5) et la *fin de ligne* ne sont pas mentionnées dans la description de la syntaxe acontextuelle du CHILL. Quand elles sont utilisées, elles ont le même effet de délimitation qu'un espace. Les espaces et les commandes de mise en page ne peuvent pas être utilisés à l'intérieur d'éléments lexicaux (sauf littéraux de chaîne de caractères).

2.6 Directives au compilateur

syntaxe:

<clause de directive> ::= *(1)*
<> <directive> { , <directive> } <>* *(1.1)*

<directive> ::= *(2)*
<directive d'implémentation> *(2.1)*

sémantique: une clause de directive donne de l'information au compilateur. Cette information est spécifiée dans un format défini par l'implémentation.

Une directive d'implémentation ne doit pas influencer la sémantique d'un programme, c'est-à-dire qu'un programme contenant des directives d'implémentation est correct, au sens du CHILL, si et seulement si il est correct sans ces directives.

Une *clause de directive* se termine par la première occurrence du symbole de fin de directive (<>). Une *directive* peut contenir un caractère quelconque de l'ensemble de caractères (voir l'Appendice I).

propriétés statiques: une *clause de directive* peut s'insérer à tout endroit où des espaces sont admis. Elle a le même effet de délimitation qu'un espace. Les noms utilisés dans une *clause de directive* obéissent à un système de dérivation de nom défini par l'implémentation et qui n'influence pas les règles de dérivation de nom du CHILL (voir 12.2).

2.7 Noms et leurs occurences de définitions

syntaxe:

<nom> ::= *(1)*
<chaîne de nom> *(1.1)*
| *<nom qualifié>* *(1.2)*
| *<nom de composante moreta>* *(1.3)*

<chaîne de nom> ::= *(2)*
<chaîne de nom simple> *(2.1)*
| *<chaîne de nom préfixe>* *(2.2)*

<chaîne de nom préfixe> ::= *(3)*
<préfixe> ! <chaîne de nom simple> *(3.1)*

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

<préfixe simple> ::= *(5)*
<chaîne de nom simple> *(5.1)*

<occurence de définition> ::= *(6)*
<chaîne de nom simple> *(6.1)*

<liste d'occurences de définitions> ::= *(7)*
*<ocurrence de définitions> { , <ocurrence de définitions> }** *(7.1)*

<nom d'élément d'ensemble> ::= *(8)*
<chaîne de nom simple> *(8.1)*

<ocurrence de définition de nom d'élément d'ensemble> ::= *(9)*
<chaîne de nom simple> *(9.1)*

<nom de champ> ::= *(10)*
<chaîne de nom simple> *(10.1)*

<occurence de définition de nom de champ> ::= *(11)*
< chaîne de nom simple> *(11.1)*

<liste d'ocurences de définitions de noms de champ> ::= *(12)*
<occurence de définition de nom de champ>
*{ , <occurence de définition de nom de champ> }** *(12.1)*

<nom d'exception> ::= *(13)*
<chaîne de nom simple> *(13.1)*
| *<chaîne de nom préfixe>* *(13.2)*

<nom de référence de texte> ::= *(14)*
<chaîne de nom simple> *(14.1)*
| *<chaîne de nom préfixe>* *(14.2)*

<nom de composante> ::= *(15)*
<chaîne de nom simple> *(15.1)*

<occurence de définition de nom de composante> ::= *(16)*
<chaîne de nom simple> *(16.1)*

<nom qualifié> ::= *(17)*
<chaîne de nom simple > ! *<nom de composante>* *(17.1)*

<nom de composante moreta> ::= *(18)*
<locus de moreta> . { *<chaîne de nom simple >* | *<nom qualifié>* *(18.1)*

sémantique: les noms d'un programme désignent des objets. Etant donné l'occurrence d'un *nom* (formellement: l'occurrence d'une production terminale de *nom*) dans un programme, les règles de dérivation du 12.2 donnent les ocurrences de *définitions* (formellement: occurrences de productions terminales d'occurence de *définition*) auxquelles ce (cette occurrence de) *nom* est **identifié**. Ainsi, le *nom* désigne l'objet défini ou déclaré par les occurences de *définitions.* (Il ne peut y avoir plus d'une occurence de *définition* pour un *nom* que dans le cas de *noms* avec occurences de **quasi-***définitions* et dans le cas de *noms* de composantes de modes moreta.)

On dit des occurences de *définitions* qu'elles définissent le *nom*. On dit qu'un *nom* est une application du nom créé par *l'occurence de définition* à laquelle il est **identifié**. Le *nom* a sa *chaîne de nom simple* le plus à droite égale à celle du nom.

De même, les noms de champ sont **identifiés** aux occurences de définitions de nom de champ et désignent les champs (d'un mode structure) définis par ces occurences de définitions de nom de champ. Les *noms de composantes moreta* sont identifiés aux occurences de *définitions de composantes* et désignent les composantes (d'un mode moreta) définies par ces occurences de *définitions de nom de composante.*

Les noms d'exception sont utilisés pour identifier des filets d'exceptions selon les règles énoncées à l'article 8.

Les noms de référence de texte servent à identifier les fragments de texte source d'une manière définie par l'implémentation, sous réserve des règles énoncées au 10.10.1.

Lorsqu'un nom est **identifié** à plus d'une occurence de définition, chacune des occurences de définitions auxquelles le nom est **identifié** définit ou énonce le même objet (voir les règles exactes au 10.10 et 12.2.2).

Les *noms qualifiés* sont utilisés pour identifier des composantes de *modes moreta*.

définition de notation: soit une *chaîne de nom* NS, et une chaîne de caractères P, qui est un *préfixe* ou qui est vide, le résultat du préfixe NS avec P, écrit P ! NS, se définit de la manière suivante:

- si P est vide, P ! NS est NS;
- ou autrement, P ! NS est la chaîne de nom rattachée à la *chaîne de nom préfixée* obtenue par concaténation de tous les caractères de P, d'un opérateur de préfixage et de tous les caractères de NS.

Par exemple, si P est "*q ! r*" et NS est "*s ! n*", P ! NS est "*q ! r ! s ! n*".

propriétés statiques: a chaque *chaîne de nom simple* est attachée une chaîne de nom **canonique** qui est la *chaîne de nom simple* proprement dite. A une *chaîne de nom* est attachée une chaîne de nom **canonique** définie comme suit:

- si la *chaîne de nom* est une *chaîne de nom simple*, c'est la chaîne de nom **canonique** de cette *chaîne de nom simple*;
- si la *chaîne de nom* est une *chaîne de nom préfixe*, la concaténation reste dans l'ordre juste de toutes les *chaînes de nom simple* de la *chaîne de nom*, séparée par les opérateurs de préfixage, c'est-à-dire que les espaces, commentaires et commandes de mise en page (s'il en existe) sont omis.

Dans le reste de la présente Recommandation | Norme internationale:

- la chaîne de nom d'un *nom*, *nom d'exception*, ou le *nom de texte de référence*, est utilisée pour désigner la chaîne du nom **canonique** de la *représentation textuelle de nom* du *nom*, du *nom d'exception*, ou du *nom de référençage de texte*, respectivement;

- la chaîne de nom d'une *occurence de définition*, d'un *nom de champ*, d'une occurence de définition de *nom de champ*, d'un *nom de composante moreta* ou d'une *occurence de définition de composante moreta* sert à désigner la chaîne de nom **canonique** de la *chaîne de nom simple* dans cette *occurence de définition*, ce *nom de champ*, cette *occurence de définition de nom de champ*, ce *nom de composante moreta* ou d'*occurence de définition de composante*, respectivement.

Les règles de dérivation sont telles que:

- les noms appartenant à une chaîne de nom simple sont **identifiés** aux ocurences de définitions ayant la même chaîne;

- les noms appartenant à une chaîne de nom préfixée sont **identifiés** aux occurences de définitions d'une chaîne de nom identique à la chaîne de nom simple, la plus à droite, de la chaîne de nom préfixée du nom;

- les noms de champ sont **identifiés** à des occurences de définitions de nom de champ ayant la même chaîne de nom que les noms de champ;

- les *noms de composante moreta* sont **identifiés** aux occurences de *définitions de nom de composante moreta* ayant la même *chaîne* que les *noms de composante moreta*.

Un *nom* hérite toutes les propriétés statiques liées au nom défini par l'*occurence de définition* à laquelle il est **identifié**. Un *nom de champ* hérite toutes les propriétés statiques liées au nom de champ défini par l'*occurence de définition de nom de champ* à laquelle il est **identifié**. Un *nom de composante moreta* hérite toutes les propriétés statiques liées au *nom de composante moreta* défini par l'*occurence de définition de nom de composante moreta* à laquelle il est identifié.

conditions statiques: la *représentation textuelle de nom simple* désignée par un *nom qualifié* et suivie d'un ! doit être un *nom de mode moreta.*

Si un nom qualifié de la forme "M ! nom de composante" survient hors de la définition du mode moreta M, le nom de la composante doit être le nom d'une composante SYN, SYNMODE ou NEWMODE de M.

3 Modes et classes

3.1 Généralités

A un locus est attaché un mode, à une valeur, une classe. Le mode attaché à un locus définit l'ensemble des valeurs que le locus peut contenir, les méthodes d'accès au locus et les opérations permises sur les valeurs. La classe attachée à une valeur est un moyen de déterminer les modes des locus qui peuvent contenir la valeur. Certaines valeurs sont **fortes**. A une valeur **forte**, on attache une classe et un mode. Des valeurs **fortes** sont requises dans les contextes de valeur où une information de mode est nécessaire.

3.1.1 Modes

Le CHILL a des modes statiques (c'est-à-dire des modes dont on peut déterminer statiquement toutes les propriétés), et des modes dynamiques (c'est-à-dire des modes dont certaines propriétés sont seulement connues à l'exécution). Les modes dynamiques sont toujours des modes paramétrés dont les paramètres sont déterminés à l'exécution.

Les modes statiques sont notés dans le programme au moyen de productions terminales de la catégorie syntaxique *mode.*

Les modes sont également paramétrés par des valeurs non explicitement indiquées dans le texte du programme.

3.1.2 Classes

Les classes n'ont pas de notation en CHILL.

Les espèces suivantes de classes existent et toute valeur dans un programme CHILL a une classe d'une de ces espèces:

pour un mode M, il peut exister la M-classe par valeur. Toutes les valeurs d'une telle classe et seules ces valeurs sont **fortes** et le mode attaché à ces valeurs est M.

- Pour un mode M, il peut exister la M-classe par dérivation.
- Pour tout mode M, il existe la M-classe par référence.
- La classe **nulle**.
- La classe **toute**.

Les deux dernières sont des classes constantes, c'est-à-dire qu'elles ne dépendent pas d'un mode M. Une classe est dite dynamique si et seulement si c'est une M-classe par valeur, une M-classe par dérivation ou une M-classe par référence, où M est un mode dynamique.

3.1.3 Propriétés des modes, des classes et leurs relations

Les modes CHILL ont des propriétés. Celles-ci peuvent être héréditaires ou non héréditaires. Une propriété héréditaire est transmise d'un mode définissant à un nom de **mode** défini par celui-ci. Un résumé est donné ci-après des propriétés qui s'appliquent à tous les modes (sauf pour la première, elles sont toutes définies au 12.1):

- un mode a une **nouveauté** (définie aux 3.2.2, 3.2.3 et 3.3);
- un mode peut avoir la **propriété d'être protégé**;
- un mode peut être **paramétrable**;
- un mode peut avoir la **propriété de référencer**;
- un mode peut avoir la **propriété d'étiquetage et de paramétrage**;
- un mode peut avoir la **propriété de non-valeur**.

En CHILL, des classes peuvent avoir les propriétés suivantes (définies au 12.1):

- une classe peut avoir un mode **racine**;
- une ou plusieurs classes peuvent avoir une **classe résultante**.

En CHILL, les opérations sont déterminées par les modes et les classes des locus et des valeurs. Cela est exprimé par les règles de vérification des modes définies au 12.1 sous la forme d'un certain nombre de relations entre les modes et les classes. Les relations suivantes peuvent exister:

- deux modes peuvent être **similaires**;
- deux modes peuvent être **v-équivalents**;
- deux modes peuvent être **équivalents**;
- deux modes peuvent être **l-équivalents**;
- deux modes peuvent être **semblables**;
- deux modes peuvent être **identifiés par la nouveauté**;
- deux modes peuvent être **compatibles en lecture**;
- deux modes peuvent être **compatibles en lecture dynamique**;
- deux modes peuvent être **équivalents dynamiques**;
- un mode peut être **limitable** à un mode;
- un mode peut être **compatible** avec une classe;
- une classe peut être **compatible** avec une classe.

3.2 Définitions de modes

3.2.1 Généralités

syntaxe:

<définition de mode> ::= *(1)*
<liste d'occurences de définitions> = <mode définissant> *(1.1)*

<mode définissant> ::= *(2)*
<mode> *(2.1)*

syntaxe dérivée: une *définition de mode* où la *liste d'occurences de définitions* comporte plus d'une *occurence de définition* est dérivée de plusieurs définitions de mode, une pour chaque *occurence de définition*, séparées par des virgules, avec le même *mode définissant*. Par exemple:

NEWMODE *dollar, pound = INT*;

est dérivé de:

NEWMODE *dollar = INT, pound = INT*;

sémantique: une définition de mode définit un nom qui désigne le mode spécifié. Des définitions de mode apparaissent dans les énoncés de définition de synmode et de néomode. Une définition de synmode est **synonyme** de son mode définissant. Une définition de néomode n'est pas **synonyme** de son mode définissant. La différence est définie en fonction de la **nouveauté** de la propriété, qui est utilisée dans la vérification des modes (voir 12.1).

propriétés statiques: dans une *définition de mode*, une *occurence de définition* définit un nom de **mode**.

Les noms de **mode** prédéfinis et les noms de **mode** entier définis par l'implémentation (le cas échéant, voir 3.4.2 et 3.5.1) sont également des noms de **mode**.

Un nom de **mode** a un mode **définissant** qui est le *mode définissant* contenu dans la *définition de mode* qui le définit. (Pour les noms de **mode** prédéfinis et définis par l'implémentation, ce mode **définissant** est un mode virtuel.) Les propriétés héréditaires d'un nom de mode sont celles de son mode **définissant**.

Un ensemble de définitions récursives est un ensemble de définitions de modes ou de définitions de synonymes (voir 5.1) tel que le *mode définissant* dans chaque *définition de mode* ou la *valeur constante* ou le *mode* dans chaque *définition de synonyme* est, ou contient directement, un nom de **mode** ou un nom de **synonyme** défini par une définition dans l'ensemble.

Un ensemble de définitions de modes récursives est un ensemble de définitions récursives ne contenant que des définitions de modes.

Tout mode qui est ou qui contient un nom de **mode** défini dans un ensemble de définitions de modes récursives est dit désigner un mode récursif. Un chemin dans un ensemble de définitions de modes récursives est une liste de noms de **mode** où chaque nom est indiqué par un marqueur et telle que:

- tous les noms du chemin ont une définition différente;
- le successeur de chaque nom est ou apparaît directement dans le mode définissant de ce nom (le successeur du dernier nom est le premier);
- le marqueur indique précisément la position du nom dans le mode définissant de son prédécesseur (le prédécesseur du premier nom est le dernier).

[Exemple: **NEWMODE** *M* = **STRUCT** *(i M, n* **REF** *M)*; contient deux chemins: { M_i } et { M_n }.]

Un chemin est **sûr** si et seulement si au moins un de ses noms est contenu dans un *mode référence*, un *mode descripteur* ou un *mode procédure* à l'endroit marqué.

conditions statiques: pour tout ensemble de définitions de modes récursives, tous les chemins doivent être **sûrs**. (Le premier chemin de l'exemple ci-dessus n'est pas **sûr**.)

exemples:

1.15 *operand_mode = INT* *(1.1)*

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

3.2.2 Définitions de synmodes

syntaxe:

<énoncé de définition de synmode> ::= *(1)*
SYNMODE *<définition de mode>* { , *<définition de mode>* }*; *(1.1)*
| *<unité de programme à distance>* *(1.2)*

sémantique: les énoncés de définition de synmode définissent des noms dénotant des **modes** qui sont **synonymes** de leur mode définissant.

propriétés statiques: une *occurence de définition* figurant dans une *définition de mode* dans un *énoncé de définition de synmode* définit un nom de **synmode** (qui est aussi un nom de **mode**). Un nom de **synmode** est dit **synonyme** d'un mode M (et réciproquement, le mode donné est dit **synonyme** du nom de **synmode**) si et seulement si:

- le mode M est le mode **définissant** du nom de **synmode**;
- si le mode **définissant** du nom de **synmode** est lui-même un nom de **synmode**, **synonyme** du mode M.

Deux noms de mode A et B sont **synonymes** si et seulement si:

- A et B sont identiques;
- si A est le mode **définissant** de B et que B est un nom de **synmode**;
- si B est le mode **définissant** de A et que A est un nom de **synmode**;
- si le nom du mode **définissant** de A est **synonyme** de B et que A est un nom de **synmode**;
- si le nom du mode **définissant** de B est **synonyme** de A et que B est un nom de **synmode**;

La **nouveauté** d'un nom de **synmode** est celle de son mode **définissant**.

Si le mode **définissant** est un mode intervalle discret ou un mode intervalle à virgule flottante, le mode **parent** du nom **synmode** est celui de son mode **définissant**. Si le mode **définissant** est un mode chaîne **variable**, le mode **composant** du nom **synmode** est celui de son mode **définissant**.

exemple:

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

3.2.3 Définitions de néomodes

syntaxe:

<énoncé de définition de néomode> ::= *(1)*
NEWMODE *<définition de mode>* { , *<définition de mode>*}*; *(1.1)*
| *<unité de programme à distance>* *(1.2)*

sémantique: les énoncés de définition de néomode définissent des noms de **modes** qui ne sont pas **synonymes** de leur mode définissant.

propriétés statiques: une *occurence de définition* apparaissant dans une *définition de mode* apparaissant dans un *énoncé de définition de néomode* définit un nom de **néomode** (qui est aussi un nom de **mode**).

La **nouveauté** du nom de **néomode** est l'*occurence de définition* qui le définit. Si le mode **définissant** du nom de **néomode** est un mode intervalle discret ou un mode intervalle à virgule flottante, le mode virtuel *&nom* est introduit comme le mode **parent** du nom de **néomode**. Le mode **définissant** de *&nom* est le mode **parent** du mode intervalle discret ou celui du mode intervalle à virgule flottante et la **nouveauté** de *&nom* est celle du nom de **néomode**.

Si le mode **définissant** est un mode chaîne **variable**, le mode virtuel *&nom* est introduit comme le mode **composant** du nom de **néomode**. Le mode définissant de *&nom* est le mode **composant** du mode chaîne **variable** et la **nouveauté** du *&nom* est celle du nom de **néomode**.

Si l'*occurence de définition* de la définition de mode est une **quasi-***définition*, la **nouveauté** est une **quasi-nouveauté**, sinon c'est une **nouveauté réelle.**

conditions statiques: si la **nouveauté** est une **quasi-nouveauté**, une **nouveauté réelle** au moins doit **être identifiée** à sa **nouveauté**.

exemples:

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

11.12 **NEWMODE** *board* = **ARRAY** *(line)* **ARRAY** *(column) square*; *(1.1)*

3.3 Classification des modes

syntaxe:

<mode> ::= *(1)*
[**READ**] *<mode non composite>* *(1.1)*
| [**READ**] *<mode composite>* *(1.2)*
| *<indication de mode générique formelle>* *(1.3)*

<mode non composite> ::= *(2)*
<mode discret> *(2.1)*
| *<mode réel>* *(2.2)*
| *<mode ensembliste>* *(2.3)*
| *<mode référence>* *(2.4)*
| *<mode procédure>* *(2.5)*
| *<mode instance>* *(2.6)*
| *<mode synchronisation>* *(2.7)*
| *<mode entrée-sortie>* *(2.8)*
| *<mode temporisation>* *(2.9)*

sémantique: un mode définit un ensemble de valeurs et les opérations autorisées sur ces valeurs. Un mode peut être un mode **protégé**, indiquant qu'un locus de ce mode peut ne pas être accessible pour enregistrer une valeur. Un mode a une **nouveauté**, indiquant s'il a été introduit par l'intermédiaire d'un énoncé de définition de néomode ou non.

propriétés statiques: un mode a les propriétés héréditaires suivantes:

- il est un mode **protégé** s'il est explicitement ou implicitement **protégé**;
- il est un mode **protégé** explicitement si **READ** est spécifié ou s'il est un mode matrice **paramétré**, un mode chaîne **paramétré** ou un mode structure **paramétré** dans lequel le nom de mode matrice **d'origine**, le nom de mode chaîne **d'origine** ou le nom de mode structure **variable originel**, respectivement, est un mode **protégé**;
- il est un mode **protégé** implicitement s'il n'est pas un mode **protégé** explicitement et si:
  - c'est le mode **élément** d'un mode chaîne **protégé** ou d'un mode matrice **protégé** (voir 3.13.2 et 3.13.3);
  - c'est un mode **champ** d'un mode structure **protégé** ou le mode d'un champ **étiquette** d'un mode structure **paramétré** (voir 3.13.4).

Un *mode* a les mêmes propriétés que le *mode non composite* ou *mode composite* qu'il contient. Dans les paragraphes ci-après, les propriétés sont définies pour les noms de **mode** prédéfinis et pour les *modes* qui ne sont pas des *noms de mode*; les propriétés des *noms de mode* sont définies au 3.2. Les modes **protégés** ont les mêmes propriétés que leurs modes correspondants non **protégés**, à l'exception de la **propriété de protection** (voir 12.1.1.1).

Un mode a les propriétés non héréditaires suivantes:

- il a une **nouveauté** qui est soit **nulle** soit la *définition* existant dans une *définition de mode* figurant dans une déclaration *de définition de néomode*. La **nouveauté** d'un mode qui n'est pas un *nom de mode* (ni un *nom de mode* **READ**) est définie comme suit:
  - si c'est un mode chaîne **paramétré**, un mode rangée **paramétré** ou un mode structure **paramétré**, sa **nouveauté** est celle de son mode chaîne **d'origine**, de son mode matrice **d'origine** ou de son mode structure **variable originel**, respectivement;
  - si c'est un mode intervalle discret ou un mode intervalle à virgule flottante, sa **nouveauté** est celle de son mode **parent**;
  - sinon, sa **nouveauté** est **nulle**.

    La **nouveauté** d'un mode, c'est-à-dire d'un *nom de mode* (*nom de mode* **READ**) est définie aux 3.2.2 et 3.2.3.
- Il a une **taille**, c'est-à-dire la valeur livrée par *SIZE (&M)*, où *&M* est un nom de **synmode** virtuel **synonyme** du *mode*.

3.4 Modes discret

3.4.1 Généralités

syntaxe:

<mode discret> ::= *(1)*
<mode entier> *(1.1)*
| *<mode booléen>* *(1.2)*
| *<mode caractère>* *(1.3)*
| *<mode ensemble>* *(1.4)*
| *<mode intervalle discret>* *(1.5)*

sémantique: les modes discret définissent des ensembles et sous-ensembles de valeurs bien ordonnées.

3.4.2 Modes entier

syntaxe:

<mode entier> ::= *(1)*
<nom de mode entier> *(1.1)*

noms prédéfinis: le nom *INT* est prédéfini comme nom de **mode entier**.

sémantique: un mode entier définit un ensemble de valeurs entières avec signe, entre deux bornes définies par l'implémentation, sur lequel l'ordre et les opérations arithmétiques usuels sont définis (voir 5.3). Une implémentation peut définir d'autres modes entiers de bornes différentes (par exemple, *LONG_INT*, *SHORT_INT*, *UNSIGNED_INT*)) qui peuvent aussi être utilisés comme modes **parent** d'intervalles (voir 13.2). Le mode *&INT* est introduit en tant que mode virtuel contenant toutes les valeurs de tous les modes entiers **prédéfinis** définis par l'implémentation. La représentation interne d'une valeur entière est la valeur entière elle-même. On notera que *&INT* n'est pas un mode **prédéfini** (même s'il peut avoir les mêmes bornes que celles du mode entier **prédéfini**).

propriétés statiques: un mode entier a les propriétés héréditaires suivantes:

- la **borne supérieure** et la **borne inférieure** qui sont les littéraux dénotant respectivement la plus grande et la plus petite valeur définies par le mode entier. Elles sont définies par l'implémentation;
- le **nombre de valeurs**, qui est **borne supérieure** – **borne inférieure** + 1.

exemple:

1.5 INT *(1.1)*

3.4.3 Modes booléen

syntaxe:

<mode booléen> ::= *(1)*
<nom de mode booléen> *(1.1)*

noms prédéfinis: le nom *BOOL* est prédéfini comme nom de **mode booléen**.

sémantique: un mode booléen définit les valeurs logiques de vérité (*TRUE* et *FALSE*) avec les opérations booléennes usuelles (voir 5.3). Les représentations internes de *FALSE* et *TRUE* sont les valeurs entières 0 et 1, respectivement. La représentation définit aussi l'ordre des valeurs.

propriétés statiques: un mode booléen a les propriétés héréditaires suivantes:

- la **borne supérieure** qui est *TRUE*, la **borne inférieure** est *FALSE*;
- le **nombre de valeurs** qui est 2.

exemple:

5.4 BOOL *(1.1)*

3.4.4 Modes caractère

syntaxe:

<mode caractère> ::= *(1)*
<nom de mode caractère> *(1.1)*

noms prédéfinis: les noms *CHAR* et *WCHAR* sont prédéfinis comme noms de **mode caractère**.

sémantique: un mode caractère définit les valeurs caractère telles qu'elles sont décrites dans le jeu de caractères CHILL (voir l'Appendice I) dans le cas de *CHAR* et dans l'ISO/CEI 10646-1 dans le cas de *WCHAR*. Ces alphabets définissent également l'ordre des caractères et les valeurs entières qui sont leur représentation interne.

propriétés statiques: un mode caractère a les propriétés héréditaires suivantes:

- la **borne supérieure** et la **borne inférieure** d'un mode caractère qui sont les littéraux de chaîne de caractères dénotant respectivement la plus grande et la plus petite valeur définies respectivement par *CHAR et WCHAR*.
- un **nombre de valeurs** qui est 256 dans le cas de *CHAR* et qui est donné dans l'ISO/CEI 10646-1 dans le cas de *WCHAR*.

exemple:

8.4 CHAR *(1.1)*

3.4.5 Modes ensemble

syntaxe:

<mode ensemble> ::= *(1)*
SET (*<extension d'ensemble>*) *(1.1)*
| *<nom de mode ensemble>* *(1.2)*

<extension d'ensemble> ::= *(2)*
<extension d'ensemble avec numéros> *(2.1)*
| *<extension d'ensemble sans numéros>* *(2.2)*

<extension d'ensemble avec numéros> ::= *(3)*
<élément d'ensemble avec numéros> { , *<élément d'ensemble avec numéros>*}* *(3.1)*

<élément d'ensemble avec numéros> ::= *(4)*
<occurence de définition de nom d'élément d'ensemble> =
<expression de littéral entier> *(4.1)*

<extension d'ensemble sans numéros> ::= *(5)*
<élément d'ensemble> { , *<élément d'ensemble>*}* *(5.1)*

<élément d'ensemble> ::= *(6)*
<occurence définition de nom d'élément d'ensemble> *(6.1)*

sémantique: un mode ensemble définit un ensemble de valeurs nommées ou anonymes. Les valeurs nommées sont dénotées par les noms définis par les *occurences de définitions* apparaissant dans l'*extension d'ensemble*; les valeurs anonymes sont les autres valeurs. La représentation interne d'une valeur nommée est la valeur entière associée à la valeur nommée. Cette représentation définit également l'ordre des valeurs.

Le **nombre de valeurs** maximal d'un mode ensemble est défini par l'implémentation.

propriétés statiques: une *définition* d'une *extension d'ensemble* définit un nom d'**élément d'ensemble**. A un nom d'**élément d'ensemble** est attaché un mode **ensemble**, qui est le mode ensemble.

Un mode ensemble a les propriétés héréditaires suivantes:

- un ensemble de noms d'**élément d'ensemble** qui est l'ensemble de noms dans son *extension d'ensemble*;
- a tout nom d'**élément d'ensemble** d'un mode ensemble est attachée une valeur de représentation interne qui est, dans le cas d'un *élément d'ensemble avec numéros*, la valeur rendue par l'*expression de littéral entier* qu'il contient, sinon, une des valeurs 0, 1, 2, etc., d'après sa position dans l'*extension d'ensemble sans numéros*. Par exemple: **SET** *(a, b)*, à *a* est attachée la valeur de représentation 0 et à *b* la valeur de représentation 1;
- une **borne inférieure** et une **borne supérieure** qui sont ses noms d'**élément d'ensemble** et qui sont respectivement les valeurs nommées la plus petite et la plus grande;

- un **nombre de valeurs** qui est la plus grande des valeurs attachées aux noms d'**élément d'ensemble** augmentée de 1;
- c'est un mode ensemble **avec numéros**, si et seulement si l'*extension d'ensemble* est une *extension d'ensemble avec numéros*. Dans la négative, il s'agit d'un mode ensemble **sans numéros**.

conditions statiques: pour une quelconque paire d'*expressions de littéraux entiéres* e_1 et e_2 dans l'*extension d'ensemble NUM* (e_1) et *NUM* (e_2) doivent rendre des résultats non négatifs différents.

exemples:

11.7 **SET** *(occupied, free)* *(1.1)*

6.3 *month* *(1.2)*

3.4.6 Modes intervalle discret

syntaxe:

<*mode intervalle discret*> ::= *(1)*
<*nom de mode discret*> (<*intervalle littéral*>) *(1.1)*
| **RANGE** (<*intervalle littéral*>) *(1.2)*
| **BIN** (<*expression de littéral entier*>) *(1.3)*
| <*nom de mode intervalle discret*> *(1.4)*

<*intervalle littéral*> ::= *(2)*
<*borne inférieure*> : <*borne supérieure*> *(2.1)*

<*borne inférieure*> ::= *(3)*
<*expression littérale discrète*> *(3.1)*

<*borne supérieure*> ::= *(4)*
<*expression littérale discrète*> *(4.1)*

syntaxe dérivée: la notation **BIN** *(n)* est dérivée de **RANGE** *(0 : 2^n – 1)*, par exemple, **BIN** *(2+1)* tient lieu de **RANGE** *(0 : 7)*.

sémantique: un mode intervalle discret définit l'ensemble de valeurs de l'intervalle dont les bornes sont spécifiées (bornes incluses) par l'*intervalle littéral*. L'intervalle est pris dans un mode **parent** spécifique qui détermine les opérations et l'ordre définis sur les valeurs intervalle.

propriétés statiques: un mode intervalle discret a la propriété non héréditaire suivante: il a un mode **parent** unique, défini comme suit:

- si le mode intervalle discret est de la forme:

  <*nom de mode discret*> (<*intervalle littéral*>)

  si le nom de mode discret n'est pas un mode intervalle discret, le mode **parent** est le nom de mode discret; sinon, c'est le mode **parent** du nom de mode discret;

- si le mode intervalle est de la forme:

  RANGE *(<intervalle littéral>)*

  le mode **parent** dépend de la **classe résultante** des classes de la *borne supérieure* et de la *borne inférieure* de l'*intervalle littéral*.

  – s'il est une M-classe par dérivation, où M est un mode entier, le mode **parent** est un mode entier **prédéfini** choisi par l'implémentation de manière à contenir les intervalles de valeurs produites par l'*intervalle littéral*.

  – sinon il est le mode **racine** de la **classe résultante**.

- Si le mode intervalle est un *nom de mode intervalle discret* qui est un nom de **synmode**, son mode **parent** est celui du mode **définissant** du nom de **synmode**. Sinon, c'est un nom de **néomode** et son mode **parent** est le mode **parent** introduit virtuellement (voir 3.2.3).

Un mode intervalle discret a les propriétés héréditaires suivantes:

- une **borne supérieure** et une **borne inférieure** qui sont les littéraux dénotant les valeurs rendues respectivement par la *borne inférieure* et la *borne supérieure* de l'*intervalle littéral*;
- le **nombre de valeurs** d'un mode intervalle est la valeur rendue par *NUM (U) – NUM (L)* + 1, où *U* et *L* dénotent respectivement la **borne supérieure** et la **borne inférieure** du mode intervalle discret;
- c'est un mode intervalle **avec numéros**, si et seulement si son mode **parent** est un mode ensemble **avec numéros**.

conditions statiques: les classes de la *borne supérieure* et de la *borne inférieure* doivent être **compatibles** entre elles et **compatibles** avec le *nom de mode discret* si ce dernier est spécifié.

La *borne inférieure* doit rendre une valeur inférieure ou égale à la valeur rendue par la *borne supérieure*, et ces deux valeurs doivent appartenir à l'intervalle de valeurs défini par le *nom de mode discret*, s'il est spécifié.

L'*expression de littéral entier* dans le cas de **BIN** doit rendre une valeur non négative.

Si le mode **parent** est un mode entier, il doit exister un mode entier **prédéfini** qui contient l'ensemble de valeurs comprises entre la **borne inférieure** et la **borne supérieure.**

Si l'intervalle discret est de la forme:

RANGE (<*intervalle littéral*>) ou <*nom de mode discret*> (<*intervalle littéral*>)

l'évaluation de 1.*borne inférieure*, 2.*borne supérieure* ne doit pas dépendre directement ou indirectement de la valeur de 1.**borne inférieure**, 2.**borne supérieure** du mode intervalle discret. Si ce dernier est de la forme:

BIN (<*expression de littéral entier*>)

l'évaluation de *l'expression de littéral entier* ne doit pas dépendre directement ou indirectement de la valeur de la **borne supérieure** du mode intervalle discret.

exemples:

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

11.12 *line* *(1.4)*

3.5 Modes réels

syntaxe:

<*mode réel*> *::=* *(1)*
<*mode virgule flottante*> *(1.1)*
| <*mode intervalle à virgule flottante*> *(1.2)*

sémantique: un mode réel spécifie un ensemble de valeurs numériques qui se rapprochent d'un intervalle continu de nombres réels.

3.5.1 Modes virgule flottante

syntaxe:

<*mode virgule flottante*> *::=* *(1)*
<*nom de mode à virgule flottante*> *(1.1)*

noms prédéfinis: le nom *FLOAT* est prédéfini en tant que nom de **mode virgule flottante**.

sémantique: un mode virgule flottante définit un ensemble d'approximations numériques d'un intervalle de valeurs réelles, avec leur précision relative minimale, entre des bornes définies par l'implémentation, sur lesquelles sont définies les opérations arithmétiques et de classement (voir 5.3). Cet ensemble contient uniquement les valeurs pouvant être représentées par l'implémentation. Une implémentation peut définir d'autres modes virgule flottante ayant des bornes et une **précision** différentes (p. ex. *LONG_FLOAT*, *SHORT_FLOAT*) pouvant aussi être utilisés comme modes **parent** pour les intervalles (voir 13.3). Le mode *&FLOAT* est introduit comme le mode virtuel contenant toutes les valeurs des modes virgule flottante **prédéfinis** fixés par l'implémentation. La représentation interne d'une valeur à virgule flottante est la valeur à virgule flottante elle-même. Il faut noter que *&FLOAT* n'est pas un mode **prédéfini** (même s'il peut avoir les mêmes bornes que celles d'un mode virgule flottante **prédéfini**).

propriétés statiques: un mode virgule flottante a les propriétés héréditaires suivantes:

- une **borne supérieure** et une **borne inférieure** qui sont les littéraux dénotant respectivement la valeur la plus élevée et la valeur la plus faible définies par le mode virgule flottante. Ils sont fixés par l'implémentation;
- une **précision** qui est le nombre maximal de décimales significatives définies par le mode;

- une **limite inférieure positive** et une **limite supérieure négative** qui sont les littéraux dénotant respectivement la valeur positive la plus petite et la valeur négative la plus grande qui peuvent être reproduits avec exactitude dans le mode à virgule flottante, à l'exception du zéro.

exemple:

FLOAT *(1.1)*

3.5.2 Modes intervalle à virgule flottante

syntaxe:

<mode intervalle à virgule flottante> ::= *(1)*
<nom de mode virgule flottante> (<intervalle de valeurs flottantes>) *(1.1)*
| **RANGE** *(<intervalle à de valeurs flottantes> [, <chiffres significatifs>])* *(1.2)*
| *<nom de mode intervalle à virgule flottante>* *(1.3)*

<intervalle de valeurs flottantes> ::= *(2)*
<borne flottante inférieure> : <borne flottante supérieure> *(2.1)*

<borne flottante inférieure> ::= *(3)*
<expression littérale virgule flottante> *(3.1)*

<borne flottante supérieure> ::= *(4)*
<expression littérale virgule flottante> *(4.1)*

<chiffres significatifs> ::= *(5)*
<expression de littéral entier> *(5.1)*

sémantique: un mode intervalle à virgule flottante définit l'ensemble de valeurs comprises entre deux bornes données (bornes comprises) par l'*intervalle de valeurs flottantes*, le nombre de chiffres significatifs étant spécifié par *chiffres significatifs*. L'intervalle est repris d'un mode **parent** spécifique qui détermine les opérations sur et le classement de l'intervalle de valeurs. Par exemple, **RANGE** (–10.0*E*1 : 10.0*E*1, 2) dénote les valeurs : –10.0, –9.9, …, –0.11, –0.1, 0, 0.1, …,10.0.

propriétés statiques: un mode intervalle à virgule flottante a les propriétés non héréditaires suivantes: il a un mode **parent** défini comme suit:

- si le mode intervalle à virgule flottante est de la forme:

  <nom de mode virgule flottante> (*<intervalle de valeurs flottantes>*)

  et que le nom de *mode virgule flottante* n'est pas un mode intervalle à virgule flottante, le mode **parent** est nom de *mode virgule flottante*; sinon il est le mode **parent** du nom de *mode virgule flottante*.

- Si le mode intervalle à virgule flottante est de la forme:

  RANGE (*<intervalle de valeurs flottantes>* [, *<chiffres significatifs>*])

  le mode **parent** dépend de la **classe résultante** des classes de la *borne flottante supérieure* et de la *borne flottante inférieure* dans *l'intervalle littéral*:

  – s'il s'agit d'une M-classe par dérivation, où M est un mode virgule flottante, le mode **parent** est le mode virgule flottante **prédéfini** choisi par l'implémentation de telle manière qu'il contienne l'intervalle de valeurs fournies par l'*intervalle de valeurs flottantes*, avec la **précision** définie ci-après;

  – sinon c'est le mode **racine** de la **classe résultante**.

- Si le mode intervalle à virgule flottante est un *nom de mode intervalle à virgule flottante* qui est un nom **synmode**, son mode **parent** est celui du mode **définissant** du nom **synmode**; sinon c'est un nom **néomode** et dans ce cas son mode **parent** est le mode **parent** virtuellement introduit (voir 3.2.3).

Un mode intervalle à virgule flottante a les propriétés héréditaires suivantes:

- Une **borne supérieure** et une **borne inférieure** qui sont les littéraux dénotant respectivement les valeurs fournies par la *borne flottante inférieure* et la *borne flottante supérieure dans l'intervalle de valeurs flottantes.*

- Une **précision** qui est, si le mode intervalle à virgule flottante est de la forme suivante:

  RANGE (*<intervalle de valeurs flottantes>* [, *<chiffres significatifs>*])

  – si la valeur fournie par *chiffres significatifs* est spécifiée;

  – ou bien la **précision** la plus grande des **précisions** de la *borne flottante inférieure* et de la *borne flottante supérieure*.

  Sinon elle est celle de *nom de mode virgule flottante* ou du *nom de mode intervalle à virgule flottante*.

conditions statiques: *borne flottante inférieure* doit fournir une valeur qui est inférieure ou égale à la valeur fournie par *borne flottante supérieure*, et les deux valeurs doivent appartenir à l'ensemble de valeurs définies par le *nom de mode virgule flottante*, s'il est spécifié.

Il faut qu'il y ait un mode virgule flottante **prédéfini** qui contient la **borne supérieure** et la **borne inférieure** et la **précision** spécifiée.

La valeur de *chiffre significatif* doit être supérieure à zéro.

L'évaluation de 1.*borne flottante inférieure*, 2.*borne flottante supérieure* ne doit pas dépendre directement ou indirectement de la valeur de 1.**borne inférieure**, 2.**borne supérieure** du mode intervalle à virgule flottante.

3.6 Modes ensembliste

syntaxe:

<mode ensembliste> ::= *(1)*
POWERSET *<mode primitif>* *(1.1)*
| *<nom de mode ensembliste>* *(1.2)*

<mode primitif> ::= *(2)*
<mode discret> *(2.1)*

sémantique: un mode ensembliste définit des valeurs qui sont des ensembles de valeurs de son mode primitif. Les valeurs ensembliste comprennent tous les sous-ensembles du mode primitif. Les opérateurs usuels d'opérations sur les ensembles sont définis sur les valeurs de mode ensembliste (voir 5.3).

Le **nombre de valeurs** maximal du mode primitif est fixé par l'implémentation.

propriétés statiques: un mode ensembliste a la propriété héréditaire suivante:

- Il a un mode **primitif** unique qui est le *mode primitif*.

exemples:

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

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

9.6 *number_list* *(1.2)*

3.7 Modes référence

3.7.1 Généralités

syntaxe:

<mode référence> ::= *(1)*
<mode référence liée> *(1.1)*
| *<mode référence libre>* *(1.2)*
| *<mode descripteur>* *(1.3)*

sémantique: un mode référence définit des références (adresses ou descripteurs) de locus **référençable**. Par définition, les références liées référencent des locus d'un mode statique donné ou d'un ensemble donné de modes moreta connexes; les références libres peuvent référencer des locus de n'importe quel mode statique; les descripteurs référencent des locus de mode dynamique.

L'opération de déréférenciation est définie sur les valeurs référence (voir 4.2.3, 4.2.4 et 4.2.5), rendant le locus qui est référencé.

Deux valeurs référence sont égales si et seulement si toutes deux, soit référencent le même locus, soit ne référencent aucun locus (c'est-à-dire sont la valeur *NULL*).

3.7.2 Modes référence liée

syntaxe:

<mode référence liée> ::= *(1)*
REF *<mode référence>* *(1.1)*
| *<nom de mode référence liée>* *(1.2)*

<mode référence> ::= *(2)*
<mode> *(2.1)*

sémantique: les références liées définissent des valeurs référence de locus du mode référencé spécifié.

Si le mode référence est un mode M non moreta, le mode référence liée définit les valeurs références aux locus de M.

Si le mode référence liée est un mode MM moreta, le mode référence liée définit les valeurs références aux locus de MM ou de tout successeur de MM.

propriétés statiques: un mode référence liée a la propriété héréditaire suivante:

- Il a un mode **référencé** qui est le *mode référencé*.

exemple:

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

3.7.3 Modes référence libre

syntaxe:

<*mode* référence *libre*> *::=* *(1)*
<*nom de mode référence libre*> *(1.1)*

noms prédéfinis: le nom *PTR* est prédéfini comme nom de **mode référence libre**.

sémantique: un mode référence libre définit des valeurs référence de locus de tout mode statique.

exemple:

19.8 *PTR* *(1.1)*

3.7.4 Modes descripteur

syntaxe:

<*mode descripteur*> *::=* *(1)*
ROW <*mode chaîne*> *(1.1)*
| **ROW** <*mode matrice*> *(1.2)*
| **ROW** <*mode structure variable*> *(1.3)*
| <*nom de mode descripteur*> *(1.4)*

sémantique: un mode descripteur définit des valeurs référence de locus de mode dynamique (qui sont des locus d'un mode paramétré aux paramètres non **constants**).

Une valeur descripteur peut référencer:

- des locus **chaîne de longueur** non **constante**;
- des locus matrices à **borne supérieure** non **constante**;
- des locus structure paramétrée dont les paramètres sont non **constants**.

propriétés statiques: un mode descripteur a la propriété héréditaire suivante:

- Il a un mode **référencé originel**, qui est respectivement le *mode chaîne*, le *mode matrice*, ou le *mode structure variable*.

condition statique: le *mode structure variable* doit être **paramétrable**.

exemple:

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

3.8 Modes procédure

syntaxe:

<*mode procédure*> *::=* *(1)*
PROC *(* [<*liste de paramètres*>] *)* [<*spec de résultat*>]
[**EXCEPTIONS** *(*<*liste d'exceptions*>*)*] *(1.1)*
| <*nom de mode procédure*> *(1.2)*

<liste de paramètres> ::= *(2)*
<spec de paramètre> { , *<spec de paramètre>* }* *(2.1)*

<spec de paramètre> ::= *(3)*
<mode> [*<attribut de paramètre>*] *(3.1)*

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

<spec de résultat> ::= *(5)*
RETURNS *(<mode>* [*<attribut de résultat>*] *)* *(5.1)*

<attribut de résultat>::= *(6)*
[**NONREF**] **LOC** [**DYNAMIC**] *(6.1)*

<liste d'exceptions> ::= *(7)*
<nom d'exception> { , *<nom d'exception>* }* *(7.1)*

sémantique: un mode procédure définit des valeurs procédure (**générales**), c'est-à-dire les objets dénotés par des noms de **procédures générales**, qui sont eux-mêmes des noms définis dans les énoncés de définition de procédure. Les valeurs procédure indiquent des fragments de code dans un contexte dynamique. Les modes procédure permettent de manipuler dynamiquement une procédure, c'est-à-dire de la passer comme paramètre à d'autres procédures, de l'envoyer comme valeur message à un tampon, de la placer dans un locus, etc.

Les valeurs procédure peuvent être appelées (voir 6.7).

Deux valeurs procédure sont égales si et seulement si toutes deux soit dénotent la même procédure dans le même contexte dynamique, soit ne dénotent aucune procédure (c'est-à-dire sont la valeur *NULL*).

propriétés statiques: un mode procédure a les propriétés héréditaires suivantes:

- il a une liste de **specs de paramètre**, chacune étant constituée d'un mode et, éventuellement, d'un attribut de paramètre. Les **specs de paramètre** sont définies par la *liste de paramètres*;
- il a une **spec de résultat** facultative, constituée d'un mode et d'un attribut de résultat facultatif. La **spec de résultat** est définie par la *spec de résultat*;
- il a un ensemble éventuellement vide de noms d'**exception**, qui sont les noms mentionnés dans la *liste d'exceptions*.

conditions statiques: tous les noms mentionnés dans la *liste d'exceptions* doivent être différents.

Le *mode* apparaissant dans la *spec de paramètre* ou dans la *spec de résultat* ne peut avoir la **propriété de non-valeur** que si **LOC** y est spécifié.

Si **DYNAMIC** est spécifié dans la *spec de paramètre* ou la *spec de résultat*, le *mode* doit y être **paramétrable**.

3.9 Modes instance

syntaxe:

<mode instance> ::= *(1)*
<nom de mode instance> *(1.1)*

noms prédéfinis: le nom *INSTANCE* est prédéfini comme nom de **mode instance**.

sémantique: un mode instance définit des valeurs qui identifient des processus. La création d'un nouveau processus (voir 5.2.15, 6.13 et 11.1) produit une valeur instance unique comme identification pour le processus créé.

Deux valeurs instance sont égales si, et seulement si, toutes deux, soit identifient le même processus, soit n'identifient aucun processus (c'est-à-dire sont la valeur *NULL*).

exemple:

15.39 INSTANCE *(1.1)*

3.10 Modes synchronisation

3.10.1 Généralités

syntaxe:

<mode synchronisation> ::= *(1)*
<mode événement> *(1.1)*
| *<mode tampon>* *(1.2)*

sémantique: le mode synchronisation donne un moyen de synchronisation des processus et de communication entre eux (voir l'article 11). Il n'existe pas d'expressions en CHILL dénotant une valeur définie par un mode synchronisation. En conséquence, il n'y a pas d'opérations définies sur ces valeurs.

3.10.2 Modes événement

syntaxe:

<mode événement> ::= *(1)*
EVENT [(*<longueur d'événement>*)] *(1.1)*
| *<nom de mode événement>* *(1.2)*

<longueur d'événement> ::= *(2)*
<expression de littéral entier> *(2.1)*

sémantique: un locus de mode événement donne des moyens de synchronisation entre processus. Les opérations définies sur les locus de mode événement sont l'action continuer, l'action mettre en attente et l'action mettre en attente et choisir, décrites respectivement aux 6.15, 6.16 et 6.17.

La *longueur d'événement* spécifie le nombre maximal de processus qui peuvent être différés dans un locus événement; ce nombre n'a pas de limite si aucune *longueur d'événement* n'est spécifiée.

Un locus de mode événement qui contient la valeur **non définie** est un événement "vide", c'est-à-dire qu'aucun processus en attente n'y est rattaché.

propriétés statiques: un mode événement a la propriété héréditaire suivante:

- une **longueur d'événement** facultative qui est la valeur rendue par *longueur d'événement*.

conditions statiques: la *longueur d'événement* doit rendre une valeur positive.

L'évaluation de la *longueur d'événement* ne doit pas dépendre directement ou indirectement de la valeur de **longueur d'événement** du mode événement.

exemple:

14.10 **EVENT** *(1.1)*

3.10.3 Modes tampon

syntaxe:

<mode tampon> ::= *(1)*
BUFFER [(*<longueur de tampon>*)] *<mode d'élément tampon>* *(1.1)*
| *<nom de mode tampon>* *(1.2)*

<longueur de tampon> ::= *(2)*
<expression de littéral entier> *(2.1)*

<mode d'élément tampon> ::= *(3)*
<mode> *(3.1)*

sémantique: un locus de mode tampon donne des moyens de synchronisation des processus et de communication entre eux. Les opérations définies sur les locus tampon sont l'action envoyer et l'action recevoir avec cas, décrites respectivement aux 6.18 et 6.19.

La *longueur de tampon* spécifie le nombre maximal de valeurs qui peut être stocké dans un locus événement; ce nombre n'a pas de limite si aucune *longueur de tampon* n'est spécifiée.

Un locus de mode tampon qui contient la valeur **non définie** est un tampon "vide", c'est-à-dire qu'aucun processus en attente n'y est rattaché et que le tampon ne contient aucun message.

propriétés statiques: un mode tampon a les propriétés héréditaires suivantes:

- une **longueur de tampon** facultative, qui est la valeur rendue par *longueur de tampon*;
- un mode d'**élément tampon** qui est le *mode d'élément tampon*.

conditions statiques: la *longueur de tampon* doit rendre une valeur non négative.

Le *mode d'élément tampon* ne doit pas avoir la **propriété de non-valeur**.

L'évaluation de la *longueur de tampon* ne doit pas dépendre directement ou indirectement de la valeur de **longueur de tampon** du mode tampon.

exemples:

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

16.34 *user_buffers* *(1.2)*

3.11 Modes entrée-sortie

3.11.1 Généralités

syntaxe:

<mode entrée-sortie> ::= *(1)*
<mode association> *(1.1)*
| *<mode accès>* *(1.2)*
| *<mode texte>* *(1.3)*

sémantique: un mode entrée-sortie permet de réaliser des opérations d'entrée-sortie définies dans l'article 7. Il n'existe pas dans le CHILL d'expression désignant une valeur définie par un mode entrée-sortie. Il n'y a donc pas d'opérations définies sur les valeurs.

exemple:

20.17 *ASSOCIATION* *(1.1)*

3.11.2 Modes association

syntaxe:

<mode association> ::= *(1)*
<nom de mode association> *(1.1)*

noms prédéfinis: le nom *ASSOCIATION* est prédéfini comme un nom de **mode association**.

sémantique: un locus de mode association peut contenir une valeur qui représente une relation avec un objet du monde extérieur. Dans le CHILL cette relation est appelée une association; des associations peuvent être créées par l'opération prédéfinie *ASSOCIATE* et terminées par *DISSOCIATE*.

Un locus de mode association qui contient la valeur **non définie** est "vide", c'est-à-dire qu'il ne contient aucune association.

3.11.3 Modes accès

syntaxe:

<mode accès> ::= *(1)*
ACCESS [(*<mode d'indice>*)] [*<mode enregistrement>* [**DYNAMIC**]] *(1.1)*
| *<nom de mode accès>* *(1.2)*

<mode enregistrement> ::= *(2)*
<mode> *(2.1)*

<mode d'indice> ::= *(3)*
<mode discret> *(3.1)*
| *<intervalle littéral>* *(3.2)*

syntaxe dérivée: la notation de mode d'indice *intervalle littéral* est tirée du mode discret **RANGE** (*intervalle littéral*).

sémantique: un locus de mode accès donne le moyen de trouver la position d'un fichier et de transférer des valeurs du programme CHILL à un fichier du monde extérieur, et vice versa.

Un mode accès peut définir un *mode enregistrement*; ce mode enregistrement définit le mode **racine** de la classe des valeurs qui peuvent être transférées par un locus de ce mode accès à un fichier ou à partir de celui-ci. Le mode de la valeur transférée peut être dynamique, c'est-à-dire que la **taille** de l'enregistrement peut varier lorsque l'attribut **DYNAMIC** est spécifié dans la notation du mode accès ou lorsque le *mode enregistrement* est un mode chaîne **variable**. Dans ce dernier cas, il n'est pas nécessaire de spécifier **DYNAMIC**.

Un mode accès peut aussi définir un *mode indice*; ce mode indice définit la taille d'une "fenêtre" ouverte sur le (une partie du) fichier, à partir de laquelle il est possible de lire (ou d'écrire) des enregistrements au hasard. Cette fenêtre peut être placée dans un fichier (**indexable**) par l'opération connexion. Si aucun *mode indice* n'est spécifié, les enregistrements ne peuvent être transférés qu'en séquence.

Un locus de mode accès qui contient la valeur **indéfini** est "vide", c'est-à-dire qu'il n'est pas connecté à une association.

propriétés statiques: un mode accès a les propriétés héréditaires suivantes:

- il a un mode **enregistrement** facultatif qui est le *mode enregistrement*, s'il existe. C'est un mode **enregistrement dynamique** si **DYNAMIC** est spécifié ou si le *mode enregistrement* est un mode chaîne **variable**; autrement, c'est un mode **enregistrement statique**;
- il a un mode **indice** facultatif, qui est le *mode indice*;
- une **borne supérieure** et une **borne inférieure** facultatives qui sont la **borne supérieure** et la **borne inférieure** du *mode indice*, s'il est présent.

conditions statiques: le *mode enregistrement* facultatif ne doit pas avoir la **propriété de non-valeur**.

Si **DYNAMIC** est spécifié, le mode **enregistrement** doit être **paramétrable** et ne doit pas être un mode structure **sans étiquettes**.

Le *mode indice* ne doit pas être un mode ensemble **avec numéros**, ni un mode intervalle **avec numéros**.

Si le *mode indice* est un *intervalle littéral* ayant la forme:

<*borne inférieure*> : <*borne supérieure*>

l'évaluation de 1.*borne inférieure*, 2.*borne supérieure*, ne doit pas dépendre directement ou indirectement de la valeur de 1.**borne inférieure**, 2.**borne supérieure** du mode d'accès.

exemples:

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 Modes texte

syntaxe:

<*mode texte*> *::=* *(1)*
<*mode texte étroit*> *(1.1)*
| <*mode texte large*> *(1.2)*

<*mode texte étroit*> *(2)*
TEXT (<*longueur de texte*>) [<*mode d'indice*>] [**DYNAMIC**] *(2.1)*

<*mode texte large*> *(3)*
WTEXT (<*longueur de texte*>) [<*mode d'indice*>] [**DYNAMIC**] *(3.1)*

<*longueur de texte*> *::=* *(4)*
<*expression de littéral entier*> *(4.1)*

sémantique: un locus de mode texte permet de transférer les valeurs, représentées sous forme accessible en lecture par l'homme, du programme CHILL à un fichier du monde extérieur, et vice versa. Un locus de mode texte a des sous-locus **enregistrement de texte** et **accès**. L'**enregistrement de texte** est initialisé avec une chaîne vide.

Un mode texte a une **longueur de texte** qui définit la longueur maximale des enregistrements qui peut être transférée, et éventuellement un mode **indice** qui a la même signification que pour les modes accès. L'attribut **longueur effective** d'un locus de mode texte est la **longueur effective** de son **enregistrement de texte.**

Un locus de mode texte qui contient une valeur **non définie** a un sous-locus **enregistrement de texte** contenant la chaîne vide et un sous-locus d'**accès** contenant la valeur **non définie**.

propriétés statiques: un mode texte a les propriétés héréditaires suivantes:

- il a une **longueur de texte** qui est la valeur fournie par la *longueur de texte*;
- il a un mode **enregistrement de texte** qui est **CHARS** (<*longueur de texte*>) **VARYING** dans le cas de **TEXT** et **WCHARS** (<*longueur de texte*>) **VARYING** dans le cas de **WTEXT**;
- il a un mode **accès** qui est **ACCESS** [(<*mode d'indice*>)] **CHARS** (<*longueur de texte*>) [**DYNAMIC**] dans le cas de **TEXT** et **WCHARS** (<*longueur de texte*>) [**DYNAMIC**] dans le cas de **WTEXT** (<*mode indice*>, **DYNAMIC** font partie du mode seulement s'ils sont spécifiés).
- Une **borne supérieure** et une **borne inférieure** facultatives qui sont la **borne supérieure** et la **borne inférieure** du *mode indice*, s'il est présent.

conditions statiques: si le *mode indice* est un *intervalle littéral* ayant la forme:

<*borne inférieure*> : <*borne supérieure*>

l'évaluation de 1.*borne inférieure*, 2.*borne supérieure* ne doit pas dépendre directement ou indirectement de la valeur de 1.**borne inférieure**, 2.**borne supérieure** du mode texte.

exemple:

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

3.12 Modes temporisation

3.12.1 Généralités

syntaxe:

<*mode temporisation*> *::=* *(1)*
<*mode durée*> *(1.1)*
| <*mode temps absolu*> *(1.2)*

sémantique: un mode temporisation est utilisé pour la surveillance des processus décrits à l'article 9. Les valeurs de temporisation sont créées par un ensemble d'opérations prédéfinies. Les opérateurs relationnels sont définis sur les valeurs de temporisation.

3.12.2 Modes durée

syntaxe:

<*mode durée*> *::=* *(1)*
<*nom de mode durée*> *(1.1)*

noms prédéfinis: le nom *DURATION* est prédéfini comme un nom de **mode durée**.

sémantique: un mode durée définit des valeurs qui représentent des périodes de temps. L'ensemble de valeurs définies par le mode durée est défini par l'implémentation. Une implémentation peut choisir de représenter les valeurs de durée comme des paires de précision et de valeur. Les valeurs de durée sont ordonnées de façon intuitive.

3.12.3 Modes temps absolu

syntaxe:

<*mode temps absolu*> ::= *(1)*
<*nom de mode temps absolu*> *(1.1)*

noms prédéfinis: le nom *TIME* est prédéfini comme un nom de **mode temps absolu**.

sémantique: un mode temps absolu définit des valeurs qui représentent des instants. L'ensemble de valeurs définies par le mode temps absolu est défini par l'implémentation. Les valeurs de temps absolu sont ordonnées de façon intuitive.

3.13 Modes composites

3.13.1 Généralités

syntaxe:

<mode composite> ::= *(1)*
<mode chaîne> *(1.1)*
| *<mode matrice>* *(1.2)*
| *<mode structure>* *(1.3)*
| *<mode moreta>* *(1.4)*

sémantique: un mode composite définit des valeurs composites, c'est-à-dire des valeurs constituées par des sous-composantes auxquelles on peut avoir accès ou qu'on peut obtenir (voir 4.2.6-4.2.10 et 5.2.6-5.2.10).

3.13.2 Modes chaîne

syntaxe:

<mode chaîne> ::= *(1)*
<type de chaîne> (*<longueur de chaîne>*) [**VARYING**] *(1.1)*
| *<mode chaîne paramétré>* *(1.2)*
| *<nom de mode chaîne>* *(1.3)*

<mode chaîne paramétré> ::= *(2)*
<nom de mode chaîne d'origine> (*<longueur de chaîne>*) *(2.1)*
| *<nom de mode chaîne paramétré>* *(2.2)*

<nom de mode chaîne d'origine> ::= *(3)*
<nom de mode chaîne> *(3.1)*

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

longueur de chaîne> ::= *(5)*
<expression de littéral entier> *(5.1)*

sémantique: un mode chaîne **fixe** définit des valeurs chaîne binaire ou chaîne de caractères de longueur indiquée ou impliquée par le mode chaîne. Un mode chaîne **variable** définit des valeurs chaîne binaire ou chaîne de caractères de longueur variant dynamiquement de 0 à la **longueur de la chaîne**. La longueur n'est connue qu'au moment de l'exécution à partir de la valeur de l'attribut **longueur effective**. Pour un mode de chaîne **fixe**, la **longueur effective** est toujours égale à la **longueur de la chaîne**. Les chaînes de caractères sont des séquences de valeurs de caractères; les chaînes binaires sont des séquences de valeurs booléennes.

Les valeurs chaîne sont vides ou bien ont des éléments qui sont numérotés à partir de 0 vers les valeurs croissantes.

Les valeurs chaîne d'un mode chaîne donné sont entièrement ordonnées selon l'ordre des valeurs composantes et de la définition suivante.

Deux chaînes *s* et *t* sont égales si et seulement si elles sont vides ou ont la même longueur *l* et $s(i) = t(i)$ pour tout $0 \le i < l$. Une chaîne *s* précède *t* quand:

- il existe un indice *j* tel que $s(j) < t(j)$ et $s(0 : j - 1) = t(0 : j - 1)$,
- *LENGTH* (*s*) < *LENGTH* (*t*) et *s* = *t*(0 **UP** *LENGTH* (*s*)).

L'opérateur de concaténation est défini sur les valeurs de chaîne. Les opérateurs logiques habituels sont définis sur des valeurs chaîne de bits et agissent entre leurs éléments correspondants (voir 5.3).

La longueur maximale des modes chaîne est fixée par l'implémentation.

propriétés statiques: un mode chaîne a les propriétés héréditaires suivantes:

- il a une **longueur de chaîne**, qui est la valeur rendue par la *longueur de chaîne.*
- il a une **borne supérieure** et une **borne inférieure** qui sont les valeurs rendues par la **longueur de chaîne** – 1 et 0, respectivement;

- un mode **élément** qui est *M* ou **READ** *M*, où *M* est *BOOL*, *CHAR* ou *WCHAR* selon que *type de chaîne* spécifie **BOOLS**, **CHARS** ou **WCHARS**, ou le mode **élément** du *nom de mode chaîne d'origine*, respectivement. Le mode **élément** sera **READ** *M* si et seulement si le *mode chaîne* est un mode **protégé**; dans ce cas, c'est un mode **protégé** implicite;
- c'est un mode chaîne **variable** si **VARYING** est spécifié ou si le *nom de mode chaîne d'origine* est un mode chaîne **variable**, sinon c'est un mode chaîne **fixe**.

Un mode chaîne est **paramétré** si et seulement s'il est un *mode chaîne paramétré*.

Un mode chaîne **paramétré** a un mode chaîne **d'origine** qui est le mode désigné par le *nom de mode chaîne d'origine*.

Un mode chaîne **variable** a la propriété non héréditaire suivante: il a un mode **composite**, défini ainsi:

- si le mode chaîne **variable** a la forme:

  <*type de chaîne*> (<*longueur de chaîne*>) **VARYING**

  c'est le <*type de chaîne*> (<*longueur de chaîne*>);

- si le mode chaîne **variable** a la forme:

  <*nom de mode chaîne d'origine*> (<*longueur de chaîne*>)

  le mode **composite** est &*nom* (*longueur de chaîne*), où &*nom* est un nom de **synmode** introduit virtuellement **synonyme** du mode **composante** du *nom de mode chaîne variable d' origine*;

- si le mode chaîne **variable** est un *nom de mode chaîne* qui est un nom de **synmode**, son mode **composite** est celui du mode **définissant** du nom de **synmode**; sinon, c'est un nom de **néomode** et son mode **composante** est le mode **composante** virtuellement introduit (voir 3.2.3).

conditions statiques: la *longueur de chaîne* doit rendre une valeur non négative.

La valeur rendue par la *longueur de chaîne* contenue directement dans un *mode chaîne paramétré* doit être inférieure ou égale à la **longueur de chaîne** du *nom de mode chaîne d'origine*. Cette condition ne s'applique qu'aux modes chaîne **paramétrés** qui ne sont pas introduits virtuellement.

L'évaluation de la *longueur de chaîne* ne doit pas dépendre directement ou indirectement de la valeur de la **longueur de chaîne** du mode chaîne.

exemples:

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

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

3.13.3 Modes rangée

syntaxe:

<*mode matrice*> *::=* *(1)*
  ARRAY (<*mode indice*> {, <*mode indice*> }*)
  <*mode d'élément*> { <*implantation d'élément*> }* *(1.1)*
| <*mode matrice paramétré*> *(1.2)*
| <*nom de mode matrice*> *(1.3)*

<*mode matrice paramétré*> *::=* *(2)*
  <*nom de mode matrice d' origine*> <*indice supérieur*>) *(2.1)*
| <*nom de mode matrice paramétré*> *(2.2)*

<*nom de mode matrice origine*> *::=* *(3)*
  <*nom de mode matrice*> *(3.1)*

<*indice supérieur*> *::=* *(4)*
  <*expression littérale discrète*> *(4.1)*

<*mode d'élément*> *::=* *(5)*
  <*mode*> *(5.1)*

syntaxe dérivée: un *mode matrice* avec plus d'un mode indice (dénotant une matrice multidimensionnelle), est une syntaxe dérivée pour un *mode matrice* avec un *mode élément* qui est lui-même un *mode matrice*. Par exemple:

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

est dérivé de:

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

C'est uniquement dans le cas où cette syntaxe dérivée est utilisée qu'il est permis plus d'une occurrence d'*implantation d'élément*. Le nombre d'occurrences d'*implantation d'élément* doit être inférieur ou égal au nombre d'occurrences de *mode indice*. Dans ce cas, l'*implantation d'élément* la plus à gauche est associée au *mode élément* le plus interne, etc.

sémantique: un mode matrice définit des valeurs composites, qui sont des listes de valeurs définies par son mode élément. L'implantation physique d'un locus ou d'une valeur matrice peut être contrôlée par une spécification d'*implantation d'élément* (voir 3.13.5). Deux valeurs matrice sont égales si et seulement si elles ont le même **nombre d'éléments** et que toutes les valeurs élément correspondantes sont égales.

Le **nombre d'éléments** maximal des modes matrice est fixé par l'implémentation.

propriétés statiques: un mode matrice a les propriétés héréditaires suivantes:

- il a un mode **indice** qui est le *mode indice* dans le cas où il ne s'agit pas d'un *mode matrice paramétré*, sinon le mode **indice** est le mode intervalle construit comme:

  &nom (*borne inférieure : borne supérieure*),

  où *&nom* est un nom de **synmode** virtuel **synonyme** du mode **indice** du *nom de mode matrice d'origine*, *borne inférieure* est la borne inférieure du mode **indice** du *nom de mode matrice d'origine*, et *borne supérieure* est l'*indice supérieur*;

- il a une **borne supérieure** et une **borne inférieure** qui sont, respectivement, la **borne supérieure** et la **borne inférieure** de son mode **indice**;

- il a un mode **élément** qui est soit *M* soit **READ** *M*, où *M* est le *mode élément* ou le mode **élément** du *nom de mode matrice d'origine*, selon le cas. Le mode **élément** est **READ** *M* si et seulement si *M* n'est pas un mode **protégé** et que le *mode matrice* est un mode **protégé**. Le mode **élément** est un mode **protégé** implicitement s'il est **READ** *M*;

- il a une **implantation d'élément** qui, s'il est un *mode matrice paramétré*, est l'**implantation d'élément** de son *nom de mode matrice d'origine* et, sinon, est soit l'*implantation d'élément* spécifiée, soit un choix par défaut de l'implémentation, qui est soit **PACK** soit **NOPACK**.

- Il a un **nombre d'éléments** qui est la valeur rendue par:

  NUM (*borne supérieure*) – *NUM* (*borne inférieure*) + 1,

  où *borne supérieure* et *borne inférieure* sont respectivement la **borne supérieure** et la **borne inférieure** de son mode **indice**.

- C'est un mode **mappé** si et seulement si une *implantation d'élément* est spécifiée et s'il s'agit d'un *pas*.

Un mode matrice est **paramétré** si et seulement s'il est un *mode matrice paramétré*.

Un mode matrice **paramétré** a un mode matrice **d'origine** qui est le mode désigné par le *nom de mode matrice d'origine*.

conditions statiques: la classe de l'*indice supérieur* doit être **compatible** avec le mode **indice** du *nom de mode matrice d'origine* et la valeur qu'il rend doit se trouver dans l'intervalle défini par ce mode **indice**.

Si le mode matrice est un *mode matrice paramétré*, l'évaluation de l'*indice supérieur* ne doit pas dépendre directement ou indirectement de la valeur de la **borne supérieure** du mode matrice; si le mode indice est ni un *mode matrice paramétré*, ni un *nom de mode matrice*, et si le *mode indice* est un *intervalle littéral* de la forme

<*borne inférieure*> : <*borne supérieure*>

l'évaluation de 1.*borne inférieure*, 2.*borne supérieure* ne doit pas dépendre directement ou indirectement de la valeur de 1.**borne inférieure**, 2.**borne supérieure** du mode matrice.

exemples:

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 Modes structure

syntaxe:

<mode structure> ::= *(1)*
STRUCT (*<champ>* { , *<champ>* }*) *(1.1)*
| *<mode structure paramétré>* *(1.2)*
| *<nom de mode structure>* *(1.3)*

<champ> ::= *(2)*
<champ fixe> *(2.1)*
| *<champ alternatif>* *(2.2)*

<champ fixe> ::= *(3)*
<liste d'occurences de définitions de noms de champ> <mode>
[*<implantation de champ>*] *(3.1)*

<champ alternatif> ::= *(4)*
CASE [*<liste d'étiquettes>*] **OF**
<alternative variant> { , *<alternative variant>* }*
[**ELSE** [*<champ récurrent>* { , *<champ récurrent>* }*]] **ESAC** *(4.1)*

<alternative variant> ::= *(5)*
[*<spécification d'étiquettes de cas>*] : *champ récurrent>* { , *<champ récurrent>* }*] *(5.1)*

<liste d'étiquettes> ::= *(6)*
<nom de champ étiquette> { , *<nom de champ étiquette>* }* *(6.1)*

<champ récurrent> ::= *(7)*
<liste d'occurences de définitions de noms de champ> <mode>
[*<implantation de champ>*] *(7.1)*

<mode structure paramétré> ::= *(8)*
<nom de mode structure variable originel> (*<liste d'expressions littérales>*) *(8.1)*
| *<nom de mode structure paramétré>* *(8.2)*

<nom de mode structure variable originel> ::= *(9)*
<nom de mode structure variable> *(9.1)*

<liste d'expressions littérales> ::= *(10)*
<expression littérale discrète> { , *<expression littérale discrète>* }* *(10.1)*

syntaxe dérivée: une occurrence de *champ fixe*, ou une occurrence de *champ récurrent*, où la *liste d'occurences de définitions de noms de champ* comporte plus d'une occurence de *définition de nom de champ*, est une syntaxe dérivée pour plusieurs occurrences de *champs fixes* ou de *champs récurrents*, selon le cas, chacune comportant une occurence de *définition de nom de champ*, le *mode* spécifié et l'*implantation de champ* facultative. Cette dernière ne doit pas être *pos* dans ce cas. Par exemple:

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

est dérivé de:

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

sémantique: les modes structure définissent des valeurs composites constituées d'une liste de valeurs sélectionnables par un nom de composante. Chacune de ces valeurs est définie par un mode attaché au nom de composante. Les valeurs structure peuvent résider dans des locus structure (composites) où le nom de composante sert d'accès au sous-locus. Les composantes d'une valeur ou d'un locus structure sont appelées champs et leurs noms, noms **de champ**.

Il existe des structures **fixes**, des structures **variables** et des structures **paramétrées**.

Les structures **fixes** sont constituées uniquement de champs fixes, c'est-à-dire de champs qui sont toujours présents et auxquels on peut accéder sans aucun contrôle dynamique.

Les structures **variables** ont des champs récurrents, c'est-à-dire des champs qui ne sont pas toujours présents. Pour les structures **variables avec étiquettes**, la présence de ces champs est connue seulement à l'exécution d'après la ou les valeurs de certains champs fixes associés, nommés champs **étiquettes**. Les structures **variables sans étiquettes** n'ont pas de champs **étiquettes**. Comme la composition d'une structure **variable** peut changer durant l'exécution, la **taille** d'un locus structure variable est basée sur le cas de taille maximale de l'ensemble des champs alternatifs (pire des cas).

Dans un *champ alternatif*, l'*alternative variant* choisi est celui pour lequel les valeurs donnent dans l'étiquette de cas une correspondance de spécification; dans le cas où il n'y a pas de correspondance de spécification, l'*alternative variant* qui suit **ELSE** (qui sera présent) est choisi.

Une structure **paramétrée** est déterminée par un mode structure **variable** pour lequel le choix de champs alternatifs est spécifié statiquement au moyen d'expressions **littérales**. La composition est fixée au point de création de la structure paramétrée et ne peut changer durant l'exécution. Les champs **étiquettes**, s'ils sont présents, sont **protégés** et initialisés automatiquement avec les valeurs spécifiées. Pour un locus structure paramétré, une quantité précise de mémoire peut être allouée au point de déclaration ou de génération. A noter qu'il existe également des modes structure **paramétrés** dynamiques. Leur sémantique est définie au 3.14.4.

L'implantation d'un locus ou d'une valeur structure peut être contrôlée au moyen d'une spécification d'implantation de champs (voir 3.13.5).

Deux valeurs structure sont égales si et seulement si les valeurs composantes correspondantes sont égales. Cependant, si les valeurs structure sont **variables sans étiquettes**, le résultat de la comparaison est défini par l'implémentation.

Pour un mode ayant la **propriété d'étiquetage et de paramétrage**, la valeur **non définie** dénote une valeur dans laquelle des sous-valeurs de champ **étiquette** sont égales aux valeurs des paramètres correspondants et toutes les autres sont égales à la valeur **non définie**.

propriétés statiques:

généralités: un mode structure a les propriétés héréditaires suivantes:

- c'est un mode structure **fixe** si et seulement s'il est un *mode structure* qui ne contient pas directement d'occurrence de *champs alternatifs*;
- c'est un mode structure **variable** si et seulement s'il est un *mode structure* contenant au moins une occurrence de *champs alternatifs*;
- c'est un mode structure **paramétré** si et seulement s'il est un *mode structure paramétré*;
- il a un ensemble de noms **de champ**. Cet ensemble est déterminé ci-dessous pour les différents cas. Un nom est dit nom **de champ** si et seulement s'il est défini dans une *liste d'occurence de définitions de noms de champ* dans les *champs fixes* ou les *champs récurrents* dans un *mode structure*;

  chaque *champ fixe*, *champ récurrent* et donc chaque nom de **champ** donné d'un mode structure donné a un mode de **champ** qui lui est attaché, et qui est soit *M* soit **READ** *M*, où *M* est le *mode* dans le *champ fixe* ou *champ récurrent*. Le mode **de champ** sera **READ** *M* si *M* n'est pas un mode **protégé** et soit que le mode structure est un mode **protégé**, soit que le champ est un **champ étiquette** d'un mode structure **paramétré**. Le mode de **champ** est un mode **protégé** implicitement s'il est **READ** *M*.

  Un *champ fixe*, *champ récurrent* et donc un nom **de champ** d'un mode structure donné a une **implantation de champ** qui lui est attachée et qui est l'*implantation de champ* dans le *champ fixe* ou *champ récurrent* si elle est présente, sinon l'implantation de champ par défaut, qui est **PACK** ou **NOPACK**.
- C'est un mode **mappé** si ses noms **de champ** ont une *implantation de champ* qui est *pos*.

structures fixes: un mode structure **fixe** a la propriété héréditaire suivante:

- il a un ensemble de noms **de champ** qui est l'ensemble des noms définis par toute *occurence de définition de nom de champs* dans les *champs fixes*. Ces noms **de champ** sont des noms **de champ fixe**.

structures variables: un mode structure **variable** a les propriétés héréditaires suivantes:

- il a un ensemble de noms **de champ** qui est l'union de l'ensemble des noms définis par toute liste d'occurence de définitions de noms de champ dans les champs fixes et de l'ensemble des noms définis par toute liste d'occurence de définitions de noms de champ dans les choix de champs alternatifs. Les noms **de champ** définis par une liste d'occurence de définitions de noms dans les champs fixes sont les noms **de champ fixe** du mode structure **variable**, ses autres noms **de champ** sont les noms **de champ récurrent**;
- un nom **de champ** d'un mode structure **variable** est un nom **de champ étiquettes** si et seulement s'il apparaît dans une des listes d'étiquettes d'un champs alternatif. Les champs alternatifs dans lesquels aucune liste d'étiquettes n'est spécifiée, sont des *champs alternatifs* **sans étiquettes**;
- un mode structure **variable** est un mode structure **variable sans étiquettes** si toutes ses occurrences de *champs alternatifs* sont **sans étiquettes**. Sinon, c'est un mode structure **variable avec étiquettes**;
- un mode structure **variable** est un mode structure **variable paramétrable** s'il est soit un mode structure **variable avec étiquettes**, soit un mode structure **variable sans étiquettes** dans lequel, pour chaque occurrence de *champs alternatifs*, une *spécification d'étiquettes de cas* est donnée pour toutes les occurrences d'alternative variant qu'elle contient;

- a un mode structure **variable paramétrable** est attachée une liste de classes déterminées comme suit:
  - si c'est un mode structure **variable avec étiquettes**, la liste des M_i – classes par valeur, où les M_i représentent les modes des noms **de champ étiquette** dans l'ordre où ils sont définis dans les *champs fixes*;
  - si c'est un mode structure **variable sans étiquettes**, la liste est construite à partir **des listes** individuelles **résultantes des classes** de chaque *champs alternatifs* en les concaténant dans l'ordre où les *champs alternatifs* apparaissent. La **liste résultante des classes** d'une occurrence de *champs alternatifs* est la **liste résultante des classes** de la liste d'occurrences de *spécification d'étiquettes de cas* qu'elle contient (voir 12.3).

structures paramétrées: un mode structure **paramétré** a les propriétés héréditaires suivantes:

- il a un mode structure **variable originel**, qui est le mode dénoté par le *nom de mode structure variable originel*;
- il a un ensemble de noms **de champ** qui est l'union de l'ensemble des noms **de champ fixe** de son mode structure **variable originel** et de l'ensemble des noms **de champ récurrent** de son mode structure **variable originel** qui sont définis dans les occurrences d'*alternatives variants* sélectionnées par la liste de valeurs définies par la *liste d'expressions littérales*;
- l'ensemble des noms de **champ étiquette** d'un *mode structure paramétré* est l'ensemble des noms de **champ étiquette** de son mode structure **variable originel**;
- il a une liste de valeurs définies par la *liste d'expressions littérales*;
- c'est un mode structure **paramétré avec étiquettes** si son mode structure **variable originel** est un mode structure **variable avec étiquettes**, sinon le mode structure **paramétré** est **sans étiquettes**.

Pour les modes structure **paramétrés** dynamiques, voir 3.14.4.

conditions statiques:

généralités: tous les noms **de champ** d'un mode structure doivent être différents.

Si un champ a une implantation de champ qui est *pos*, tous les champs doivent avoir une implantation de champ qui doit être *pos*.

structures variables: un nom **de champ étiquette** doit être un nom **de champ fixe** et doit être textuellement défini avant toutes les occurrences de *champs alternatifs* dans la *liste d'étiquettes* desquels il est mentionné. (En conséquence, un champ **étiquette** précède tous les champs **récurrents** qui dépendent de lui.) Le mode d'un nom **de champ étiquette** doit être un mode discret.

Le *mode* de *champ récurrent* peut n'avoir ni la **propriété de non-valeur** ni celle **d'étiquetage et de paramétrage**.

Dans un mode structure **variable**, les occurrences de *champs alternatifs* doivent être ou bien toutes **avec étiquettes** ou bien toutes **sans étiquettes**. Pour des *champs alternatifs* **sans étiquettes**, la *spécification d'étiquettes de cas* doit être spécifiée dans chaque *champ à choisir*. Pour des *champs alternatifs* **avec étiquettes**, la *spécification d'étiquettes de cas* peut être omise dans toutes les occurrences d'alternatives variants ou doit être spécifiée pour toutes les occurrences d'alternatives variants.

Si, pour un mode structure **variable sans étiquettes**, un des *champs alternatifs* a une *spécification d'étiquettes de cas*, alors tous les *champs alternatifs* doivent avoir une *spécification d'étiquettes de cas*.

Pour les *champs alternatifs*, il faut que soient satisfaites les conditions de sélection de cas (voir 12.3) ainsi que les mêmes exigences de complétude, cohérence et compatibilité que pour l'action de cas (voir 6.4). Chacun des noms **de champ étiquette** de la *liste d'étiquettes*, s'ils sont présents, sert de sélecteur de cas avec la M-classe par valeur, où M est le mode du nom **de champ étiquette**. Dans le cas de *champs alternatifs* **sans étiquette**, les contrôles impliquant les sélecteurs de cas sont ignorés.

Pour un mode structure **variable paramétrable**, aucune des classes de la liste de classes qui lui est attachée ne peut être la classe **toute**. (Cette condition est satisfaite automatiquement par un mode structure **variable avec étiquettes**.)

structures paramétrées: le *nom de mode structure variable originel* doit être **paramétrable**.

Il doit y avoir autant d'expressions **littérales** dans la *liste d'expressions littérales* qu'il y a de classes dans la liste de classes du *nom de mode structure variable originel*. La classe de chaque expression **littérale** doit être **compatible** avec la classe correspondante (par sa position) de la liste de classes. Si cette dernière classe est une M-classe par valeur, la valeur rendue par l'expression **littérale** doit être une des valeurs définies par M.

exemples:

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 Description d'implantation pour modes matrice et modes structure

syntaxe:

<implantation d'élément> ::= *(1)*
PACK | **NOPACK** | *<pas>* *(1.1)*

<implantation de champ> ::= *(2)*
PACK | **NOPACK** | *<pos>* *(2.1)*

<pas> ::= *(3)*
STEP (*<pos>* [, *<taille de pas>*]) *(3.1)*

<pos> ::= *(4)*
POS (*<mot>*,*<bit initial>*,*<longueur>*) *(4.1)*
| **POS** (*<mot>* [, *<bit initial>* [: *<bit final>*]]) *(4.2)*

<mot> ::= *(5)*
<expression de littéral entier> *(5.1)*

<taille de pas> ::= *(6)*
<expression de littéral entier> *(6.1)*

<bit initial> ::= *(7)*
<expression de littéral entier> *(7.1)*

<bit final> ::= *(8)*
<expression de littéral entier> *(8.1)*

<longueur> ::= *(9)*
<expression de littéral entier> *(9.1)*

sémantique: il est possible de commander l'implantation d'une matrice ou d'une structure en donnant des informations de compactage ou de mappage dans son mode. L'information de compactage est soit **PACK**, soit **NOPACK**, l'information de mappage est soit un *pas* dans le cas de modes matrice, soit un *pos* dans le cas des modes structure. L'absence *d'implantation d'élément* ou *d'implantation de champ* dans un mode matrice ou structure sera toujours interprétée comme de l'information de compactage, c'est-à-dire comme **PACK** ou comme **NOPACK**.

Si on spécifie **PACK** pour les éléments d'une matrice ou pour les champs d'une structure, cela signifie que l'emploi de l'espace mémoire est optimisé pour les éléments de la matrice ou les champs de la structure, tandis que **NOPACK** implique que le temps d'accès aux éléments de matrice ou aux champs de structure est optimisé. **NOPACK** implique aussi la **référenciation**.

L'information **PACK**, **NOPACK** ne s'applique qu'à un niveau, c'est-à-dire elle ne s'applique qu'aux éléments de la matrice ou aux champs de la structure, mais pas aux composantes possibles des éléments de la matrice ou des champs de la structure. L'information d'implantation s'attache toujours au mode le plus proche possible et qui n'a pas déjà d'information d'implantation. Par exemple, si le compactage par défaut est **NOPACK**:

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

est équivalent à:

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

Il est également possible de commander l'implantation précise d'une matrice ou d'une structure en spécifiant une information de position pour ses composantes dans le mode. Cette information de position est donnée de la façon suivante:

- pour les modes matrice, l'information de position est donnée pour tous les éléments en même temps, sous la forme d'un *pas* suivant le mode matrice;
- pour les modes structure, l'information de position est donnée pour chaque champ individuellement, sous la forme d'un *pos* suivant le mode du champ.

L'information de mappage avec *pos* est donnée en décalages de mots et de bits. Un *pos* ayant la forme:

POS (<*mot*> , <*bit initial*> , <*longueur*>)

définit un décalage de bits de

NUM (*mot*) ∗ *WIDTH* + *NUM* (*bit initial*)

et une longueur de *NUM* (*longueur*) bits, où *WIDTH* est le nombre (défini par l'implémentation) de bits d'un mot et *mot* est une *expression de littéral entier*.

Lorsque *pos* est spécifié en *implantation de champ*, elle précise que le champ correspondant commence au décalage de bits donnés à partir du départ de chaque locus de ce mode et occupe la longueur donnée.

Un *pas* ayant la forme

STEP (<*pos*> , <*taille de pas*>)

définit une série de décalages de bits b_i lorsque *i* prend les valeurs *0* à *n–1*, où *n* est le **nombre d'éléments** de la matrice, et

$b_i = i * NUM$ (*taille du pas*)

Le *j*ème élément de la matrice commence à un décalage de bits de $p + b_j$ à partir du début de chaque locus du mode matrice, où *p* est le décalage de bits spécifié dans *pos*. Chaque élément occupe la longueur donnée dans *pos*.

Défauts

La notation:

POS (<*mot*> , <*bit initial*> : <*bit final*>)

est sémantiquement équivalente à:

POS (<*mot*> , <*bit initial*> , *NUM (bit final) – NUM* (<*bit initial*>) + 1)

La notation:

POS (<*mot*> , <*bit initial*>)

est sémantiquement équivalente à:

POS (<*mot*> , <*bit initial*> , *BTAILLE*)

où *BTAILLE* est le nombre minimal de bits nécessaire à représenter la composante pour laquelle le *pos* est spécifié.

La notation:

POS (<*mot*>)

est sémantiquement équivalente à:

POS (<*mot*> , *0* , *BTAILLE*)

La notation:

STEP (<*pos*>)

est sémantiquement équivalente à:

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

où *STAILLE* est la <*longueur*> spécifiée dans le *pos*, ou déductible du *pos* par les règles ci-dessus.

propriétés statiques: pour tout locus d'un mode matrice implanté, l'implantation d'élément du mode détermine la référenciation de ses sous-locus (y compris les sous-matrices et bande matricielle) comme suit:

- soit tous les sous-locus sont **référençables**, soit aucun d'entre eux ne l'est;
- si l'implantation d'élément est **NOPACK** tous les sous-locus sont **référençables**.

Pour tout locus d'un mode structure implanté, la référenciation d'un champ de structure sélectionné par un nom **de champ** est déterminée par l'implantation de champ du nom **de champ** comme suit:

- le nom **de champ** est **référençable** si l'implantation de champ est **NOPACK**.

conditions statiques: si le mode **des éléments** d'un mode matrice donné, ou le mode **de champ** d'un nom de **champ** d'un mode structure donné, est lui-même un mode matrice ou structure, ce doit être un mode **mappé** si le mode matrice ou structure donné est un mode **mappé**.

NUM (*mot*), *NUM* (*bit initial*), *NUM* (*bit final*), *NUM* (*longueur*) et *NUM* (*taille de pas*) ≥ 0;
NUM (*bit initial*) et *NUM* (*bit final*) ≤ *WIDTH*; *NUM* (*bit initial*) ≤ *NUM* (*bit final*).

Toute implémentation définit pour chaque mode le nombre minimal de bits nécessaire pour représenter ses valeurs, c'est-à-dire l'occupation minimale de bits. Pour des modes discrets, c'est un nombre quelconque de bits qui n'est pas inférieur au log de base deux du **nombre de valeurs** du mode. Pour les modes matrice, c'est le décalage de l'élément de l'indice le plus élevé, plus ses bits occupés. Pour des modes structure, c'est le décalage du bit occupé le plus élevé.

Pour chaque *pos* la *longueur* spécifiée ne doit pas être inférieure à l'occupation minimale de bit du mode des composantes de champ ou de matrice associées.

Pour chaque mode matrice **mappé**, la *taille de pas* ne doit pas être inférieure à la *longueur* donnée ou implicite dans le *pos*.

Cohérence et faisabilité

Cohérence: aucune composante d'une structure ne peut se voir imposer d'occuper un bit déjà occupé par une autre composante du même objet, sauf dans le cas de deux noms **de champ récurrent** définis dans le même *champ alternatif*; cependant, dans ce dernier cas, les noms **de champ récurrent** ne peuvent être tous deux définis dans la même *alternative variant*, ni tous deux suivre **ELSE**.

Faisabilité: le langage n'impose pas de conditions de faisabilité, sauf celle qui peut se déduire de la règle disant que la référenciation d'un sous-locus de tout locus (**référençable** ou non **référençable**) est déterminée seulement par l'implantation (de champ ou d'élément), ce qui est une propriété du mode du locus. Ceci restreint le mappage de composantes qui ont elles-mêmes des composantes **référençables**.

exemples:

17.5 **PACK** *(1.1)*

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

3.14 Modes dynamiques

3.14.1 Généralités

Un mode dynamique est un mode dont certaines propriétés ne sont connues qu'à l'exécution. Les modes dynamiques sont toujours des modes paramétrés avec un ou plusieurs paramètres connus à l'exécution. Cependant, pour les besoins de la description, des notations virtuelles sont introduites dans la présente Recommendation | Norme internationale. Ces notations virtuelles sont précédées (&) afin de les distinguer des notations qui peuvent effectivement apparaître dans un texte de programme en CHILL.

3.14.2 Modes chaîne dynamiques

dénotation virtuelle: &<*nom de mode chaîne d'origine*> (<*expression entière*>)

sémantique: un mode chaîne dynamique est un mode chaîne paramétré de longueur non **constante**.

propriétés statiques: les modes chaîne dynamiques ont les mêmes propriétés que les modes chaîne paramétrés, sauf en ce qui concerne les propriétés ci-après.

propriétés dynamiques:

- un mode chaîne dynamique a une **longueur** dynamique de **chaîne**, qui est la valeur rendue par l'*expression entière*;
- un mode chaîne dynamique a une **borne supérieure** et une **borne inférieure**, qui sont les valeurs fournies par la **longueur de chaîne**, respectivement –1 et 0.

3.14.3 Modes matrices dynamiques

dénotation virtuelle: &<*nom de mode matrice d'origine*> (<*expression discrète*>)

sémantique: un mode matrice dynamique est un mode matrice paramétré de **borne supérieure** non **constante**.

propriétés statiques: les modes matrice dynamiques ont les mêmes propriétés que les modes matrice, sauf en ce qui concerne les propriétés ci-après.

propriétés dynamiques:

- a un mode matrice dynamique sont attachés une **borne supérieure** dynamique qui est la valeur rendue par l'*expression discrète* et un **nombre d'éléments** dynamique qui est la valeur rendue par:

  NUM (*expression discrète*) – *NUM* (*borne inférieure*) + 1

  où *borne inférieure* est la **borne inférieure** du *nom de mode matrice d'origine.*

3.14.4 Modes structure paramétrés dynamiques

dénotation virtuelle: &<*nom de mode structure variable originel*> (<*liste d'expressions*>)

sémantique: un mode structure **paramétré** dynamique est un mode structure **paramétré** aux paramètres non **constants**.

propriétés statiques: les propriétés statiques d'un mode structure **paramétré** dynamique sont les mêmes que celles d'un mode structure paramétré statique sauf en ce qui concerne la propriété suivante:

- l'ensemble des noms **de champ** d'un mode structure **paramétré** dynamique est l'ensemble des noms **de champ** de son mode structure **variable originel**.

propriétés dynamiques:

- a un mode structure **paramétré** dynamique est attachée une liste de valeurs qui est la liste de valeurs rendues par les expressions de la *liste d'expressions.*

3.15 Modes Moreta

3.15.1 Généralités

syntaxe:

<mode moreta> ::= *(1)*
- *<mode module>* *(1.1)*
- | *<mode région>* *(1.2)*
- | *<mode tâche>* *(1.3)*
- | *<instanciation de mode moreta générique>* *(1.4)*
- | *<mode interface>* *(1.5)*
- | *<nom de mode moreta>* [(*<liste des paramètres effectifs>*)] *(1.6)*

sémantique:

mode module – Un locus de *mode module* a les mêmes propriétés qu'un *module* sans *liste des énoncés d'action*.

mode région – Un locus de *mode région* a les mêmes propriétés qu'une *région*.

mode tâche – Un locus de *mode tâche* a essentiellement la même structure qu'un *locus de mode module sans définitions de procédure*. Les accès directs aux composantes d'un locus dont le mode est un *mode tâche* s'excluent mutuellement. Un locus dont le mode est un *mode tâche*, peut être exécuté en même temps que d'autres fils d'exécution (voir 11.1).

instanciation de mode moreta générique – Une instanciation de mode moreta générique est obtenue statistiquement par instanciation d'un gabarit de mode moreta générique (voir 10.11).

interface de mode – Une *interface de mode* est constituée de spécifications et de signatures seulement.

conditions statiques:

Les *modes moreta* ne sont pas paramétrables.

Les *modes moreta* et les *gabarits de mode moreta générique* ne peuvent pas être imbriqués.

(1.1) – (1.5) sont uniquement permis dans les définitions synmode et néomode. Autrement dit, les modes moreta anonymes ne sont pas acceptés.

3.15.2 Modes module

syntaxe:

<mode module> ::= *(1)*
<spécification de mode module> *(1.1)*
| *<corps de mode module>* *(1.2)*

<spécification de mode module> ::= *(2)*
MODULE SPEC [[**ASSIGNABLE** [**FINAL**] | **ABSTRACT**] |
[**NOT_ASSIGNABLE** [**ABSTRACT** | **FINAL**]]]
<clause d'héritage de module>
{*<composante de spécification de module>*}* [*<partie invariable>*]
END [*<chaîne de nom simple>*] *(2.1)*

<corps de mode module> ::= *(3)*
MODULE BODY [[**ASSIGNABLE** [**FINAL**] | **ABSTRACT**] |
[**NOT_ASSIGNABLE** [**ABSTRACT** | **FINAL**]]]
<clause d'héritage de module>
{*<composante de corps de module>*}* [*<partie invariable>*]
END [*<filet>*] [*<chaîne de nom simple>*] *(3.1)*

<clause d'héritage de module> ::= *(4)*
[*<héritage de module>*] [*<clause d'implémentation>*] *(4.1)*

<héritage de module> ::= *(5)*
BASED_ON *<nom de mode module>* *(5.1)*

<clause d'implémentation> ::= *(6)*
IMPLEMENTS *<interface de nom de mode>* { , *<interface de nom de mode>* }* *(6.1)*

<composante de spécification de module> ::= *(7)*
<composante de module commune> *(7.1)*
| *<énoncé déclaratif>* *(7.2)*
| *<énoncé de signature de procédure protégée simple>* *(7.3)*
| *<énoncé de déclaratif de procédure protégée alignée>* *(7.4)*
| *<énoncé de spécification de processus>* *(7.5)*
| *<énoncé de définition de signal>* *(7.6)*
| *<énoncé d'octroi>* *(7.7)*

<composante de corps de module> ::= *(8)*
<composante de module commune> *(8.1)*
| *<énoncé de définition de procédure protégée simple>* *(8.2)*
| *<énoncé de définition de processus>* *(8.3)*

<composante de module commune> ::= *(9)*
<énoncé de définition synonyme> *(9.1)*
| *<énoncé de définition synmode>* *(9.2)*
| *<énoncé de définition néomode>* *(9.3)*
| *<énoncé de saisie>* *(9.4)*

<partie invariable> ::= *(10)*
INVARIANT *<expression booléenne>* *(10.1)*

sémantique: un mode module définit des valeurs composites formées d'une liste de composantes sélectionnables par des noms de composante.

Les valeurs module peuvent résider dans des locus **module** (composites).

Un *mode module* est défini au moyen de deux parties: une *spécification de mode module* et un *corps de mode module*.

La partie **spécification** définit l'interface des valeurs du *mode module*.

La partie **corps** définit le comportement des valeurs du *mode module.*

L'*expression booléenne* de la *partie invariable* doit être "Vraie" avant et après tout appel d'un procédure de composante **publique** ou d'un processus de composante **publique**.

propriétés statiques: si l'attribut **ASSIGNABLE** est spécifié, le mode est un mode module **assignable**. Un mode module **assignable** peut être utilisé de la même manière qu'un mode pour lequel **READ** n'est pas spécifié (voir 3.3).

Si l'attribut **NOT_ASSIGNABLE** est spécifié, le mode a la propriété **non assignable**, ce qui indique que le locus de ce mode n'est pas accessible pour stocker la valeur ou pour copier sa valeur.

Si aucun mode n'est spécifié – ni **ASSIGNABLE**, ni **NOT_ASSIGNABLE –** le mode est **non assignable** par défaut.

Si l'attribut **ABSTRACT** est spécifié, le mode est un mode **abstrait**.

Si un *héritage module* est donné, le mode MD en cours de définition est directement extrait du mode MB donné dans *héritage module*, et MB est le mode de base direct de MD.

Si une *clause d'implémentation* IC est donnée, le mode MD en cours de définition est directement extrait des modes donnés dans IC, et ces modes sont des modes de base directs de MD.

L'effet de la *clause d'héritage module* est que le mode extrait se comporte comme s'il contenait toutes les composantes de ses modes de base directs, sauf les procédures des composantes constructeur et destructeur. Si l'un de ces modes de base est lui-même un mode extrait, cet héritage de composantes doit être tenu pour transitoire. En ce qui concerne la visibilité, voir 12.2.

Une *composante de spécification de module* contenue dans une *spécification de mode module* M_S ou SEIZEd dans M_S, qui est octroyée par M_S, est appelée une composante **publique** du mode de M_S.

Une *composante de spécification de module* contenue dans une *spécification de mode module* M_S ou SEIZEd dans M_S, qui n'est pas octroyée par M_S, est appelée une composante **interne** du mode de M_S.

Une *composante de corps de module* C contenue dans un *corps de mode module* M_B ou SEIZEd dans M_B est appelée une composante **privée** du mode de M_B si C est une composante ni **publique**, ni **interne** du mode de M_B.

Un mode module **abstrait** a la propriété **non assignable**.

conditions statiques: un mode module ne peut pas être utilisé comme mode dans une *définition de synonyme*.

Pour chaque *spécification de mode module*, il faut un *corps de mode module* ayant la même chaîne dans *l'occurence de définition.*

Si elle est spécifiée, la *chaîne de nom simple* après **END** doit concorder avec la chaîne de nom de l'occurence de définition de cette définition de mode. Cela s'applique aussi à *spécification de mode module* et à *corps de mode module*.

Si l'un des attributs **ASSIGNABLE**, **NOT_ASSIGNABLE**, **ABSTRACT** ou **FINAL** est spécifié dans une *spécification de mode module*, il doit aussi être spécifié dans le *corps de mode module* correspondant.

Si une *spécification de mode module* contient une *clause d'héritage module*, le *corps de mode module* correspondant doit contenir la même *clause d'héritage module*.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans une *signature de procédure protégée simple*, cette procédure a la propriété **incomplète**.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée simple*, cette procédure doit être **publique**.

Pour chaque *énoncé de signature de procédure protégée* **simple**, **complet** S d'une *spécification de mode module*, le *corps de mode module* correspondant doit contenir un *énoncé de définition de procédure protégée simple* D dans lequel la *signature de procédure protégée* de S concorde avec la *définition de procédure protégée* de D (voir 12.1.3).

Si P est une *signature de procédure protégée* **simple**, **incomplète** d'une *spécification de mode module*, le *corps de mode module* correspondant ne doit pas contenir de *définition de procédure protégée simple* concordant avec P.

Pour chaque *spécification de processus* d'une *spécification de mode module*, *corps de mode module* correspondant doit contenir une *définition de processus* correspondante (voir 12.1.3).

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée simple*, cette procédure doit être **publique**.

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée simple* PD contenu dans une *spécification de mode module* M, le mode de base direct MB de M doit contenir ou avoir hérité un *énoncé de signature de procédure protégée simple* PB **public**, où PB concorde avec PD, où PB est ni un constructeur ni un destructeur, et où PB n'est pas SEIZEd.

Un *mode module* est un mode module **abstrait** s'il contient au moins une *procédure de composante* **incomplète** (voir 10.4). Dans ce cas, il faut spécifier l'attribut **ABSTRACT**.

Un nom de mode module **abstrait** peut uniquement être utilisé en tant que *nom* du mode module dans un *héritage module* ou dans un *mode référencé*.

Si un mode module M a au moins une (sous-) composante ayant la **propriété de non-valeur**, M a également la **propriété de non-valeur** est l'attribut **ASSIGNABLE** ne doit pas être spécifié (voir 12.1.15).

Si un *mode module* M contient l'attribut **FINAL**, M est appelé un mode module **final**. Un mode module **final** ne peut pas être utilisé comme mode de base dans un *héritage* moreta.

Un mode module **final** ne doit pas contenir de procédure de composante **incomplète**.

3.15.3 Modes région

syntaxe:

<mode région> ::= *(1)*
<spécification de mode région> *(1.1)*
| *<corps de mode région>* *(1.2)*

<spécification de mode région> ::= *(2)*
REGION SPEC [**ABSTRACT** | **FINAL**] [*<héritage région>*]
{*<composante de spécification de région>*}* [*<partie invariable>*]
END [*<chaîne de nom simple>*] *(2.1)*

<corps de mode région> ::= *(3)*
REGION BODY [**ABSTRACT** | **FINAL**] [*<clause d'héritage région>*]
{*<composante de corps région>*}* [*<partie invariable>*]
END [*<filet>*] [*<chaîne de nom simple>*] *(3.1)*

<clause d'héritage région> ::= *(4)*
[*<héritage région>*] [*<clause d'implémentation>*] *(4.1)*

<héritage région> ::= *(5)*
BASED_ON {*<nom de mode module>* | *<nom de mode région>*} *(5.1)*

<composante de spécification de région>::= *(6)*
<composante de module commune> *(6.1)*
| *<énoncé déclaratif>* *(6.2)*
| *<énoncé de signature de procédure protégée simple>* *(6.3)*
| *<énoncé de définition de signal>* *(6.4)*
| *<énoncé d'octroi> ::=* *(6.5)*

<composante de corps région> ::= *(7)*
<composante de module commune> *(7.1)*
| *<énoncé de définition de procédure protégée simple>* *(7.2)*

sémantique: un *mode région* définit des valeurs composites formées d'une liste de composantes sélectionnables par des noms de composante.

Les valeurs région peuvent résider dans des locus région (composites).

Un *mode région* est défini au moyen de deux parties: une *spécification de mode région* et un *corps de mode région*.

La partie **spécification** définit l'interface des valeurs du *mode région*.

La partie **corps** définit le comportement des valeurs du *mode région*.

L'*expression booléenne* de la *partie invariable* doit être "Vraie" avant et après tout appel d'un procédure de composante **publique**.

propriétés statiques: un *mode région* a la propriété **non_assignable**.

Si l'attribut **ABSTRACT** est spécifié, le mode est un mode **abstrait**.

Si un *héritage région* est donné, le mode MD en cours de définition est directement extrait du mode MB donné dans *héritage région*, et MB est le mode de base direct de MD.

Si une *clause d'implémentation* IC est donnée, le mode MD en cours de définition est directement extrait des modes donnés dans IC, et ces modes sont des modes de base directs de MD.

L'effet de la *clause d'héritage région* est que le mode extrait se comporte comme s'il contenait toutes les composantes de ses modes de base directs, sauf les procédures des composantes constructeur et destructeur. Si l'un de ces modes de base est lui-même un mode extrait, cet héritage de composantes doit être tenu pour transitoire. En ce qui concerne la visibilité, voir 12.2.

Une *composante de spécification de région* contenue dans une *spécification de mode région* M_S ou SEIZEd dans M_S, qui est octroyée par M_S, est appelée une composante **publique** du mode de M_S.

A *composante de spécification de région* contenue dans une *spécification de mode région* M_S ou SEIZEd dans M_S, qui n'est pas octroyée par M_S, est appelée une composante **interne** du mode de M_S.

Une *composante de corps de région* C contenue dans un *corps de mode région* M_B ou SEIZEd dans M_B est appelée une composante **privée** du mode de M_B si C est une composante ni **publique**, ni **interne** du mode de M_B.

conditions statiques: un mode région ne peut pas être utilisé comme mode dans une *définition de synonyme*.

Pour chaque *spécification de mode région*, il faut un *corps de mode région* ayant la même chaîne dans *l'occurence de définition*.

Si elle est spécifiée, la *chaîne de nom simple* après **END** doit concorder avec la chaîne de nom de *l'occurence de définition* de cette définition de mode. Cela s'applique aussi à *spécification de mode région* et à *corps de mode région*.

Si l'un des attributs **ABSTRACT** ou **FINAL** est spécifié dans une *spécification de mode région*, il doit aussi être spécifié dans le *corps de mode région* correspondant.

Si une *spécification de mode région* contient une *clause d'héritage région*, le *corps de mode région* correspondant doit contenir la même *clause d'héritage région*.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans une *signature de procédure protégée* *simple*, cette procédure a la propriété **incomplète**.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée* *simple*, cette procédure doit être **publique**.

Pour chaque *énoncé de signature de procédure protégée* **simple**, **complet** S d'une *spécification de mode région*, le *corps de mode région* correspondant doit contenir un *énoncé de définition de procédure protégée* *simple* D dans lequel la *signature de procédure protégée* de S concorde avec la *définition de procédure protégée* de D (voir 12.1.3).

Si P est une *signature de procédure protégée* **simple, incomplète** d'une *spécification de mode région*, le *corps de mode région* correspondant ne doit pas contenir de *définition de procédure protégée* *simple* concordant avec P.

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée* *simple*, cette procédure doit être **publique**.

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée* *simple* PD contenu dans une *spécification de mode région* M, le mode de base direct MB de M doit contenir ou avoir hérité un *énoncé de signature de procédure protégée* *simple* PB **public**, où PB concorde avec PD, où PB est ni un constructeur ni un destructeur et où PB n'est pas SEIZEd.

Un *mode région* est un mode région **abstrait** s'il contient au moins une *procédure de composante* **incomplète** (voir 10.4). Dans ce cas, il faut spécifier l'attribut **ABSTRACT**.

Un nom de mode région **abstrait** peut uniquement être utilisé en tant que *nom* du mode région dans un *héritage région* ou dans un *mode référencé*.

Une spécification de *mode région* ne doit pas octroyer de locus.

Si le mode de base d'un *mode région* est un *mode module* M, M ne doit pas avoir la propriété **non_assignable**, ne doit octroyer aucun locus et ne doit contenir aucune *procédure de composante protégée* *en ligne* ou *processus de composante*.

Si un *mode région* M contient l'attribut **FINAL**, M est appelé un mode région **final**. Un mode région final ne peut pas être utilisé comme mode de base dans un *héritage région*.

Un mode région **final** ne doit pas contenir de procédure de composante **incomplète**.

3.15.4 Modes tâche

syntaxe:

<mode tâche> ::= *(1)*
<spécification de mode tâche> *(1.1)*
| *<corps de mode tâche>* *(1.2)*

<spécification de mode tâche> ::= *(2)*
TASK SPEC [**ABSTRACT** | **FINAL**] [*<clause d'héritage tâche>*]
[*<partie invariable>*] {*<composante de spécification de tâche>*}
END [*<chaîne de nom simple>*] *(2.1)*

<corps de mode tâche> ::= *(3)*
TASK BODY [**ABSTRACT** | **FINAL**] [*<clause d'héritage tâche>*]
{*<composante de corps de tâche>*}* [*<partie invariable>*]
END [*<filet>*] [*<chaîne de nom simple>*] *(3.1)*

<clause d'héritage tâche> ::= *(4)*
[*<héritage tâche>*] [*<clause d'implémentation>*] *(4.1)*

<héritage tâche> ::= *(5)*
BASED_ON {*<nom de mode module>* | *<nom de mode tâche>*} *(5.1)*

<composante de spécification de tâche> ::= *(6)*
<composante de spécification de région> *(6.1)*

<composante de corps de tâche> ::= *(7)*
<composante de corps région> *(7.1)*

sémantique: un *mode tâche* définit des valeurs composites formées d'une liste de composantes sélectionnables par des noms de composante.

Les valeurs tâche peuvent résider dans des locus tâche (composites).

Une *mode tâche* est défini au moyen de deux parties: une *spécification de mode tâche* et un *corps de mode tâche*.

La partie **spécification** définit l'interface des valeurs du *mode tâche*.

La partie **corps** définit le comportement des valeurs du *mode tâche*.

L'*expression booléenne* de la *partie invariable* doit être vraie avant et après tout appel d'une procédure de composante **publique**.

propriétés statiques: un *mode tâche* a la propriété **non assignable**.

Si l'attribut **ABSTRACT** est spécifié, le mode est un mode **abstrait**.

Si un *héritage tâche* est donné, le mode MD en cours de définition est directement extrait du mode MB donné dans *héritage tâche*, et MB est le mode de base direct de MD.

Si une *clause d'implémentation* IC est donnée, le mode MD en cours de définition est directement extrait des modes donnés dans IC, et ces modes sont des modes de base directs de MD.

L'effet de la *clause d'héritage tâche* est que le mode extrait se comporte comme s'il contenait toutes les composantes de ses modes de base directs, sauf les procédures des composantes constructeur et destructeur. Si l'un de ces modes de base est lui-même un mode extrait, cet héritage de composantes doit être tenu pour transitoire. En ce qui concerne la visibilité, voir 12.2.

Une *composante de spécification de tâche* contenue dans une *spécification de mode tâche* M_S ou SEIZEd dans M_S, qui est octroyée par M_S, est appelée une composante **publique** du mode de M_S.

Une *composante de spécification de tâche* contenue dans une *spécification de mode tâche* M_S ou SEIZEd dans M_S, qui n'est pas octroyée par M_S, est appelée une composante **interne** du mode de M_S.

Une *composante de corps de tâche* C contenue dans un *corps de mode tâche* M_B ou SEIZEd dans M_B est appelée une composante **privée** du mode de M_B si C est une composante ni **publique**, ni **interne** du mode de M_B.

conditions statiques: un mode tâche ne peut pas être utilisé comme mode dans une *définition de synonyme*.

Pour chaque *spécification de mode tâche*, il faut un *corps de mode tâche* ayant la même dans *l'occurence de définition*.

Si elle est spécifiée, la chaîne *de nom simple* après **END** doit concorder avec la chaîne de nom de *l'occurence de définition* de cette définition de mode. Cela s'applique aussi à *spécification de mode tâche* et à *corps de mode tâche*.

Si l'un des attributs **ABSTRACT** ou **FINAL** est spécifié dans une *spécification de mode tâche*, il doit aussi être spécifié dans le *corps de mode tâche* correspondant.

Si une *spécification de mode tâche* contient une *clause d'héritage tâche*, le *corps de mode tâche* correspondant doit contenir la même *clause d'héritage tâche*.

Toutes les procédures de composante publique d'un mode tâche doivent avoir uniquement des paramètres IN et ne doivent pas avoir de *spec de résultat*.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans une *signature de procédure protégée simple*, cette procédure a la propriété **incomplète**.

Si l'attribut **INCOMPLETE** (voir 10.4) est spécifié dans un énoncé de *signature de procédure protégée simple*, cette procédure doit être **publique**.

Pour chaque *énoncé de signature de procédure protégée* **simple**, **complet** S d'une *spécification de mode tâche*, le *corps de mode tâche* correspondant doit contenir un *énoncé de définition de procédure protégée simple* D dans lequel la *signature de procédure protégée* de S concorde avec la *définition de procédure protégée* de D (voir 12.1.3).

Si P est une *signature de procédure protégée* **simple**, **incomplète** d'une *spécification de mode tâche*, le *corps de mode tâche* correspondant ne doit pas contenir de *définition de procédure protégée simple* concordant avec P.

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée simple*, cette procédure doit être **publique**.

Si l'attribut **REIMPLEMENT** (voir 10.4) est spécifié dans un *énoncé de signature de procédure protégée simple* PD contenu dans une *spécification de mode tâche* M, le mode de base direct de M doit contenir ou avoir hérité un *énoncé de signature de procédure protégée simple* PB **public**, où PB concorde avec PD, où PB est ni un constructeur ni un destructeur, et où PB n'est pas SEIZEd.

Un *mode tâche* est un mode tâche **abstrait** s'il contient au moins une *procédure de composante* **incomplète** (voir 10.4). Dans ce cas, il faut spécifier l'attribut **ABSTRACT**.

Un nom de mode tâche **abstrait** peut uniquement être utilisé en tant que *nom* du mode tâche dans un *héritage tâche* ou dans un *mode référencé*.

Une spécification de *mode tâche* ne doit pas octroyer de locus.

Si le mode de base direct d'un *mode tâche* est un *mode module* M, M ne doit pas avoir la propriété **non_assignable**, ne doit octroyer aucun locus et ne doit contenir aucune *procédure de composante protégée en ligne* ou *processus de composante*, et doit contenir uniquement des procédures **publiques** qui satisfont aux restrictions des procédures de composante **publiques** des modes tâche.

Si un *mode tâche* M contient l'attribut **FINAL**, M est appelé un mode tâche **final**. Un mode tâche **final** ne peut pas être utilisé comme mode de base dans un *héritage tâche*.

Un mode tâche **final** ne doit pas contenir de procédure de composante **incomplète**.

3.15.5 Modes interface

syntaxe:

<mode interface> ::= *(1)*
INTERFACE [*<héritage interface>*] {*<composante d' interface>*}*
END [*<chaîne de nom simple>*] *(1.1)*

<héritage interface> ::= *(2)*
BASED_ON *<nom de mode interface>* { , *<nom de mode interface>* }* *(2.1)*

<composante d' interface> ::= *(3)*
<composante de module commune> *(3.1)*
| *<énoncé déclaratif>* *(3.2)*
| *<énoncé de signature de procédure protégée simple>* *(3.3)*
| *<énoncé de spécification de processus>* *(3.4)*
| *<énoncé de définition de signal>* *(3.5)*

sémantique: un *mode interface* définit un mode moreta qui peut être utilisé uniquement comme mode de base dans la définition d'autres modes moreta et comme mode référencé d'un mode référence liée.

propriétés statiques:

si un *héritage interface* II est donné, le mode MD en cours de définition est directement extrait des modes contenus dans II, et ces modes sont des modes de base directs de MD.

L'effet de *héritage interface* est que le mode extrait se comporte comme s'il contenait toutes les composantes de ses modes de base directs. Si l'un de ces modes de base est lui-même un mode extrait, cet héritage de composantes doit être tenu pour transitoire. En ce qui concerne la visibilité, voir 12.2.

Toutes les *composantes interface* (y compris celle qui sont SEIZEd) sont implicitement GRANTed, raison pour laquelle elles sont toutes appelées des composantes **publiques**.

Un mode interface est un mode **abstrait**.

conditions statiques: un mode interface ne peut pas être utilisé comme mode dans une *définition de synonyme*.

Si elle est spécifiée, la *chaîne de nom simple* après **END** doit concorder avec la chaîne de nom de l'occurence de définition de cette définition de mode.

L'attribut **INCOMPLETE** (voir 10.4) doit être spécifié dans toutes les *signatures de procédure protégée simple;* toutes les procédures ont la propriété **incomplète**.

L'attribut **REIMPLEMENT** (voir 10.4) ne doit pas être spécifié dans un *énoncé de signature de procédure protégée simple* d'une *composante d'interface*.

4 Les locus et leurs accès

4.1 Déclarations

4.1.1 Généralités

syntaxe:

<énoncé déclaratif> ::= *(1)*
DCL *<déclaration>* { , *<déclaration>* }* ; *(1.1)*

<déclaration> ::= *(2)*
<déclaration de locus> *(2.1)*
| *<déclaration d'identité de locus>* *(2.2)*

sémantique: un énoncé déclaratif déclare qu'un ou plusieurs noms sont un accès à un locus.

exemples:

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 Déclarations de locus

syntaxe:

<déclaration de locus> ::= *(1)*
<liste d'occurences de définitions> *<mode>* [**STATIC**] [*<initialisation>*] *(1.1)*

<initialisation> ::= *(2)*
<initialisation domaniale> *(2.1)*
| *<initialisation à vie>* *(2.2)*
| *<initialisation de type moreta>* *(2.3)*

<initialisation domaniale> ::= *(3)*
<symbole d'affectation> *<valeur>* [*<filet>*] *(3.1)*

<initialisation à vie> ::= *(4)*
INIT *<symbole d'affectation>* *<valeur constante>* *(4.1)*

<initialisation de type moreta> ::= *(5)*
([*<liste de paramètres effectifs de constructeur>*]) [*<filet>*] *(5.1)*

sémantique: une déclaration de locus crée autant de locus qu'on spécifie l'occurence de définitions apparaissant dans la *liste d'occurences de définitions.*

Pour une *initialisation domaniale*, la *valeur* est évaluée chaque fois qu'on entre dans le domaine dans lequel la déclaration est placée (voir 10.2) et la valeur obtenue est affectée au(x) locus. Avant que la *valeur* ne soit évaluée, le ou les locus contiennent une valeur **non définie**.

Pour une *initialisation à vie*, la valeur délivrée par la *valeur constante* est affectée au(x) locus une fois seulement au début de sa (leur) durée de vie (voir 10.2 et 10.9).

Si le mode est un *mode moreta* toutes les initialisations des composantes sont effectuées en premier lieu, dans l'ordre textuel de leur apparition. Si une liste de paramètres (éventuellement vide) est spécifiée, le **constructeur** correspondant du *mode* est appliqué au locus nouvellement créé. Si le *mode* est un *mode tâche*, la tâche appartenant aux locus nouvellement créés est lancée.

Ne pas spécifier d'*initialisation* est équivalent, sémantiquement, à la spécification d'une *initialisation à vie* avec la valeur **non définie** (voir 5.3.1).

La signification de la valeur **non définie** en tant qu'initialisation pour un locus auquel est attaché un mode avec la **propriété d'étiquetage et de paramétrage** ou la **propriété de non-valeur** est la suivante:

- **propriété d'étiquetage et de paramétrage**: le ou les sous-locus de champ avec **étiquettes** créés sont initialisés avec la valeur de paramètre correspondante.
- **propriété de non-valeur**:
  - l'événement créé et le ou les (sous-)locus tampon sont initialisés comme étant "vides", c'est-à-dire qu'aucun processus retard ne s'attache à l'événement ou au tampon et qu'il n'y a pas de messages dans le tampon;
  - la région et les (sous-) locus de tâche créés sont initialisés comme étant "vides", c'est-à-dire qu'aucun fil d'exécution n'y est rattaché;
  - le ou les (sous-)locus d'association créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne contiennent pas d'association;
  - le ou les (sous-)locus d'accès créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne sont pas connectés à une association;
  - le ou les (sous-)locus texte créés ont un sous-locus **enregistrement texte** qui est initialisé avec une chaîne vide et un sous-locus **accès** qui est initialisé comme étant "vide", c'est-à-dire qu'il n'est pas connecté à une association.
- La sémantique de **STATIC** et de *filet* sera trouvée, respectivement au 10.9 et à l'article 8.

Si la durée de vie d'un locus L de **moreta** se termine et que le mode du locus contient un destructeur, celui-ci est appliqué à L (voir 10.2).

propriétés statiques: une *occurence de définition* apparaissant dans une *déclaration de locus* définit un nom de **locus**. Le mode attaché au nom de **locus** est le *mode* spécifié dans la *déclaration de locus.* Un nom de **locus** est **référençable**.

conditions statiques: la classe de la *valeur* ou *valeur constante* doit être **compatible** avec le *mode* et la valeur obtenue doit être une des valeurs définies par le *mode*, ou la valeur **non définie**.

Si le *mode* a la **propriété de protection**, *initialisation* doit être spécifiée. Si le *mode* a la **propriété de non-valeur**, on ne peut pas spécifier d'*initialisation domaniale*.

Si *initialisation* est spécifiée, la *valeur* doit être **régionalement sûre** pour le locus (voir 11.2.2).

conditions dynamiques: dans le cas d'une *initialisation domaniale*, les conditions d'affectation doivent être respectées par *valeur* en tenant compte du *mode* (voir 6.2).

exemples:

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 Déclarations d'identité de locus

syntaxe:

<déclaration d'identité de locus> ::= *(1)*
<liste d'occurences de définitions> <mode> **LOC** [**DYNAMIC**]
<symbole d'affectation> <locus> [*<filet>*] *(1.1)*

sémantique: une déclaration d'identité de locus crée autant de noms d'accès au locus spécifié qu'il y a *d'occurences de définitions* spécifiées dans la *liste d'occurences de définitions.* Le mode du locus ne peut être dynamique que si **DYNAMIC** est spécifié.

Si le *locus* est évalué dynamiquement, cette évaluation se fait chaque fois que le domaine, dans lequel la déclaration d'identité de locus est placée, est entamé. Dans ce cas, un nom déclaré dénote un locus **indéfini** avant la première évaluation durant la durée de vie de l'accès dénoté par le nom déclaré (voir 10.2 et 10.9).

propriétés statiques: une *occurence de définition* apparaissant dans une *déclaration d'identité de locus* définit un nom d'**identité de locus**. Le mode qui s'attache à un nom d'**identité de locus** est, si **DYNAMIC** n'est pas spécifié, le *mode* spécifié dans la *déclaration d'identité de locus*; sinon, c'est une version paramétrée dynamiquement de celui-ci, qui a les mêmes paramètres que le mode du *locus*.

Il n'est pas permis de créer un locus de mode moreta ayant la propriété **DYNAMIC**.

Un nom d'**identité de locus** est **référençable** si et seulement si le *locus* spécifié est **référençable**.

conditions statiques: si **DYNAMIC** est spécifié dans la *déclaration d'identité de locus*, le *mode* doit être **paramétrable**. Le *mode* spécifié doit être **compatible en lecture dynamique** avec le mode *locus* si **DYNAMIC** est spécifié et, dans les autres cas, **compatible en lecture** avec le mode *locus*.

Le locus ne doit pas être un *élément de chaîne* ni un *segment de chaîne* dans lequel le mode *locus* *chaîne* est un mode chaîne **variable**.

conditions dynamiques: l'exception *RANGEFAIL* ou *TAGFAIL* se produit si **DYNAMIC** est spécifié et si le contrôle **compatible en lecture dynamique** susmentionné est négatif.

exemple:

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

4.2 Les locus

4.2.1 Généralités

syntaxe:

<locus> ::= *(1)*
<nom d'accès> *(1.1)*
| *<référence liée déréférencée>* *(1.2)*
| *<référence libre déréférencée>* *(1.3)*
| *<descripteur déférencé>* *(1.4)*
| *<élément de chaîne>* *(1.5)*
| *<segment de chaîne>* *(1.6)*
| *<élément de matrice>* *(1.7)*
| *<bande matricielle>* *(1.8)*
| *<champ de structure>* *(1.9)*
| *<appel de procédure rendant locus>* *(1.10)*
| *<appel de routine prédéfinie rendant locus>* *(1.11)*
| *<conversion de locus>* *(1.12)*
| *<locus moreta prédéfini>* *(1.13)*

sémantique: un locus est un objet qui peut contenir des valeurs. Il faut accéder aux locus pour y placer ou en obtenir une valeur.

propriétés statiques: un *locus* a les propriétés suivantes:

- Il a un mode, tel que défini dans les sections appropriées. Ce mode est statique ou dynamique.
- Il peut être **statique** ou non (voir 10.9).
- Il peut être **intrarégional** ou **extrarégional** (voir 11.2.2).
- Il peut être **référençable** ou non. La définition du langage exige que certains locus soient **référençables** et que d'autres ne le soient pas, comme indiqué dans les sections appropriées. Une implémentation peut étendre la référenciation à d'autres locus sauf quand elle est explicitement interdite.

4.2.2 Noms d'accès

syntaxe:

<nom d'accès> ::= *(1)*
<nom de locus> *(1.1)*
| *<nom d'identité de locus>* *(1.2)*
| *<nom d'énumération de locus>* *(1.3)*
| *<nom de locus faire-avec>* *(1.4)*

sémantique: un nom d'accès donne un locus. Un nom d'accès entre dans une des catégories suivantes:

- un nom **de locus**, c'est-à-dire un nom déclaré explicitement dans une *déclaration de locus* ou déclaré implicitement dans un *paramètre formel* sans l'attribut **LOC**;
- un nom **d'identité de locus**, c'est-à-dire un nom déclaré explicitement dans une *déclaration d'identité de locus* ou déclaré implicitement dans un *paramètre formel* avec l'attribut **LOC**;
- un nom **d'énumération de locus**, c'est-à-dire un *compteur de boucle* dans une *énumération de locus*;
- un nom **de locus faire-avec**, c'est-à-dire un nom de **champ** employé comme accès direct dans l'*action faire* avec une *partie avec*.

Si le locus dénoté par un *nom de locus faire-avec* est un champ récurrent d'un locus structure variable sans étiquettes, la sémantique est définie par l'implémentation.

propriétés statiques: le mode (éventuellement dynamique) attaché à un *nom d'accès* est respectivement le mode du *nom de locus*, du *nom d'identité de locus*, du *nom d'énumération de locus* ou du *nom de locus faire-avec*.

Un *nom d'accès* est **référençable** si et seulement si c'est un *nom de locus*, un *nom d'identité de locus* **référençable**, un *nom d'énumération de locus* **référençable**, ou un *nom de locus faire-avec* **référençable**.

conditions dynamiques: quand on accède à un locus via un *nom d'identité de locus*, il ne peut pas dénoter un locus **indéfini**.

En cas d'accès, via un nom *d'identité de locus*, à un locus qui est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir 4.2.10). Accéder à un locus via un *nom de locus faire-avec* cause l'exception *TAGFAIL* si le locus dénoté est un champ **récurrent** et si les conditions d'accès au champ récurrent pour le locus ne sont pas satisfaites.

exemples:

4.12 *a* *(1.1)*

11.39 *starting* *(1.2)*

15.35 *each* *(1.3)*

5.10 *c1* *(1.4)*

4.2.3 Références liées déférencées

syntaxe:

<référence liée déférencée> ::= *(1)*
<valeur primitive référence liée> –> [*<nom de mode>*] *(1.1)*

sémantique: une référence liée déférencée donne le locus qui est référencée par la valeur référence liée.

propriétés statiques: le mode attaché au *référence liée déférencée* est le *nom de mode* s'il y en a un, sinon le mode **référencé** du mode de la *valeur primitive référence liée*. Une *référence liée déférencée* est **référençable**.

conditions statiques: la *valeur primitive référence liée* doit être **forte**. Si le *nom de mode* optionnel est spécifié, il doit être **compatible en lecture** avec le mode **référencé** du mode de la *valeur primitive référence liée*.

conditions dynamiques: la durée de vie du locus référencé ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive référence liée* donne la valeur *NULL*.

Si le locus référencé est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir 4.2.10).

exemple:

10.54 p –> *(1.1)*

4.2.4 Références libres déréférencées

syntaxe:

<référence libre déréférencées> ::= *(1)*
<valeur primitive référencé libre> –> <nom de mode> *(1.1)*

sémantique: un référence libre déréférencée donne le locus qui est référencé par la valeur référence libre.

propriétés statiques: le mode attaché à une *référence libre déréférencée* est le *nom de mode*. Une *référence libre déréférencée* est **référençable**.

conditions statiques: la *valeur primitive référence libre* doit être **forte**.

conditions dynamiques: la durée de vie du locus déréférencée ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive référence libre* donne la valeur *NULL*.

Le *nom de mode* doit être **compatible en lecture** avec le mode du locus référencé.

Si le locus référencé est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir 4.2.10).

4.2.5 Références ligne déréférencées

syntaxe:

<référence ligne déréférencée> ::= *(1)*
<valeur primitive référence ligne> –> *(1.1)*

sémantique: un descripteur déréférencée donne le locus qui est référencé par la valeur descripteur.

propriétés statiques: le mode dynamique attaché à un *descripteur déréférencé* est construit comme suit:

&<nom de mode originel> (*<paramètre>* { , *<paramètre>* }*)

où le *&nom de mode originel* est un nom virtuel de **synmode synonyme** du mode **référencé originel** du mode de la *valeur primitive descripteur* et où les paramètres sont, selon le mode **référencé originel**:

- la **longueur de la chaîne** dynamique, dans le cas d'un mode chaîne;
- la **borne supérieure** dynamique, dans le cas d'un mode matrice;
- la liste des valeurs associées au mode du locus de structure paramétré, dans le cas d'un mode structure **variable**.

Un *descripteur déréférencé* est **référençable**.

conditions statiques: la *valeur primitive descripteur* doit être **forte**.

conditions dynamiques: la durée de vie du locus référencé ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive descripteur* donne *NULL*.

Si le locus référencé est un champ **récurrent**, les conditions d'accès au champ récurrent du locus doivent être satisfaites (voir 4.2.10).

exemple:

8.11 input –> *(1.1)*

4.2.6 Eléments de chaîne

syntaxe:

<élément de chaîne> ::= *(1)*
<locus chaîne> (<élément de début>) *(1.1)*

<élément de début> ::= *(2)*
<expression entière> *(2.1)*

sémantique: un élément de chaîne fournit un (sous-)locus qui est l'élément du locus de chaîne spécifié indiqué par un *élément de départ*.

propriétés statiques: le mode attaché à l'*élément de chaîne* est le mode de l'**élément** du *locus de chaîne*.

Si le mode du *locus chaîne* est un mode chaîne **variable**, l'*élément de chaîne* n'est pas **référençable**.

conditions dynamiques: l'exception *RANGEFAIL* se produit si la relation suivante n'est pas vérifiée:

$$0 \leq NUM(\text{élément de début}) \leq L - 1$$

où L est la **longueur effective** du *locus chaîne*.

exemple:

18.16 string –> (i) *(1.1)*

4.2.7 Segments de chaîne

syntaxe:

<segment de chaîne> ::= *(1)*
<locus chaîne> (<élément de gauche> : <élément de droite>) *(1.1)*
| *<locus chaîne> (<élément de début>* **UP** *<taille de segment>)* *(1.2)*

<élément de gauche> ::= *(2)*
<expression entière> *(2.1)*

<élément de droite> ::= *(3)*
<expression entière> *(3.1)*

<taille de segment> ::= *(4)*
<expression entière> *(4.1)*

sémantique: un segment de chaîne donne un locus chaîne (éventuellement dynamique) qui est la partie du locus de chaîne spécifié indiqué par l'*élément de gauche* et l'*élément de droite* ou par l'*élément de début* et la *taille de la tranche*. La longueur (éventuellement dynamique) du segment de chaîne est déterminée à partir des expressions spécifiées.

Une *segment de chaîne* dans lequel l'*élément de droite* fournit une valeur inférieure à celle fournie par l'*élément de gauche* ou dans lequel la *taille de tranche* fournit une valeur non positive désigne une chaîne vide.

propriétés statiques: le mode (éventuellement dynamique) attaché à un *segment de chaîne* est un mode chaîne **paramétré** formé comme suit:

&nom (*taille de chaîne*)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) du *locus chaîne* si c'est un mode **chaîne** fixe, sinon synonyme du mode **composante** et dans lequel la *taille de chaîne* est soit:

NUM (*élément de droite*) – *NUM* (*élément de gauche*) + 1

soit

NUM (*taille de segment*)

Toutefois, si on dénote une chaîne vide, la *taille de chaîne* est 0. Le mode attaché à un *segment de chaîne* est statique si la *taille de chaîne* est **littérale**, c'est-à-dire si l'*élément de gauche* et l'*élément de droite* sont **littéraux** ou si la *taille de segment* est **littérale**; sinon, le mode est dynamique.

Si le mode du *locus chaîne* est un mode chaîne **variable**, le *segment de chaîne* n'est pas **référençable**.

conditions statiques: les relations suivantes sont valables:

$0 \leq NUM\ (élément\ de\ gauche) \leq L - 1$

$0 \leq NUM\ (élément\ de\ droite) \leq L - 1$

$0 \leq NUM\ (élément\ de\ début) \leq L - 1$

$NUM\ (élément\ de\ début) + NUM\ (taille\ de\ tranche) \leq L$

où *L* est la **longueur effective** du *locus chaîne*. Si *L* et les *expressions* valeurs toutes *entières* sont connues statiquement, les relations peuvent être vérifiées statiquement.

conditions dynamiques: l'exception *RANGEFAIL* est causée si une partie dynamique de la vérification des relations ci-dessus ne se vérifie pas.

exemples:

18.26 blanks (count : 9) *(1.1)*

18.23 string –>(scanstart **UP** *10)* *(1.2)*

4.2.8 Eléments de matrice

syntaxe:

<élément de matrice> ::= *(1)*
<locus matrice> (*<liste d'expressions>*) *(1.1)*

<liste d'expressions> ::= *(2)*
<expression> { , *<expression>* }* *(2.1)*

syntaxe dérivée: la notation: (*<liste d'expressions>*) est une syntaxe dérivée pour:

(*<expression>*) { (*<expression>*) }*

avec autant d'expressions entre parenthèses qu'il y a d'expressions dans la *liste d'expressions*. Ainsi, un *élément de matrice* en syntaxe stricte n'a qu'une seule expression (d'indice).

sémantique: un élément de matrice donne un (sous-)locus qui est un élément du locus matrice spécifié indiqué par *expression*.

propriétés statiques: le mode attaché à l'*élément de matrice* est le mode **élément** du mode du *locus matrice*.

Un *élément de matrice* est **référençable** si **l'implantation d'élément** du mode du *locus matrice* est **NOPACK**.

conditions statiques: la classe de l'*expression* doit être **compatible** avec le mode **indice** du mode du *locus matrice*.

conditions dynamiques: l'exception *RANGEFAIL* est causée si la relation suivante ne se vérifie pas:

$L \leq expression \leq U$

où *L* et *U* sont respectivement la **borne inférieure** et la **borne supérieure** (éventuellement dynamique) du mode du *locus matrice*.

exemple:

11.36 b(m.lin_1)(m.col_1) *(1.1)*

4.2.9 Bandes matricielles

syntaxe:

<bande matricielle> ::= *(1)*
<locus matricielle> (*<élément inférieur>* : *<élément supérieur>*) *(1.1)*
| *<locus matricielle>* (*<premier élément>* **UP** *<taille de segment>*) *(1.2)*

<élément inférieur> ::= *(2)*
<expression> *(2.1)*

<élément supérieur> ::= *(3)*
<expression> *(3.1)*

<premier élément> ::= *(4)*
<expression> *(4.1)*

sémantique: une bande matricielle donne un locus matricielle (éventuellement dynamique) qui est la partie du locus matricielle spécifié indiqué par *l'élément inférieur* et *l'élément supérieur* ou par le *premier élément* et la *taille de segment*. La **borne inférieure** de la bande matricielle est égale à la borne inférieure de la matrice spécifiée; la **borne supérieure** (éventuellement dynamique) est déterminée à partir des expressions spécifiées.

propriétés statiques: le mode (éventuellement dynamique) attaché à une *bande matricielle* est un mode matrice **paramétré** formé comme suit:

&nom (*indice supérieur*)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) du *locus matrice* et *l'indice supérieur* est soit une expression dont la classe est **compatible** avec les classes de *l'élément inférieur* et de *l'élément supérieur* et donne une valeur telle que:

NUM (*indice supérieur*) = *NUM* (*L*) + *NUM* (*élément supérieur*) – *NUM* (*élément inférieur*)

soit une expression dont la classe est **compatible** avec la classe du *premier élément* et donne une valeur telle que:

NUM (*indice supérieur*) = *NUM* (*L*) + *NUM* (*taille de segment*) – 1

où *L* est la **borne inférieure** du mode du *locus matricielle*.

Le mode attaché à une *bande matricielle* est statique si *l'indice supérieur* est **littéral**, c'est-à-dire que l'*élément inférieur* et l'*élément supérieur* sont tous deux **littéraux**, ou si la *taille de segment* est **littérale**; sinon, le mode est dynamique.

Une *bande matricielle* est **référençable** si **l'implantation d'élément** du mode du *locus matricielle* est **NOPACK**.

conditions statiques: les classes de *l'élément inférieur* et de *l'élément supérieur* ou la classe du *premier élément* doivent être **compatibles** avec le mode **indice** du *locus matricielle*.

Les relations suivantes doivent être vérifiées:

$L \leq NUM$ (*élément inférieur*) $\leq$ (*élément supérieur*) $\leq U$

$1 \leq NUM$ (*taille de segment*) $\leq NUM$ (U) $- NUM$ (L) $+ 1$

NUM (L) $\leq NUM$ (*premier élément*) $\leq NUM$ (*premier élément*) $+ NUM$ (*taille de segment*) $- 1 \leq NUM$ (U)

où *L* et *U* sont respectivement la **borne inférieure** et la **borne supérieure** du mode du *locus matrice*. Si *U* et la valeur de toutes les *expressions* sont statiquement connues, les relations peuvent être vérifiées statiquement.

conditions dynamiques: l'exception *RANGEFAIL* est causée si une partie dynamique de la vérification des relations ci-dessus ne se vérifie pas.

exemple:

17.27 res (0 : count – 1) *(1.1)*

4.2.10 Champs de structure

syntaxe:

<champ de structure> ::= *(1)*
<locus structure> . <nom de champ> *(1.1)*

sémantique: un champ de structure donne un (sous-)locus qui est un champ du locus structure spécifié indiqué par le *nom de champ*. Si le *locus structure* a un mode **variable sans étiquettes**, et que le *nom de champ* est un nom de **champ récurrent**, la sémantique est définie par l'implémentation.

propriétés statiques: le mode du *champ de structure* est le mode du *nom de champ*.

Un *champ de structure* est **référençable** si **l'implantation de champ** du *nom de champ* est **NOPACK**.

conditions statiques: le *nom de champ* doit appartenir à l'ensemble des noms **de champ** du mode *locus structure*.

conditions dynamiques: un *locus* ne doit pas dénoter:

- un locus de mode structure **variable avec étiquettes** et la ou les valeurs du ou des champs **étiquettes** associés indiquent que le champ n'existe pas;
- un locus de mode structure **paramétré** dynamique et que la liste de valeurs associée indique que le champ n'existe pas.

Les conditions ci-dessus s'appellent les conditions d'accès au champ récurrent pour le locus. L'exception *TAGFAIL* se produit si elles ne sont pas satisfaites pour le *locus structure*.

exemple:

10.57 last –>.info *(1.1)*

4.2.11 Appels de procédure rendant locus

syntaxe:

<appel de procédure rendant locus> ::= *(1)*
<appel de procédure rendant locus> *(1.1)*

sémantique: une procédure rendant locus fournit le locus renvoyé par la procédure.

propriétés statiques: le mode attaché à un *appel de procédure rendant locus* est le mode **spec de résultat** de *l'appel de procédure rendant locus* si **DYNAMIC** n'y est pas spécifié; sinon, il s'agit d'une version paramétrée dynamiquement qui a les mêmes paramètres que le mode du locus rendu.

L'*appel de procédure rendant locus* est **référençable** si **NONREF** n'est pas spécifié dans la **spec de résultat** de *l'appel de procédure rendant locus*.

conditions dynamiques: l'*appel de procédure rendant locus* ne doit pas donner un locus **indéfini** et la durée de vie du locus donné ne doit pas être terminée.

4.2.12 Appels de routine prédéfinie rendant locus

syntaxe:

<appel de routine prédéfinie rendant locus> ::= *(1)*
<appel de routine prédéfinie rendant locus> *(1.1)*

sémantique: l'appel de routine prédéfinie rendant locus fournit le locus renvoyé par l'appel de routine prédéfinie.

propriétés statiques: le mode qui s'attache à un *appel de routine prédéfinie rendant locus* est le mode spec de **résultat** de *l'appel de routine prédéfinie rendant locus*.

conditions dynamiques: l'*appel de routine prédéfinie rendant locus* ne doit pas donner un locus **indéfini** et la durée de vie du locus donné ne doit pas être terminée.

4.2.13 Conversions de locus

syntaxe:

<conversion de locus> ::= *(1)*
<nom de mode> # (<locus de mode statique>) *(1.1)*

sémantique: une conversion de locus fournit le locus dénoté par le *locus de mode statique*. Une conversion de locus prend le pas sur les règles de vérification et de compatibilité des modes du CHILL. Elle attache explicitement un mode au locus de mode statique spécifié.

La sémantique dynamique précise d'une conversion de locus est définie par l'implémentation.

propriétés statiques: le mode de la *conversion de locus* est le *nom* de *mode*.

Une conversion de locus est **référençable**.

conditions statiques: le *locus de mode statique* doit être **référençable**.

La relation suivante doit se vérifier:

SIZE (*nom de mode*) = *SIZE* (*locus de mode statique*)

4.2.14 Locus moreta prédéfini

syntaxe:

<locus moreta prédéfini> ::= *(1)*
SELF *(1.1)*

sémantique: dans une procédure de composante et/ou un processus **P** d'un *mode moreta*, **SELF** dénote le locus ML de moreta auquel **P** est appliqué à ce moment. Le mode de **SELF** est le mode de ML.

conditions statiques: l'utilisation de **SELF** n'est permise qu'à l'intérieur de la définition d'un mode moreta.

5 Valeurs et leurs opérations

5.1 Définitions de synonymes

syntaxe:

<énoncé de définition de synonyme> ::= *(1)*
SYN *<définition de synonyme>* { , *<définition de synonyme>* }* ; *(1.1)*

<définition de synonyme> ::= *(2)*
<liste d'occurences de définitions> [*<mode>*] = *<valeur constante>* *(2.1)*

syntaxe dérivée: une *définition de synonyme*, où la *liste d'occurences de définitions* comporte plus d'une occurence de définition, est dérivée de plusieurs occurrences de *définition de synonyme*, une pour chaque occurence de définition, avec la même *valeur constante* et, s'il est présent, le même *mode*. Par exemple: **SYN** *i, j = 3;* est dérivé de: **SYN** *i = 3, j = 3;*.

sémantique: une définition de synonyme définit un nom dénotant la valeur **constante** spécifiée.

propriétés statiques: une *occurence de définition* définie dans une *définition de synonyme* est un nom **de synonyme**.

La classe du nom **de synonyme** est, si un *mode* est spécifié, la M-classe par valeur, où M est le *mode*, sinon la classe de la *valeur constante*.

Un nom **de synonyme** est **indéfini** si et seulement si la *valeur constante* est une valeur **non définie** (voir 5.3.1).

Un nom **de synonyme** est **littéral** si et seulement si la *valeur constante* est **littérale**.

conditions statiques: si un *mode* est spécifié, il doit être **compatible** avec la classe de la *valeur constante* et la valeur donnée par la *valeur constante* doit être une des valeurs définies par le *mode*.

L'évaluation de la *valeur constante* ne doit pas dépendre directement ou indirectement de la valeur **constante** du nom **synonyme**.

exemples:

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 Valeur primitive

5.2.1 Généralités

syntaxe:

<valeur primitive> ::= *(1)*
<contenu de locus> *(1.1)*
| *<nom de valeur>* *(1.2)*
| *<littéral>* *(1.3)*
| *<multiplet>* *(1.4)*
| *<valeur élément de chaîne>* *(1.5)*
| *<valeur segment de chaîne>* *(1.6)*
| *<valeur élément de matrice>* *(1.7)*
| *<valeur bande matricielle>* *(1.8)*
| *<valeur champ de structure>* *(1.9)*
| *<conversion d'expression>* *(1.10)*
| *<conversion de représentation>* *(1.11)*
| *<appel de procédure rendant valeur>* *(1.12)*
| *<appel de routine prédéfinie rendant valeur>* *(1.13)*
| *<expression démarrer>* *(1.14)*
| *<opérateur nullaire>* *(1.15)*
| *<expression parenthésée>* *(1.16)*

sémantique: une valeur primitive est le constituant de base d'une expression. Certaines valeurs primitives ont une classe dynamique, c'est-à-dire une classe basée sur un mode dynamique. Pour ces valeurs primitives, les vérifications de compatibilité ne peuvent avoir lieu qu'à l'exécution. Une détection d'anomalie entraînera l'exception *TAGFAIL* ou *RANGEFAIL*.

propriétés statiques: la classe de la *valeur primitive* est respectivement la classe du *contenu de locus*, *nom de valeur*, etc.

Une *valeur primitive* est **constante** si et seulement si c'est un *nom de valeur* **constant**, un *littéral*, un *multiplet* **constant**, une *conversion d'expression* **constante**, une *conversion de représentation* **constante**, un *appel de routine prédéfinie rendant valeur* **constant** ou une *expression parenthésée* **constante**.

Une *valeur primitive* est **littérale**, si et seulement si c'est un *nom de valeur* **littéral**, un *littéral* **discret** ou un *appel de routine prédéfinie rendant valeur* **littéral**.

5.2.2 Contenu de locus

syntaxe:

<contenu de locus> ::= *(1)*
<locus> *(1.1)*

sémantique: un contenu de locus donne la valeur contenue dans le locus spécifié. On accède au locus pour obtenir la valeur stockée.

propriétés statiques: la classe du *contenu de locus* est la M-classe par valeur, où M est le mode (éventuellement dynamique) du *locus*.

conditions statiques: le mode du *locus* ne doit pas avoir la **propriété de non-valeur**.

conditions dynamiques: la valeur donnée ne doit pas être **non définie**.

exemple:

3.7 c2.im *(1.1)*

5.2.3 Noms de valeur

syntaxe:

<nom de valeur> ::= *(1)*
<nom de synonyme> *(1.1)*
| *<nom d'énumération de valeur>* *(1.2)*
| *<nom de valeur faire-avec>* *(1.3)*
| *<nom de valeur reçue>* *(1.4)*
| *<nom de procédure générale>* *(1.5)*

sémantique: un nom de valeur donne une valeur. Un nom de valeur entre dans une des catégories suivante:

- un nom **de synonyme**, c'est-à-dire un nom défini dans un *énoncé de définition de synonyme*;
- un nom **d'énumération de valeur**, c'est-à-dire un nom défini par un *compteur de boucle* dans une *énumération de valeur*;
- un nom **de valeur faire-avec**, c'est-à-dire un nom de **champ** introduit comme nom de valeur dans *l'action faire* avec une *partie avec*;
- un nom **de valeur reçue**, c'est-à-dire un nom introduit dans une *action recevoir et avec cas*;
- un nom **de procédure générale** (voir 10.4).

Lorsque la valeur dénotée par un *nom de valeur faire-avec* est un champ récurrent d'une valeur de structure variable sans marqueurs, la sémantique est définie par l'implémentation.

propriétés statiques: la classe d'un *nom de valeur* est respectivement la classe du *nom de synonyme*, du *nom d'énumération de valeur*, du *nom de valeur faire-avec*, du *nom de valeur reçue*, ou la classe dérivée de M, où M est le mode du *nom de procédure générale*.

Un *nom de valeur* est **littéral** si et seulement si c'est un *nom de synonyme* **littéral**.

Un *nom de valeur* est **constant** si c'est un *nom de synonyme* ou un *nom de procédure générale* indiquant un nom **de procédure** qui s'est attaché à une *définition de procédure* qui n'est pas englobée par un bloc.

conditions statiques: le *nom de synonyme* ne doit pas être **indéfini**.

conditions dynamiques: evaluer un *nom de valeur faire-avec* provoque l'exception *TAGFAIL* si la valeur dénotée est un champ **récurrent** et si les conditions d'accès au champ récurrent pour la valeur ne sont pas satisfaites.

exemples:

10.12 max *(1.1)*

8.8 i *(1.2)*

15.54 this_counter *(1.4)*

5.2.4 Littéraux

5.2.4.1 Généralités

syntaxe:

<littéral> ::= *(1)*
<littéral entier> *(1.1)*
| *<littéral virgule flottante>* *(1.2)*
| *<littéral booléen>* *(1.3)*
| *<littéral caractère>* *(1.4)*
| *<littéral ensemble>* *(1.5)*
| *<littéral vide>* *(1.6)*
| *<littéral chaîne de caractères>* *(1.7)*
| *<littéral chaîne binaire>* *(1.8)*

sémantique: un littéral donne une valeur **constante**.

propriétés statiques: la classe du *littéral* est respectivement la classe du *littéral entier, littéral booléen*, etc. Un *littéral* est **discret** si c'est un *littéral entier*, un *littéral booléen*, un *littéral caractère* ou un *littéral ensemble*.

La lettre suivie d'une apostrophe qui figure au début d'un *littéral entier*, d'un *littéral booléen*, d'un *littéral chaîne binaire, d'un littéral grands caractères et d'un littéral chaîne de grands caractères* (c'est-à-dire *B'*, *D'*, *H'*, *O'*, *W'*,*b'*, *d'*, *h'*, *o*, *w'*) est une qualification de littéral.

5.2.4.2 Littéraux entier

syntaxe:

<littéral entier> ::= *(1)*
<littéral non signé entier> *(1.1)*
| *<littéral signé entier>* *(1.2)*

<littéral non signé entier> ::= *(2)*
<littéral entier décimal> *(2.1)*
| *<littéral entier binaire>* *(2.2)*
| *<littéral entier octal>* *(2.3)*
| *<littéral entier hexadecimal>* *(2.4)*

<littéral signé entier> ::= *(3)*
– *<littéral non signé entier>* *(3.1)*

<littéral entier décimal> ::= *(4)*
[{ D | d } '] *<séquence de chiffres>* *(4.1)*

<littéral entier binaire> ::= *(5)*
{ B | b } ' { 0 | l | _ }$^{+}$ *(5.1)*

<littéral entier octal> *(6)*
{ O | o } ' { *<chiffre octal>* | _ }$^{+}$ *(6.1)*

<littéral entier hexadécimal> ::= *(7)*
{ H | h } ' { *<chiffre hexadécimal>* | _ }$^{+}$ *(7.1)*

<chiffre hexadécimal> ::= *(8)*
<chiffre> | A | B | C | D | E | F | a | b | c | d | e | f *(8.1)*

<chiffre octal> ::= *(9)*
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 *(9.1)*

<séquence de chiffres> ::= *(10)*
{ *<chiffre>* | _ }$^{+}$ *(10.1)*

sémantique: un littéral entier donne une valeur entière non négative. La notation décimale usuelle (base 10) est offerte, de même que les notations binaire (base 2), octale (base 8) et hexadécimale (base 16). Le caractère souligné (_) n'est pas significatif, c'est-à-dire qu'il ne sert qu'à améliorer la lisibilité et qu'il n'a pas d'influence sur la valeur dénotée.

Un littéral signé donne une valeur qui est l'opposé de celle donnée par le *littéral entier sous le signe* qu'il contient.

propriétés statiques: la classe d'un *littéral entier* est la *&INT*-classe par dérivation. Un *littéral entier* est **constant** et **littéral**.

conditions statiques: ni la chaîne qui suit l'apostrophe (') ni la *séquence de chiffres* ne doit consister seulement en caractères soulignés.

La valeur donnée par *littéral entier* doit être une des valeurs définies par le mode *&INT*.

exemple:

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 Littéraux à virgule flottante

syntaxe:

<littéral à virgule flottante> ::= *(1)*
<littéral non signé à virgule flottante> *(1.1)*
| *<littéral signé à virgule flottante>* *(1.2)*

<littéral non signé à virgule flottante> ::= *(2)*
<séquence de chiffres> . [*<séquence de chiffres>*] [*<exposant>*] *(2.1)*
[*<séquence de chiffres>*] . *<séquence de chiffres>* [*<exposant>*] *(2.2)*

<littéral signé à virgule flottante> ::= *(3)*
– *<littéral non signe à virgule flottante>* *(3.1)*

<exposant> ::= *(4)*
E *<séquence de chiffres>* *(4.1)*
E – *<séquence de chiffres>* *(4.2)*

syntaxe dérivée: un *littéral à virgule flottante* dans lequel manque 1. une *séquence de chiffres*, 2. un *exposant* est une syntaxe dérivée pour un *littéral* dans lequel 1. la *séquence de chiffres* est 0, 2. l'*exposant* est E1.

sémantique: un littéral à virgule flottante donne une valeur à virgule flottante exprimée sous forme de nombre décimal en notation scientifique.

Un littéral signé à virgule flottante donne une valeur qui est l'opposé de celle donnée par le *littéral non signé à virgule flottante* qu'il contient.

Si le littéral à virgule flottante se situe entre la **borne supérieure** et la **borne inférieure** de l'un des modes virgule flottante **prédéfinis** de l'implémentation mais qu'il ne peut être représenté avec exactitude, sa valeur est forcée à la valeur donnée par une *conversion de représentation* implicite au mode virgule flottante **prédéfini** choisi par l'implémentation pour représenter le *littéral à virgule flottante*.

propriétés statiques: la classe d'un *littéral à virgule flottante* est la *&FLOAT*-classe par dérivation. Un *littéral à virgule flottante* est **constant** et **littéral**.

La **précision** d'un *littéral à virgule flottante* est la somme du nombre de décimales donné par les deux *séquences de chiffres* formant sa mantisse.

conditions statiques: la valeur donnée par un *littéral à virgule flottante* doit être une des valeurs définies par le mode *&FLOAT*.

exemple:

10,0E1 *(1.1)*

–365,0E–5 *(1.1)*

5.2.4.4 Littéraux booléen

syntaxe:

<littéral booléen> ::= *(1)*
<nom de littéral booléen> *(1.1)*

noms prédéfinis: les noms *FALSE* et *TRUE* sont prédéfinis comme noms de **littéral booléen**.

sémantique: un littéral booléen donne une valeur booléenne.

propriétés statiques: la classe du *littéral booléen* est la *BOOL*-classe par dérivation. Un *littéral booléen* est **constant** et **littéral**.

exemple:

5.42 *FALSE* *(1.1)*

5.2.4.5 Littéraux caractère

syntaxe:

<littéral caractère> ::= *(1)*
<littéral caractère étroit> *(1.1)*
| *<littéral caractère large>* *(1.2)*

<littéral caractère étroit> *(2)*
' { *<caractère>* | *<séquence de contrôle>*} ' *(2.1)*

<littéral caractère large> *(3)*
{ W | w }' { *<caractère>* | *<séquence de contrôle>*} ' *(3.1)*

<séquence de contrôle>} *::=* *(4)*
^ (*<expression de littéral entier>* { , *<expression de littéral entier>* }*) *(4.1)*
| ^ *<caractère non spécial>* *(4.2)*
| ^^ *(4.3)*

sémantique: un littéral caractère fournit une valeur de caractère.

Indépendamment de la représentation imprimable, on peut utiliser la représentation *séquence de contrôle*. Une *séquence de contrôle* dans laquelle l'accent circonflexe (^) est suivi d'une ouverture de parenthèses dénote la séquence de caractères dont la représentation est *l'expression de littéral entier* qu'elle contient; si elle est suivie d'un autre accent circonflexe elle se dénote elle-même, et sinon elle dénote le caractère dont la représentation est obtenu par l'inversion logique de b7 de la représentation interne du *caractère non spécial* qu'elle contient (voir 12.4.4 et l'Appendice I).

propriétés statiques: la classe d'un *littéral caractère étroit* est la *CHAR*-classe par dérivation. La classe d'un *littéral caractère large* est **constant** et **littéral**.

conditions statiques: une *séquence de contrôle* d'un *littéral caractère* doit dénoter un seul caractère.

La valeur donnée par une *expression de littéral entier* dans une *séquence de contrôle* doit appartenir à la gamme de valeurs définies par la représentation des caractères du jeu de caractères CHILL (voir l'Appendice I) en cas de *littéral caractère étroit* ou à l'ensemble de valeurs définies par la représentation de caractères dans le jeu de caractères ISO/CEI 10 646-1 en cas de *littéral caractère large*.

exemple:

7.9 'M' *(2.1)*

5.2.4.6 Littéraux ensemble

syntaxe:

<littéral ensemble> ::= *(1)*
[*<nom de mode>* .] *<nom d'élément d'ensemble>* *(1.1)*

sémantique: un littéral ensemble donne une valeur d'ensemble. Un littéral ensemble est un nom défini dans un mode ensemble.

propriétés statiques: la classe d'un *littéral ensemble* est la M-classe par dérivation, où M est le *nom de mode*, s'il est spécifié. Sinon, M dépend du contexte dans lequel survient le *littéral ensemble* conformément a ce qui suit:

- si le *littéral ensemble* est utilisé à un endroit ou l'on peut utiliser un *multiplet* sans le *nom de mode*, M est déterminé suivant les mêmes règles que celles définies pour le multiplet (voir 5.2.5);
- si le *littéral ensemble* est utilisé comme valeur dans un *multiplet*, M est le mode de cette valeur;
- si le *littéral ensemble* est utilisé dans un *intervalle littéral* pour définir un *mode intervalle discret* de la forme:

  <nom de mode discret> (*<intervalle littéral>*)

  M est le *nom de mode discret*;
- si le *littéral ensemble* est l'*expression usage*, l'*expression positionnement*, l'*expression indice* ou l'*expression écrire* dans une routine prédéfinie pour entrée-sortie (voir 7.4), M est respectivement *USAGE*, *WHERE*, le mode **indice** du *locus accès* ou du *locus texte*, le mode **enregistrement** du *locus accès*;
- si le *littéral ensemble* est utilisé dans une *expression conditionnelle*, M est déterminé de la même manière que pour l'*expression* qui le contient;

- si le *littéral ensemble* est l'*indice supérieur* dans un *mode matrice paramétré*, M est le *mode indice* correspondant du *mode matrice d'origine*;
- si le *littéral ensemble* est une *expression* dans un *mode structure paramétré*, M est le mode **racine** du *nom de champ étiquette* correspondant dans le mode structure variable originel;
- si le *littéral ensemble* est utilisé dans un *élément de matrice* ou dans une *bande matricielle*, M est le *mode indice* correspondant dans le *mode matrice*;
- si le *littéral ensemble* est utilisé dans une *étiquette de cas*, M est déterminé à partir du mode du *nom de champ étiquette* (pour le *mode structure*), à partir du sélecteur correspondant dans la *liste de sélecteurs de cas* (pour *action à ces* ou *expression conditionnelle*), ou à partir du *mode indice* (pour *multiplet*);
- si le *littéral ensemble* est utilisé comme *borne inférieure* ou comme *borne supérieure* et qu'un *nom de mode discret* est spécifié dans l'*intervalle littéral* qui le contient, M est un *nom de mode discret*.

Un *littéral ensemble* est **constant** et **littéral**.

conditions statiques: le *nom de mode* facultatif ne peut être omis que dans les contextes spécifiés ci-dessus.

Le *nom d'élément d'ensemble* doit appartenir à l'ensemble de noms d'**élément d'ensemble** de M.

exemples:

6.51 dec *(1.1)*

11.78 king *(1.1)*

5.2.4.7 Littéral vide

syntaxe:

<littéral vide> ::= *(1)*
<nom de littéral vide> *(1.1)*

noms prédéfinis: le nom NULL est prédéfini comme nom **de littéral vide**.

sémantique: le littéral vide donne soit la valeur référence vide, c'est-à-dire une valeur qui ne référence aucun locus, soit la valeur procédure vide, c'est-à-dire une valeur qui n'indique aucune procédure, soit la valeur exemplaire vide, c'est-à-dire une valeur qui n'identifie aucun processus.

propriétés statiques: la classe du *littéral vide* est la classe **nulle**. Un *littéral vide* est **constant**.

exemple:

10.43 NULL *(1.1)*

5.2.4.8 Littéraux chaîne de caractères

syntaxe:

<littéral chaîne de caractères> ::= *(1)*
<littéral chaîne de caractères étroits> *(1.1)*
| *<littéral chaîne de caractères larges>* *(1.2)*

<littéral chaîne de caractères étroits> ::= *(2)*
"{ *<caractère non réservé>* | *<citation>* | *<séquence de contrôle>*}* " *(2.1)*

<littéral chaîne de caractères larges> ::= *(3)*
{ W' | w' } " { *<caractère large non réservé>* |
<citation> | *<séquence de contrôle>*}* " *(3.1)*

<citation> ::= *(4)*
"" *(4.1)*

sémantique: un littéral chaîne de caractères donne une valeur chaîne de caractères qui peut être de longueur 0. C'est une liste de valeurs pour les éléments de la chaîne; les valeurs sont données pour les éléments par ordre d'indice croissant de gauche à droite. Pour représenter le caractère citation (") dans un littéral chaîne de caractères, il faut l'écrire deux fois ("").

propriétés statiques: la **longueur de chaîne** d'un *littéral chaîne de caractères* est le nombre de *caractère non réservé*, *de citation* et de caractères dénotés par des occurrences de *séquence de contrôle*.

La classe d'un *littéral chaîne de caractères* est la **CHARS** *(n)*-classe par dérivation, où *n* est la **longueur de chaîne** du *littéral chaîne de caractères*. La classe d'un *littéral chaîne de caractères* est la **WCHARS** *(n)*-classe par dérivation, où *n* est la **longueur de chaîne** du *littéral chaîne de caractères larges*. Un *littéral chaîne de caractères* est **constant**.

exemple:

8.20 "A-B<ZAA9K' " *(2.1)*

5.2.4.9 Littéraux chaîne binaire

syntaxe:

<littéral chaîne binaire> ::= *(1)*
<littéral de chaîne binaire simple> *(1.1)*
| *<littéral de chaîne binaire octale>* *(1.2)*
| *<littéral de chaîne binaire hexadécimale>* *(1.3)*

<littéral de chaîne binaire simple> ::= *(2)*
{ B | b } ' { 0 | 1 | _ }* ' *(2.1)*

<littéral de chaîne binaire octale> ::= *(3)*
{ O | o } ' { *<chiffre octal>* | _ }* ' *(3.1)*

<littéral de chaîne binaire hexadécimale> ::= *(4)*
{ H | h } ' { *<chiffre hexadécimal>* | _ }* ' *(4.1)*

sémantique: un littéral chaîne binaire donne une valeur chaîne binaire qui peut être de longueur 0. Les notations binaire, octale ou hexadécimale peuvent être employées. Le caractère souligné (_) n'est pas significatif, c'est-à-dire qu'il ne sert qu'à améliorer la lisibilité et n'influence pas la valeur indiquée.

Un littéral de chaîne binaire est une liste de valeurs pour les éléments de la chaîne; les valeurs sont données pour les éléments par ordre d'indice croissant de gauche à droite.

propriétés statiques: la **longueur de chaîne** d'un *littéral de chaîne binaire* est soit le nombre d'occurrences de *0* et de *1* dans un *littéral chaîne binaire simple*, soit trois fois le nombre d'occurrences de *chiffre octal* dans un *littéral chaîne binaire octal*, soit quatre fois le nombre d'occurrences de *chiffre hexadécimal* dans un *littéral chaîne binaire hexadécimale*.

La classe d'un *littéral chaîne binaire* est la **BOOLS** *(n)*-classe par dérivation, où *n* est la **longueur de chaîne** du *littéral chaîne binaire*. Un *littéral chaîne binaire* est **constant**.

exemples:

B'101011_110100' *(1.1)*

O'53_64' *(1.2)*

H'AF4' *(1.3)*

5.2.5 Multiplets

syntaxe:

<multiplet> ::= *(1)*
[*<nom de mode>*] (: { *<multiplet ensembliste>* |
<multiplet de matriciel> | *<multiplet de structure>* } *:)* *(1.1)*

<multiplet ensembliste> ::= *(2)*
[{ *<expression>* | *<intervalle>* } { , { *<expression>* | *<intervalle>* } }*] *(2.1)*

<intervalle> ::= *(3)*
<expression> : *<expression>* *(3.1)*

<multiplet matriciel> ::= *(4)*
<multiplet de matrice sans indices> *(4.1)*
| *<multiplet de matrice avec indices>* *(4.2)*

<multiplet de matrice sans indices> *(5)*
<valeur> { , *<valeur>* }* *(5.1)*

<multiplet de matrice avec indices> ::= *(6)*
*<liste d'étiquettes de cas> : <valeur> { , <liste d'étiquettes de cas> : <valeur> }** *(6.1)*

<multiplet de structure> ::= *(7)*
<multiplet de structure sans noms de champ> *(7.1)*
| *<multiplet de structure avec noms de champ>* *(7.2)*

<multiplet de structure sans noms de champ> ::= *(8)*
*<valeur> { , <valeur> }** *(8.1)*

<multiplet de structure avec noms de champ> ::= *(9)*
*<liste de noms de champ> : <valeur> { , <liste de noms de champ> : <valeur> }** *(9.1)*

<liste de noms de champ> ::= *(10)*
*. <nom de champ> { , . <nom de champ> }** *(10.1)*

syntaxe dérivée: les crochets ouvrant et fermant, [et], d'un multiplet sont une syntaxe dérivée pour respectivement (: et :). Ceci n'est pas indiqué dans la syntaxe pour éviter toute confusion avec les crochets utilisés comme métasymboles.

sémantique: un multiplet donne une valeur ensembliste, une valeur matrice ou une valeur structure.

Si c'est une valeur ensembliste, il consiste en une liste d'expressions et/ou d'intervalles, dénotant ces valeurs primitives qui appartiennent à la valeur ensembliste. Un intervalle dénote ces valeurs qui sont comprises entre les valeurs données par les expressions de l'intervalle ou sont ces valeurs elles-mêmes. Si la deuxième expression donne une valeur inférieure à la valeur donnée par la première expression, l'intervalle est vide, c'est-à-dire qu'il ne dénote aucune valeur. Le multiplet ensembliste peut dénoter la valeur ensembliste vide.

Si c'est une valeur matrice, il consiste en une liste de valeurs (éventuellement indicées) pour les éléments de matrice; dans un multiplet matriciel sans indices, les valeurs sont données pour les éléments dans l'ordre croissant de leurs indices; dans un multiplet matriciel avec indices, les valeurs sont données pour les éléments dont les indices sont spécifiés dans la liste d'étiquettes de cas qui précède la valeur. Cela peut être employé comme abréviation pour les multiplets de longues matrices dont beaucoup de valeurs sont les mêmes. L'étiquette **ELSE** dénote toutes les valeurs d'indice non mentionnées explicitement, l'étiquette * dénote toutes les valeurs d'indice (pour de plus amples détails, voir 12.3).

Si c'est une valeur structure, il consiste en un ensemble de valeurs (éventuellement nommées) pour les champs de la structure. Dans un multiplet de structure sans noms de champ, les valeurs sont données pour les champs dans l'ordre où ceux-ci sont spécifiés dans le mode structure attaché. Dans un multiplet de structure avec noms de champ, les valeurs sont données pour les champs dont les noms de champ sont spécifiés dans la liste de noms de champ pour la valeur.

L'ordre d'évaluation des expressions et des valeurs dans un multiplet est indéfini et elles peuvent être vues comme étant évaluées dans un ordre mélangé.

propriétés statiques: la classe d'un *multiplet* est la M-classe par valeur où M est le *nom de mode*, s'il y en a un, sinon M dépend du contexte où le *multiplet* se trouve, selon la liste suivante:

- si le *multiplet* est la *valeur* ou *valeur constante* d'une *initialisation* dans une *déclaration de locus*, alors M est le mode dans la déclaration de locus;
- si le *multiplet* est la *valeur* partie droite d'une *action d'affectation simple*, alors M est le mode (éventuellement dynamique) du *locus* partie gauche;
- si le *multiplet* est la *valeur constante* d'une *définition de synonyme* avec un *mode* spécifié, alors M est ce *mode*;
- si le *multiplet* est utilisé dans un *opérande-2* et que l'un des opérandes est **fort**, M est le mode de l'opérande **fort**;
- si le *multiplet* est un *paramètre effectif* dans un *appel de procédure* ou dans une *expression de début*, où **DYNAMIC** n'est pas spécifié dans la *spec de paramètre* correspondante, alors M est le mode dans la *spec de paramètre* correspondante;
- si le *multiplet* est la *valeur* dans une *action revenir* ou une *action résulter*, alors M est le mode de **la spec de résultat** de nom de **procédure** de l'*action résulter* ou de l'*action revenir* (voir 6.8);
- si le *multiplet* est une *valeur* dans une *action envoyer*, alors c'est le mode spécifié dans la définition de signal du *nom de signal* ou le mode **des éléments de tampon** du mode du *locus tampon*;
- si le *multiplet* est une *expression* dans un *multiplet matriciel*, alors M est le mode **des éléments** du mode du *multiplet matriciel*;

- si le *multiplet* est une *expression* dans un *multiplet de structure sans noms de champ* ou un *multiplet de structure avec noms de champ* où la liste de *noms de champ* associée ne consiste qu'en un *nom de champ*, alors M est le mode de champ du *multiplet de structure* pour lequel le multiplet est spécifié;
- si le *multiplet* est la *valeur* dans un appel de routine prédéfinie *GETSTACK* ou *ALLOCATE*, alors M est le mode dénoté par *argument de mode*.

Un *multiplet* est **constant** si et seulement si chaque *valeur* ou *expression* qu'il contient est **constante**.

conditions statiques: le *nom de mode* optionnel ne peut être omis que dans les contextes spécifiés ci-dessus. Selon qu'un *multiplet ensembliste*, *multiplet matriciel* ou *multiplet de structure* est spécifié, les règles de compatibilité suivantes doivent être respectées:

a) *multiplet ensembliste*

1) le mode du *multiplet* doit être un mode ensembliste.

2) la classe de chaque *expression* doit être **compatible** avec le mode **primitif** du mode du *multiplet*.

3) pour un multiplet ensembliste **constant**, la valeur donnée par chaque *expression* doit être une des valeurs définies par ce mode **primitif**.

b) *multiplet matriciel*

1) le mode du *multiplet* doit être un mode matrice;

2) la classe de chaque *valeur* doit être **compatible** avec le mode **élément** du mode du *multiplet*;

3) dans le cas d'un *multiplet matriciel sans indices*, il faut autant d'occurrences de *valeur* que le **nombre d'éléments** dans le mode matrice du *multiplet*;

4) dans le cas d'un *multiplet matriciel avec indices*, les conditions de sélection de cas doivent être remplies pour la liste d'occurrences de *liste d'étiquettes de cas* (voir 12.3). La **classe résultante** de la liste doit être **compatible** avec le mode **indice** du mode du *multiplet*. La liste de spécifications d'étiquettes de cas doit être **complète**;

5) dans le cas d'un *multiplet matriciel avec indices*, les valeurs indiquées explicitement par chaque étiquette de cas dans une *liste d'étiquettes de cas* doivent être des valeurs définies par le mode **indice** du *multiplet*;

6) dans un *multiplet matriciel sans indices*, au moins une occurrence de *valeur* doit être une *expression*;

7) pour un *multiplet* matriciel **constant** où le mode **éléments** du mode du *multiplet* est un mode discret, chaque *valeur* spécifiée doit donner une valeur définie par ce mode **éléments**, sauf si c'est une valeur **non définie**;

c) *multiplet de structure*

1) le mode de multiplet doit être un mode structure;

2) ce mode ne doit pas être un mode structure qui a des noms de **champ invisibles** (voir 12.2.5).

Dans le cas d'un multiplet de structure sans noms de champ:

– Si le mode du *multiplet* n'est ni un mode structure **variable** ni un mode structure **paramétré**, alors:

3) il doit y avoir autant d'occurrences de *valeur* qu'il y a de noms de **champ** dans la liste de **noms de champ** du mode du *multiplet*;

4) la classe de chaque *valeur* doit être **compatible** avec le mode du nom de **champ** correspondant (par position) du mode du *multiplet*.

– Si le mode du *multiplet* est un mode structure **variable avec étiquettes** ou un mode structure **paramétré avec étiquettes**, alors:

5) chaque *valeur* spécifiée pour un champ **étiquettes** doit être une *expression littérale discrète*;

6) il doit y avoir autant d'occurrences de *valeur* qu'il y a de noms de **champ** indiqués comme existants par la au les valeurs données par les occurrences d'*expression littérale discrète* spécifiées pour les champs **étiquettes**;

7) la classe de chaque *valeur* doit être **compatible** avec le mode du nom **de champ** correspondant.

– Si le mode du *multiplet* est un mode structure **variable sans étiquettes** ou un mode structure **paramétré sans étiquettes**, alors:

8) il n'est pas permis de spécifier de *multiplet de structure sans noms de champ*.

Dans le cas d'un *multiplet de structure avec noms de champ*:

– Si le mode du *multiplet* n'est ni un mode structure **variable** ni un mode structure **paramétré**, alors:

  9) chaque nom de **champ** de la liste de noms de **champ** du mode du *multiplet* doit être mentionné une seule et unique fois dans le *multiplet*;

  10) la classe de chaque *valeur* doit être **compatible** avec le mode de chaque nom de **champ** spécifié dans la *liste de noms de champ* qui précède cette *valeur*. Les modes de tous les noms de **champ** de la *liste des noms de champ* doivent être **équivalents**.

– Si le mode du *multiplet* est un mode structure **variable avec étiquettes** ou un mode structure **paramétré avec étiquettes**, alors:

  11) chaque *valeur* spécifiée pour un champ **étiquettes** doit être une *expression littérale discrète*;

  12) chaque nom de **champ** qui dénote un champ fixe ou un champ signalé existant par la ou les valeurs données par les occurrences d'*expression littérale discrète* spécifiées pour les champs **étiquettes** doit être spécifié une fois, et une seule, dans le *multiplet*;

  13) la classe de chaque *valeur* doit être **compatible** avec le mode du nom **de champ** spécifié dans la *liste de noms de champ* qui précède cette *valeur*.

– Si le mode du *multiplet* est un mode structure **variable sans étiquettes** ou un mode structure **paramétré sans étiquettes**, alors:

  14) chaque nom de **champ** doit être mentionné au maximum une fois dans le multiplet. Tous les noms de **champ fixe** doivent être mentionnés. Les noms de **champ** mentionnés dans le multiplet, et qui sont définis dans la même alternative variant doivent être tous définis dans le même champ alternatif ou après **ELSE**. Tous les noms de **champ** d'une alternative variant sélectionnés ou définis après **ELSE** doivent être mentionnés;

  15) la classe de chaque *valeur* doit être **compatible** avec le mode de chaque nom de **champ** spécifié dans la *liste de noms de champ* qui précède cette *valeur*;

  16) si le mode du *multiplet* est un mode structure **paramétré avec étiquettes**, la liste de valeurs données par les occurrences d'*expression littérale discrète* spécifiées pour les champs **étiquettes**, doit être la même que la liste de valeurs du mode du *multiplet*;

  17) pour un *multiplet de structure* **constant**, chaque *valeur* spécifiée pour un champ qui a un mode discret doit donner une valeur définie par le mode du **champ** (bornes incluses), sauf si c'est une valeur **non définie**;

  18) au moins une occurrence de *valeur* doit être une *expression.*

Aucun *multiplet* ne peut comporter deux occurrences de *valeur* telles que l'une soit **extrarégionale** et l'autre **intrarégionale** (voir 11.2.2).

conditions dynamiques: les conditions d'affectation de chaque valeur pour ce qui est du mode **primitif**, du mode **éléments**, ou du mode **champ** associé, dans le cas respectivement d'un *multiplet ensembliste*, *multiplet matriciel* ou *multiplet de structure* (voir 6.2) s'appliquent (voir les conditions a2, b2, c4, c7, c10, c13 et c15).

Si le *multiplet* a un mode matrice dynamique, l'exception *RANGEFAIL* est causée si l'une des conditions b3 ou b5 n'est pas respectée.

Si le *multiplet* a un mode structure **paramétré** dynamique, l'exception *TAGFAIL* est causée si l'une des conditions c14 ou c16 n'est pas respectée.

La valeur donnée par un *multiplet* ne doit pas être **non définie**.

exemples:

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 Valeurs élément de chaîne

syntaxe:

<valeur élément de chaîne> ::= *(1)*
<valeur primitive chaîne> (*<élément de début>*) *(1.1)*

NOTE – Si la valeur primitive *chaîne* est un locus *chaîne*, la construction syntaxique est ambiguë et sera interprétée comme un *élément de chaîne* (voir 4.2.6).

sémantique: une valeur élément de chaîne fournit une valeur qui est l'élément de la valeur de chaîne spécifiée indiquée par *l'élément de début*.

propriétés statiques: la classe de la *valeur élément* de chaîne est la M-classe par valeur, où M est le mode **élément** du mode de la *valeur primitive chaîne*.

Une *valeur élément de chaîne* est **constante** si, et seulement si, la *valeur primitive chaîne* et l'*élément de début* sont **constants**.

conditions dynamiques: la valeur fournie par une *valeur élément de chaîne* ne doit pas être **indéfinie**.

L'exception *RANGEFAIL* a lieu si la relation suivante ne se vérifie pas:

$$0 \leq NUM(\text{élément de début}) \leq L - 1$$

où L est la **longueur effective** de la *valeur primitive chaîne*.

5.2.7 Valeurs segment de chaîne

syntaxe:

<valeur segment de chaîne> ::= *(1)*
<valeur primitive chaîne> (*<élément de gauche>* : *<élément de droite>*) *(1.1)*
| *<valeur primitive chaîne>* (*<élément de début>* **UP** *<taille de tranche>*) *(1.2)*

NOTE – Si la *valeur primitive chaîne* est un *locus chaîne*, la construction syntaxique est ambiguë et sera interprétée comme un *segment de chaîne* (voir 4.2.7).

sémantique: une valeur segment de chaîne donne une valeur chaîne (éventuellement dynamique) qui est la partie de la valeur chaîne spécifiée indiquée par *l'élément de gauche* et l'*élément de droite* ou par l'*élément de début* et *la taille de segment*. La longueur (éventuellement dynamique) du segment de chaîne est déterminée à partir des expressions spécifiées.

Un *segment de chaîne* dans lequel l'*élément de droite* donne une valeur inférieure à celle que donne l'*élément de gauche* ou dans lequel la *taille de segment* donne une valeur non positive désigne une chaîne vide.

propriétés statiques: la classe (éventuellement dynamique) d'une *valeur segment de chaîne* est la M-classe par valeur si la *valeur primitive chaîne* est **forte** ou sinon la M-classe par dérivation, où M est un mode chaîne **paramétré** construit comme:

&nom (*taille de chaîne*)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode **racine** (éventuellement dynamique) de *la valeur primitive chaîne* si c'est un mode chaîne **fixe**, sinon synonyme du mode **composante**, et où *taille de chaîne* est soit

$$NUM(\text{élément de droite}) - NUM(\text{élément de gauche}) + 1$$

soit

$$NUM(\text{taille de segment}).$$

Toutefois, si l'on dénote une chaîne vide, la *taille de segment* est 0. La classe d'une *valeur segment de chaîne* est une classe statique si la *taille de segment* est **littérale**: c'est-à-dire que l'*élément de gauche* et l'*élément de droite* sont **littéraux** ou que la *taille de segment* est **littérale**; sinon, la classe est une classe dynamique.

Une valeur *segment de chaîne* est **constante** si et seulement si la *valeur primitive chaîne* et la *taille de tranche* sont **constantes**.

conditions statiques: les relations suivantes doivent être vérifiées:

$0 \leq NUM$ (*élément de gauche*) $\leq L - 1$

$0 \leq NUM$ (*élément de droite*) $\leq L - 1$

$0 \leq NUM$ (*élément de début*) $\leq L - 1$

NUM (*élément de début*) + NUM (*taille de segment*) $\leq L$

où *L* est la **longueur effective** de la *valeur primitive chaîne*. Si *L* et les *expressions* valeurs toutes *entières* sont connues statistiquement, les relations peuvent être vérifiées statistiquement.

conditions dynamiques: la valeur donnée par une *valeur segment de chaîne* ne doit pas être une valeur **nondéfinie**.

L'exception *RANGEFAIL* est causée si une partie dynamique de la vérification des relations ci-dessus ne se vérifie pas.

5.2.8 Valeurs élément de matrice

syntaxe:

<valeur élément de matrice> ::= *(1)*
<valeur primitive matrice> (*<liste d'expressions>*) *(1.1)*

NOTE – Si la *valeur primitive matrice* est un *locus matrice*, la construction syntaxique est ambiguë et sera interprétée comme un *élément de matrice* (voir 4.2.8).

syntaxe dérivée: voir 4.2.8.

sémantique: une valeur élément de matrice donne une valeur qui est un élément de la valeur matrice spécifiée indiquée par *expression*.

propriétés statiques: la classe d'une *valeur élément de matrice* est la M-classe par valeur, où M est le mode **des éléments** du mode de la *valeur primitive matrice*.

La *valeur élément de matrice* est **constante** si et seulement si la *valeur primitive matrice* et l'*expression* sont **constantes**.

conditions statiques: la classe de l'*expression* doit être **compatible** avec le mode **indice** du mode de la *valeur primitive matrice*.

conditions dynamiques: la valeur donnée par une *valeur élément de matrice* ne doit pas être une valeur **nondéfinie**.

L'exception *RANGEFAIL* est causée si la relation suivante ne se vérifie pas:

$L \leq expression \leq U$

où *L* et *U* sont respectivement la **borne inférieure** et la **borne supérieure** (éventuellement dynamique) du mode de la *valeur primitive matrice*.

5.2.9 Valeurs bande de matrice

syntaxe:

<valeur bande de matrice> ::= *(1)*
<valeur primitive matrice> (*<élément inférieur>* : *<élément supérieur>*) *(1.1)*
| *<valeur primitive matrice>* (*<premier élément>* **UP** *<taille de segment>*) *(1.2)*

NOTE – Si la *valeur primitive matrice* est un *locus matrice*, la construction syntaxique est ambiguë et sera interprétée comme une *bande de matrice* (voir 4.2.9).

sémantique: une valeur bande de matrice donne une valeur matrice (éventuellement dynamique) qui est la partie de la valeur matrice spécifiée indiquée par l'*élément inférieur* et l'*élément supérieur*, ou par le *premier élément* et la *taille de segment*. La **borne inférieure** de la valeur bande de matrice est égale à la **borne inférieure** de la valeur matrice spécifiée; la **borne supérieure** (éventuellement dynamique) est déterminée à partir des expressions spécifiées.

propriétés statiques: la classe (éventuellement dynamique) d'une *valeur bande de matrice* est la M-classe par valeur où M est un mode matrice **paramétré** construit comme:

&nom (indice supérieur)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) de la *valeur primitive* *matrice* et *l'indice supérieur* est soit une expression dont la classe est **compatible** avec les classes de *l'élément inférieur* et de *l'élément supérieur* et donne une valeur telle que:

NUM (*indice supérieur*) = *NUM* (*L*) + *NUM* (*élément supérieur*) – *NUM* (*élément inférieur*)

soit une expression dont la classe est **compatible** avec la classe du *premier élément* et donne une valeur telle que:

NUM (*indice supérieur*) = *NUM* (*L*) + *NUM* (*taille de segment*) – 1

où *L* est la **borne inférieure** du mode de la *valeur primitive* *matrice*.

La classe d'une *valeur bande de matrice* est une classe statique si *l'indice supérieur* est **littéral**: c'est-à-dire que *l'élément inférieur* et *l'élément supérieur* sont tous deux **littéraux** ou que la *taille de segment* est **littérale**; sinon, la classe est une classe dynamique.

conditions statiques: les classes de *l'élément inférieur* et de *l'élément supérieur* ou la classe du *premier élément* doivent être **compatibles** avec le mode **indice** de la *valeur primitive* *matrice*.

Les relations suivantes doivent être vérifiées:

$L \leq NUM$ (*élément inférieur*) $\leq NUM$ (*élément supérieur*) $\leq U$

$1 \leq NUM$ (*taille de segment*) $\leq NUM$ (U) – NUM (L) + 1

NUM (L) $\leq NUM$ (*premier élément*) $\leq NUM$ (*premier élément*) + NUM (*taille de segment*) – 1 $\leq NUM$ (U)

où *L* et *U* sont, respectivement, la **borne inférieure** et la **borne supérieure** du mode de la *valeur primitive* *matrice*. Si *U* et la valeur de toutes les *expressions* sont connus statiquement, les relations peuvent être vérifiées statiquement.

Une *valeur bande de matrice* est **constante** si et seulement si la *valeur primitive* *matrice* et l'*indice supérieur* sont **constants**.

conditions dynamiques: la valeur donnée par une *valeur bande de matrice* ne doit pas être une valeur **nondéfinie**.

L'exception *RANGEFAIL* est causée si une partie dynamique de la vérification des relations ci-dessus ne se vérifie pas.

5.2.10 Valeurs champ de structure

syntaxe:

<valeur champ de structure> ::= *(1)*
<valeur primitive structure> . <nom de champ> *(1.1)*

NOTE – Si la *valeur primitive* *structure* est un *locus* *structure*, la construction syntaxique est ambiguë et sera interprétée comme un *champ de structure* (voir 4.2.10).

sémantique: une valeur champ de structure donne une valeur qui est un champ de la valeur structure spécifiée indiquée par le *nom de champ*. Si la *valeur primitive* *structure* a un mode structure **variable sans étiquettes** et que le *nom de champ* est un nom de **champ récurrent**, la sémantique est définie par l'implémentation.

propriétés statiques: la classe d'une *valeur champ de structure* est la M-classe par valeur où M est le mode du *nom de champ*.

Une *valeur champ de structure* est **constante** si et seulement si la *valeur primitive* *structure* est **constante**.

conditions statiques: le *nom de champ* doit appartenir à l'ensemble des noms **de champ** du mode de la *valeur primitive* *structure*.

conditions dynamiques: la valeur donnée par une *valeur champ de structure* ne peut pas être une valeur **nondéfinie**.

Une valeur ne doit pas dénoter:

- un mode structure **variable avec étiquettes** et que la ou les valeurs des champs **étiquettes** associés indiquent que le champ dénoté n'existe pas;
- un mode structure **paramétré** dynamique et que la liste de valeurs associée indique que le champ n'existe pas.

Les conditions susmentionnées sont appelées conditions d'accès au champ récurrent pour la valeur. (Noter que la condition n'inclut pas d'exception.) L'exception *TAGFAIL* est causée si elles ne sont pas satisfaites pour la *valeur primitive* *structure*.

exemple:

11.140 b (lin)(col).status *(1.1)*

5.2.11 Conversions d'expression

syntaxe:

<conversion d'expression> ::= *(1)*
<nom de mode> # (<expression>) *(1.1)*

NOTE – Si *l'expression* est un *locus de mode statique*, la construction syntaxique est ambiguë et sera interprétée comme une *conversion de locus* (voir 4.2.13).

sémantique: une conversion d'expression prend le pas sur les règles de vérification de compatibilité et des modes du CHILL. Elle attache explicitement un mode à l'expression sans changement de la présentation interne.

propriétés statiques: la classe de la *conversion d'expression* est la M-classe par valeur, où M est le *nom de mode*. Une *conversion d'expression* est **constante** si et seulement si *l'expression* est **constante**.

conditions statiques: le *nom de mode* ne doit pas avoir la **propriété de non-valeur**. La taille du mode **racine** de l'*expression* et la taille du *nom de mode* doivent être égales.

5.2.12 Conversion de représentation

syntaxe:

<conversion de représentation> ::= *(1)*
<nom de mode> (<expression>) *(1.1)*

sémantique: une conversion de représentation annule les règles de contrôle de mode et de compatibilité du langage CHILL. Elle rattache explicitement un mode à l'expression et peut modifier la représentation interne de la valeur donnée par l'expression proprement dite. Si le mode du *nom de mode* est un mode discret et que la classe de la valeur donnée par l'expression est discrète, la valeur donnée par la conversion de représentation est telle que:

NUM (*nom de mode* (*expression*)) = *NUM* (*expression*)

Une conversion de représentation dans laquelle le *nom de mode* et le mode **racine** de la classe de l'expression sont respectivement:

- un mode entier et un mode virgule flottante;
- un mode virgule flottante et un mode entier;
- un mode virgule flottante et un autre mode virgule flottante avec des modes **racine** différents,

peut donner lieu à une approximation. Si la valeur émise par l'expression est exactement représentable au moyen de l'ensemble des valeurs du *nom de mode*, le résultat de la conversion de représentation est la valeur de l'expression proprement dite, sinon elle est l'une des deux valeurs appartenant à l'ensemble des valeurs du *nom de mode* qui délimitent l'intervalle plus petit contenant la valeur donnée par l'expression. Une conversion de représentation dans laquelle le *nom de mode* est un mode entier et le mode **racine** de la classe de l'expression un mode de durée, donne une valeur entière qui représentante, en millisecondes, la valeur donnée par l'expression.

Une conversion de représentation dans laquelle le *nom de mode* ou le mode **racine** de la classe de l'expression est un mode structure, et l'autre un mode structure **paramétré** dont le mode structure **originel** est **similaire** à celui-ci, donne une valeur de structure dans laquelle les valeurs des champs sont égales aux valeurs correspondantes de l'expression, s'il y en a une. Sinon, le résultat est défini par l'implémentation. On notera que pour les valeurs de structure **variable sans étiquettes** et pour les valeurs de structure **variable avec étiquettes** dans lesquelles la liste des valeurs d'étiquetage est différente de celle du mode de structure **paramétrée**, le résultat de la conversion de représentation est défini par l'implémentation.

Si, dans une conversion de représentation dans laquelle le mode M du *nom de mode* est un mode de référence, dans laquelle la classe de l'expression est la classe **nulle** et dans laquelle le résultat de la conversion de représentation est **nul,** M est **compatible** avec la classe de –> ((*expression*) –>), le résultat y est égal; sinon, le résultat est défini par l'implémentation.

Dans les autres cas la valeur donnée par la *conversion de représentation* est définie par l'implémentation et peut dépendre de la représentation interne des valeurs.

propriétés statiques: la classe de la *conversion de représentation* est la M-classe par valeur, où M est le *nom de mode*. Une *conversion de représentation* est **constante** si, et seulement si, l'*expression* est **constante**.

conditions statiques: le *nom de mode* ne doit pas avoir de **propriété de non-valeur**. Une implémentation peut imposer des conditions statiques additionnelles.

conditions dynamiques: dans le cas d'une *expression* qui n'est pas **constante**:

- une exception *RANGEFAIL* se présente si le *nom de mode* est un mode de durée et que le mode **racine** de la classe de l'expression est un mode entier (ou inversement) et que la valeur donnée par la *conversion de représentation* n'appartient pas à l'ensemble des valeurs définies pour le *nom de mode*;
- une exception *OVERFLOW* survient si:
  - la classe de la valeur donnée par l'*expression* est discrète et que le mode du *nom de mode* est un mode discret qui ne définit pas une valeur avec une représentation interne égale à *NUM* (*expression*);
  - le mode du *nom de mode* et le mode **racine** de la classe de l'*expression* sont, indépendamment, un mode entier ou un mode virgule flottante et que l'*expression* donne une valeur qui ne se situe pas entre les bornes du mode **racine** du *nom de mode*;
- une exception *UNDERFLOW* survient si le *nom de mode* et le mode **racine** de la classe de l'*expression* sont des modes virgule flottante et que la valeur donnée par *expression* est supérieure à la **limite inférieure négative** et inférieure à la **limite inférieure positive** du *nom de mode*, et qu'elle est différente de zéro.

Une implémentation peut imposer des conditions dynamiques additionnelles qui, en cas de non respect, produisent une exception définie par l'implémentation.

5.2.13 Appels de procédure rendant valeur

syntaxe:

<appel de procédure rendant valeur> ::= *(1)*
<appel de procédure rendant valeur> *(1.1)*

sémantique: un appel de procédure rendant valeur donne la valeur retournée par la procédure.

propriétés statiques: la classe d'un *appel de procédure rendant valeur* est la M-classe par valeur où M est le mode **spec de résultat** de *l'appel de procédure rendant valeur*.

conditions dynamiques: l'*appel de procédure rendant valeur* ne doit pas donner de valeur **nondéfinie** (voir 5.3.1 et 6.8).

exemples:

6.50 *julian_day_number ([10,dec,1979])* *(1.1)*

11.63 *ok_bishop(b,m)* *(1.1)*

5.2.14 Appels de routine prédéfinie rendant valeur

syntaxe:

<appel de routine prédéfinie rendant valeur> ::= *(1)*
<appel de routine prédéfinie rendant valeur> *(1.1)*

sémantique: un appel de routine prédéfinie rendant valeur fournit la valeur renvoyée par la routine prédéfinie.

propriétés statiques: la classe attachée à l'*appel de routine prédéfinie rendant valeur* est celle de *l'appel de routine prédéfinie rendant valeur*.

conditions dynamiques: l'*appel de routine prédéfinie rendant valeur* ne doit pas donner de valeur **indéfinie** (voir 5.3.1 et 6.8).

5.2.15 Expressions démarrer

syntaxe:

<expression démarrer> ::= *(1)*
START *<nom de processus>* ([*<liste de paramètres effectifs>*]) *(1.1)*

sémantique: l'évaluation de l'expression démarrer crée et active un nouveau processus dont la définition est identifiée par le *nom de processus* (voir l'article 11). L'expression démarrer donne une valeur instance univoque identifiant le processus créé. Le passage de paramètres est analogue au passage de paramètres pour les procédures; pourtant, des paramètres effectifs additionnels peuvent être donnés avec une signification dépendant de l'implémentation.

propriétés statiques: la classe de *l'expression démarrer* est la *INSTANCE*-classe par dérivation.

conditions statiques: le nombre d'occurrences de *paramètre effectif* dans la *liste de paramètres effectifs* ne doit pas être plus petit que le nombre d'occurrences de *paramètre formel* dans la *liste de paramètres formels* de la définition de

processus du *nom de processus*. Si le nombre de paramètres effectifs est *m* et que le nombre de paramètres formels est *n* ($m \geq n$), les règles de compatibilité et **de régionalité** pour les *n* premiers paramètres effectifs sont les mêmes que pour le passage de paramètres à des procédures (voir 6.7). Les conditions statiques pour le reste des paramètres effectifs sont définies par l'implémentation.

conditions dynamiques: pour le passage de paramètres, les conditions d'affectation de toute valeur effective par rapport au mode du paramètre formel associé s'appliquent (voir 6.7).

L'*expression démarrer* cause l'exception *SPACEFAIL* si les besoins de mémoire ne peuvent pas être satisfaits.

exemple:

15.35 **START** *counter ()* *(1.1)*

5.2.16 Opérateur nullaire

syntaxe:

<opérateur nullaire> ::= *(1)*
THIS *(1.1)*

sémantique: l'opérateur nullaire donne la valeur instance univoque qui identifie le processus qui l'exécute. S'il est exécuté par un locus tâche, un exception *THIS_FAIL* se produit.

propriétés statiques: la classe de *l'opérateur nullaire* est la *INSTANCE*-classe par dérivation.

conditions statiques: l'*opérateur nullaire* **THIS** ne doit pas se produire dans une définition de *mode tâche*.

5.2.17 Expression parenthésée

syntaxe:

<expression parenthésée> ::= *(1)*
(<expression>) *(1.1)*

sémantique: une expression parenthésée donne la valeur rendue par l'évaluation de l'expression.

propriétés statiques: la classe de *l'expression parenthésée* est la classe de *l'expression*.

Une *expression parenthésée* est **constante** (**littérale**) si et seulement si *l'expression* est **constante** (**littérale**).

exemple:

5.10 *(a1* ***OR*** *b1)* *(1.1)*

5.3 Valeurs et expressions

5.3.1 Généralités

syntaxe:

<valeur> ::= *(1)*
<expression> *(1.1)*
| *<valeur indéfinie>* *(1.2)*

<valeur indéfinie> ::= *(2)*
* *(2.1)*
| *<nom de synonyme indéfini>* *(2.2)*

sémantique: une valeur est une valeur **indéfinie** ou une valeur (définie par le CHILL) donnée comme le résultat de l'évaluation d'une expression.

Sauf indication explicite du contraire, l'ordre d'évaluation des composantes d'une expression et de leurs sous-composantes, etc., est indéfini et ils peuvent être considérés comme étant évalués en ordre mixte. Il faut les évaluer seulement pour que la valeur à fournir soit déterminée avec précision. Si le contexte exige une expression **constante** ou **littérale**, on suppose que l'évaluation est faite avant l'exécution et ne peut pas causer d'exception. L'implémentation définira des valeurs permises pour les expressions **constantes** et **littérales** et pourra refuser un programme si cette évaluation avant l'exécution livre une valeur sortant des bornes définies par l'implémentation.

propriétés statiques: la classe d'une *valeur* est la classe de *l'expression* ou de la *valeur indéfinie*, respectivement.

La classe de la *valeur indéfinie* est la classe **toute** si la valeur est un ∗; sinon, la classe est la classe du *nom* de *synonyme indéfini*.

Une *valeur* est **constante** si et seulement si c'est une *valeur indéfinie* ou une *expression* qui est **constante**. Une valeur est **littérale** si et seulement si c'est une expression qui est **littérale**.

propriétés dynamiques: une valeur est dite **indéfinie** si elle est dénotée par la *valeur indéfinie* ou lorsque c'est explicitement indiqué dans la présente Recommandation | Norme Internationale. Une valeur composite est **indéfinie** si et seulement si tous ses sous-composants (c'est-à-dire valeurs sous-chaîne, valeurs élément, valeurs champ) sont **indéfinis**.

exemple:

6.40 *(146_097∗c)/4+(1_461∗y)/4*
*+(153*m+2)/5+day+1_721_119* *(1.1)*

5.3.2 Expressions

syntaxe:

<*expression*> *::=* *(1)*
<*opérande–0*> *(1.1)*
| <*expression conditionnelle*> *(1.2)*

<*expression conditionnelle*> *::=* *(2)*
| **IF** <*expression booléenne*> <*alternative alors*>
<*alternative sinon*> **FI** *(2.1)*
| **CASE** <*liste de sélecteurs de cas*> **OF** { <*alternative cas de valeur*> }$^{+}$
[**ELSE** <*sous-expression*>] **ESAC** *(2.2)*

<*alternative alors*> *::=* *(3)*
THEN <*sous-expression*> *(3.1)*

<*alternative sinon*> *::=* *(4)*
ELSE <*sous-expression*> *(4.1)*
| **ELSIF** <*expression booléenne*>
<*alternative alors*> <*alternative sinon*> *(4.2)*

<*sous-expression*> *::=* *(5)*
<*expression*> *(5.1)*

<*alternative cas de valeur*> *::=* *(6)*
<*spécification d'étiquette de cas*> : <*sous-expression*> ; *(6.1)*

sémantique: si **IF** est spécifié, l'*expression booléenne* est évaluée et si elle donne *TRUE*, le résultat est la valeur donnée par la *sous-expression* dans l'*alternative alors*; sinon celle de l'*alternative sinon.*

La valeur fournie par une *alternative sinon* est celle de la *sous-expression* si **ELSE** est spécifié, sinon l'*expression booléenne* est évaluée et si elle donne *TRUE* c'est la valeur donnée par la *sous-expression* de l'*alternative alors*; sinon c'est celle de l'*alternative sinon.*

Si **CASE** est spécifié, les *sous-expressions* dans la liste de *sélecteurs de cas* sont évaluées et si une *spécification d'étiquette de cas* correspond, le résultat est la valeur donnée par la *sous-expression* correspondante; sinon, celui de la sous-expression suivant **ELSE** (qui sera présent).

Les sous-expressions non utilisées dans une *expression conditionnelle* ne sont pas évaluées.

propriétés statiques: si une *expression* est un *opérande–0*, la classe de l'*expression* est la classe de *l'opérande–0*. Si c'est une *expression conditionnelle*, la classe de l'*expression* est la M-classe par valeur, où M est le mode qui dépend du contexte dans lequel l'*expression conditionnelle* se produit selon les règles qui définissent aussi le mode de la classe d'un multiplet sans *nom de mode* (voir 5.2.5).

Une *expression* est **constante** (**littérale**) si et seulement si c'est un *opérande–0* qui est **constant** (**littéral**), une *expression conditionnelle* dans laquelle toutes les *expressions booléennes* ou *listes de sélecteurs de cas* sont **constantes** (**littérales**) et dans laquelle toutes les sous-expressions sont **constantes** (**littérales**).

conditions statiques: si une *expression* est une *expression conditionnelle*, les conditions suivantes s'appliquent:

- une *expression conditionnelle* peut se produire seulement dans des contextes dans lesquels peut intervenir un multiplet non précédé d'*un nom de mode*;

- chaque *sous-expression* doit être **compatible** avec le mode qui découle du contexte avec les mêmes règles que pour les multiplets. Cependant, la partie dynamique de la relation de compatibilité s'applique seulement à *la sous-expression* choisie;
- si **CASE** est spécifié, les conditions de sélection de cas doivent être remplies (voir 12.3) et les mêmes caractéristiques d'exhaustivité, de cohérence et de compatibilité que pour le cas action doivent être assurées (voir 6.4);
- pas d'*expression conditionnelle* peut comporter deux occurrences *de sous-expression*, respectivement **extrarégionale** et **intrarégionale** (voir 11.2.2).

conditions dynamiques: dans le cas d'une *expression conditionnelle*, les conditions d'affectation de la valeur fournie par la *sous-expression* choisie par rapport au mode M découlant du contexte sont applicables.

5.3.3 Opérande–0

syntaxe:

<opérande–0> ::= *(1)*
<opérande–1> *(1.1)*
| *<sous-opérande–0>* { **OR** | **ORIF** | **XOR** } *<opérande–1>* *(1.2)*

<sous-opérande–0> ::= *(2)*
<opérande–0> *(2.1)*

sémantique: si **OR**, **ORIF** ou **XOR** est spécifié, le *sous-opérande–0* et *l'opérande–1* donnent:

- des valeurs booléennes, auquel cas **OR** et **XOR** désignent respectivement les opérateurs logiques "disjonction inclusive" et "disjonction exclusive" donnant une valeur booléenne. Si **ORIF** est spécifié et si l'*opérande–0* donne une valeur booléenne qui est *TRUE*, il s'agit alors du résultat, sinon le résultat est l'*opérande–1*;
- des valeurs chaîne binaire, auquel cas **OR** et **XOR** désignent les opérations logiques sur chaque élément des chaînes binaires, donnant une valeur chaîne binaire;
- des valeurs ensemblistes, auquel cas **OR** désigne l'union des deux valeurs ensemblistes et **XOR** désigne la valeur ensembliste composée des valeurs primitives qui se trouvent dans une seule des valeurs ensemblistes spécifiées (par exemple, *A* **XOR** *B* = *A*–*B* **OR** *B*–*A*).

propriétés statiques: si un *opérande–0* est un *opérande–1*, la classe de l'*opérande–0* est celle de l'*opérande–1*. Si **OR**, **ORIF** ou **XOR** est spécifié, la classe de l'*opérande–0* est la **classe résultante** des classes du *sous-opérande–0* et de l'*opérande–1*.

Un *opérande–0* est **constant** (**littéral**) si et seulement s'il s'agit d'un *opérande–1* qui est **constant** (**littéral**), ou s'il est construit à partir d'un *opérande–0* et d'un *opérande–1* qui sont tous deux **constants** (**littéraux**).

conditions statiques: si **OR**, **ORIF** ou **XOR** est spécifié, la classe du *sous-opérande–0* doit être **compatible** avec la classe de l'*opérande–1*. Si **ORIF** est spécifié, les deux classes doivent avoir un mode **racine** booléen, sinon les deux classes doivent avoir un mode **racine** booléen, ensembliste ou chaîne binaire et la **longueur effective** du *sous-opérande4–0* et de l'*opérande–1* doit être identique. Cette vérification est dynamique si l'un des modes, ou les deux, sont des modes dynamiques ou chaîne **variable**.

conditions dynamiques: dans le cas de **OR** ou de **XOR**, une exception *RANGEFAIL* se produit si l'un des opérandes, ou les deux, ont une classe dynamique et si la partie dynamique de la vérification de compatibilité précitée échoue.

exemples:

10.31 *i<min* *(1.1)*

10.31 *i<min* **OR** *i>max* *(1.2)*

5.3.4 Opérande–1

syntaxe:

<opérande–1> ::= *(1)*
<opérande–2> *(1.1)*
| *<sous-opérande–1>* { **AND** | **ANDIF** } *<opérande–2>* *(1.2)*

<sous-opérande–1> ::= *(2)*
<opérande–1> *(2.1)*

sémantique: si **AND** ou **ANDIF** est spécifié, le *sous-opérande–1* et l'*opérande–2* donnent:

- des valeurs booléennes, auquel cas **AND** désigne l'opération logique "conjonction" donnant une valeur booléenne. Si **ANDIF** est spécifié et si *le sous-opérande–1* donne la valeur booléenne *FALSE*, il s'agit du résultat, sinon le résultat est la valeur donnée par *l'opérande–2*;
- des valeurs chaîne binaire, auquel cas **AND** désigne l'opération logique sur chaque élément des chaînes binaires donnant une valeur chaîne binaire;
- des valeurs ensemblistes, auquel cas **AND** désigne l'opération "intersection" de valeurs ensemblistes donnant une valeur ensembliste comme résultat.

propriétés statiques: si un *opérande–1* est un *opérande–2*, la classe de *l'opérande–1* est celle de *l'opérande–2*.

Si **AND** ou **ANDIF** est spécifié, la classe de *l'opérande–1* est la **classe résultante** des classes du *sous-opérande–1* et de *l'opérande–2*.

Un *opérande–1* est **constant** (**littéral**) si et seulement si c'est un *opérande–2* qui est **constant** (**littéral**) ou s'il est construit à partir d'un *opérande–1* et d'un *opérande–2* qui sont tous deux **constants** (**littéraux**).

conditions statiques: si **AND** ou **ANDIF** est spécifié, la classe du *sous-opérande–1* doit être **compatible** avec celle de *l'opérande–2*. Si **ANDIF** est spécifié, les deux classes doivent avoir un mode **racine** booléen, sinon les deux classes doivent avoir un mode **racine** booléen, ensembliste ou chaîne binaire et la **longueur effective** du *sous-opérande–1* et de *l'opérande–2* doit être la même. Cette vérification est dynamique si l'un des modes, ou les deux, sont des modes dynamiques ou des modes chaîne **variable**.

conditions dynamiques: dans le cas de **AND**, une exception *RANGEFAIL* se produit si l'un des opérandes, ou les deux, ont une classe dynamique et si la partie dynamique de la vérification de compatibilité précitée échoue.

exemples:

5.10 (*a1* **OR** *b1*) *(1.1)*

5.10 **NOT** *k2* **AND** (*a1* **OR** *b1*) *(1.2)*

5.3.5 Opérande–2

syntaxe:

<opérande–2> ::= *(1)*
<*opérande–3*> *(1.1)*
| <*sous-opérande–2*> <*opérateur–3*> <*opérande–3*> *(1.2)*

<sous-opérande–2> ::= *(2)*
<*opérande–2*> *(2.1)*

<opérateur–3> ::= *(3)*
<*opérateur relationnel*> *(3.1)*
| <*opérateur d'appartenance*> *(3.2)*
| <*opérateur d'inclusion ensembliste*> *(3.3)*

<opérateur relationnel> ::= *(4)*
= | /= | > | >= | < | <= *(4.1)*

<opérateur d'appartenance> ::= *(5)*
IN *(5.1)*

<opérateur d'inclusion ensembliste> ::= *(6)*
<= | >= | < | > *(6.1)*

sémantique: l'opérateur d'égalité (=) et les opérateurs d'inégalité (/=) sont définis entre toutes les valeurs d'un mode donné. Les autres opérateurs relationnels (inférieur à: <, inférieur ou égal à: <=, supérieur à: >, supérieur ou égal à: >=) sont définis entre les valeurs d'un mode donné discret, temporisation, chaîne ou virgule flottante. Tous les opérateurs relationnels donnent une valeur booléenne comme résultat.

L'opérateur d'appartenance est défini entre une valeur primitive et une valeur ensembliste. L'opérateur donne *TRUE* si la valeur primitive est dans la valeur ensembliste spécifiée, sinon *FALSE*.

Les opérateurs d'inclusion ensembliste sont définis entre valeurs ensemblistes pour tester si oui ou non une valeur ensembliste est contenue dans: <=, est strictement contenue dans: <, contient: >= ou contient strictement: > l'autre valeur ensembliste. Un opérateur d'inclusion ensembliste donne une valeur booléenne comme résultat.

propriétés statiques: si un *opérande–2* est un *opérande–3*, la classe de *l'opérande–2* est la classe de *l'opérande–3*. Si un *opérateur–3* est spécifié, la classe de *l'opérande–2* est la *BOOL*-classe par dérivation.

Un *opérande–2* est **constant** (**littéral**) si et seulement s'il est soit un *opérande–3* qui est **constant** (**littéral**), soit s'il est construit à partir d'un *sous-opérande–2* et d'un *opérande–3* qui sont tous deux **constants** (**littéraux**).

conditions statiques: si un *opérateur–3* est spécifié, les conditions de compatibilité suivantes entre la classe de *sous-opérande–2* et celle de l'*opérande–3* doivent se vérifier:

- si l'*opérateur–3* est = ou /=, les deux classes doivent être **compatibles**;
- si l'*opérateur–3* est un *opérateur relationnel* autre que = ou /=, les deux classes doivent être **compatibles** et doivent avoir un mode **racine** discret, temporisation, chaîne ou **racine** à virgule flottante;
- si l'*opérateur–3* est l'*opérateur d'appartenance*, la classe d'*opérande–3* doit avoir un mode **racine** ensembliste et la classe de *sous-opérande–2* doit être **compatible** avec le mode **primitif** de ce mode **racine**;
- si l'*opérateur–3* est un *opérateur d'inclusion ensembliste*, les classes doivent être **compatibles** et doivent avoir un mode **racine** ensembliste.

conditions dynamiques: dans le cas d'un *opérateur relationnel*, une exception *RANGEFAIL* ou *TAGFAIL* est causée si l'un ou les deux opérandes ont une classe dynamique et si la partie dynamique de la vérification de compatibilité précitée échoue. L'exception *TAGFAIL* est causée si et seulement si une classe dynamique est basée sur un mode structure **paramétré** dynamique.

exemples:

10.50 NULL *(1.1)*

10.50 last=NULL *(1.2)*

5.3.6 Opérande–3

syntaxe:

```
<opérande–3> ::=                                                    (1)
        <opérande–4>                                                (1.1)
    |   <sous-opérande–3> <opérateur–4> <opérande–4>                 (1.2)

<sous-opérande–3> ::=                                               (2)
        <opérande–3>                                                (2.1)

<opérateur–4> ::=                                                   (3)
        <opérateur arithmétique additif>                            (3.1)
    |   <opérateur de concaténation de chaîne>                       (3.2)
    |   <opérateur de différence ensembliste>                       (3.3)

<opérateur arithmétique additif> ::=                                (4)
        + | –                                                       (4.1)

<opérateur de concaténation de chaîne> ::=                          (5)
        //                                                          (5.1)

<opérateur de différence ensembliste> ::=                           (6)
        –                                                           (6.1)
```

sémantique: si l'*opérateur–4* est un opérateur arithmétique additif, les deux opérandes donnent des valeurs entières ou à virgule flottante et la valeur entière ou valeur à virgule flottante résultante est la somme (+) ou la différence (–) des deux valeurs.

Si l'*opérateur–4* est un opérateur de concaténation de chaîne, les deux opérandes donnent soit des valeurs chaîne binaire soit des valeurs chaîne de caractères; la valeur résultante consiste en la concaténation de ces valeurs. Des valeurs booléennes (de caractère) sont également autorisées; elles sont considérées comme des valeurs chaîne binaire (de caractères) de longueur 1.

Si l'*opérateur–4* est l'opérateur de différence ensembliste, les deux opérandes donnent des valeurs ensemblistes et la valeur résultante est la valeur ensembliste formée de ces valeurs primitives qui sont dans la valeur donnée par *sous-opérande–3* et pas dans la valeur donnée par *opérande–4*.

Si la classe de l'*opérande–3* a un mode **racine** à virgule flottante, le résultat est la valeur à virgule flottante la plus proche – compte tenu du même critère que celui utilisé pour la conversion de représentation – du résultat de l'opération mathématique exacte.

propriétés statiques: si un *opérande–3* est un *opérande–4*, la classe de l'*opérande–3* est la classe de l'*opérande–4*. Si on spécifie un *opérateur–4*, la classe de l'*opérande–3* est déterminée par l'*opérateur–4* comme suit:

- si l'*opérateur–4* est l'*opérateur de concaténation de chaîne*, la classe de l'*opérande–3* dépend des classes de l'*opérande–4* et du *sous-opérande–3*, dans lesquelles l'opérande qui est une valeur booléenne ou de caractère est considéré comme une valeur dont la classe est respectivement une **BOOLS** (1)-classe par dérivation ou une **CHARS** (1)-classe par dérivation:
  - si aucune des deux n'est **forte**, la classe est la **BOOLS** *(n)*-classe par dérivation ou **CHARS** *(n)*-classe par dérivation, selon que les deux opérandes sont des chaînes binaires ou de caractères, où *n* est la somme des **longueurs de chaîne** des modes **racine** des deux classes;
  - sinon, la classe est la *&nom(n)*-classe par valeur, où *&nom* est un nom de **synmode** virtuel **synonyme** du mode **racine** de la classe **résultante des classes** des opérandes et *n* est la somme des **longueurs de chaîne** des modes **racine** des deux classes;

    (cette classe est dynamique si l'un ou les deux opérandes ont une classe dynamique);
- si l'*opérateur–4* est un *opérateur arithmétique additif* ou *opérateur de différence ensembliste*, la classe de l'*opérande–3* est la **classe résultante** des classes de l'*opérande–4* et du *sous-opérande–3*.

Un *opérande–3* est **constant** (**littéral**) si et seulement s'il est soit un *opérande–4* qui est **constant** (**littéral**), soit s'il est construit à partir d'un *opérande–3* et d'un *opérande–4* qui sont **constants** (**littéraux**) et que l'*opérateur–4* est soit l'*opérateur arithmétique additif* soit l'*opérateur de différence ensembliste*.

Si l'*opérateur–4* est l'*opérateur de concaténation de chaîne*, un *opérande–3* est **constant** s'il est construit à partir d'un *opérande–3* et d'un *opérande–4* qui sont tous deux **constants**.

conditions statiques: si un *opérateur–4* est spécifié, les conditions de compatibilité suivantes doivent être remplies:

- si l'*opérateur–4* est l'*opérateur arithmétique additif*, les classes des deux opérandes doivent être **compatibles** et avoir toutes les deux un mode **racine** entier ou à virgule flottante. Par ailleurs, si l'*opérande–3* n'est pas **constant**, le mode **racine** de la classe de l'*opérande–3* doit être un mode entier **prédéfini** ou un mode virgule flottante **prédéfini**;
- si l'*opérateur–4* est l'*opérateur de concaténation de chaîne*:
  - les classes des deux opérandes doivent être **compatibles** et avoir toutes deux un mode **racine** chaîne **binaire** ou un mode **racine** chaîne de **caractères**;
  - les classes des deux opérandes doivent être **compatibles** avec le mode *BOOL* ou avec le mode *CHAR*;
  - la classe d'un opérande doit avoir un mode **racine** chaîne **binaire** (**de caractères**), l'autre doit être **compatible** avec le mode *BOOL (CHAR)*;
- si l'*opérateur–4* est un *opérateur de différence ensembliste*, les classes des deux opérandes doivent être **compatibles** et toutes deux doivent avoir un mode **racine** ensembliste.

conditions dynamiques: si, dans le cas d'un *opérande–3* qui n'est pas **constant**, l'*opérateur–4* est un *opérateur arithmétique additif*, une exception *OVERFLOW* est causée si une addition (+) ou une soustraction (–) donne une valeur qui n'est pas une des valeurs définies par le mode **racine** de la classe de l'*opérande–3*, ou bien un ou les deux opérandes n'appartiennent pas à l'ensemble de valeurs du mode **racine** de l'*opérande–3*.

Dans le cas d'un *opérande–3* qui n'est pas **constant**, une exception *UNDERFLOW* est causée si la classe de l'*opérande–3* a un mode **racine** à virgule flottante et que l'addition (+) ou la soustraction (–) mathématiquement exacte donne une valeur qui est plus grande que la **limite supérieure négative** et plus petite que la **limite inférieure positive** du mode **racine** de l'*opérande–3* et différente de zéro.

exemples:

1.6 *j* *(1.1)*

1.6 *i+j* *(1.2)*

5.3.7 Opérande–4

syntaxe:

<opérande–4> ::= *(1)*
<opérande–5> *(1.1)*
| *<sous-opérande–4> <opérateur arithmétique multiplicatif> <opérande–5>* *(1.2)*

<sous-opérande–4> ::= *(2)*
<opérande–4> *(2.1)*

<opérateur arithmétique multiplicatif> ::= *(3)*
∗ | / | **MOD** | **REM** *(3.1)*

sémantique: si l'opérateur arithmétique multiplicatif est spécifié est soit le produit (∗), soit le quotient (/), les *sous-opérande–4* et *l'opérande–5* donnent des valeurs entières ou à virgule flottante et la valeur entière ou à virgule flottante résultante est respectivement le produit ou le quotient des deux valeurs.

Si l'opérateur arithmétique multiplicatif est le modulo (**MOD**) ou le reste de la division (**REM**), le *sous-opérande–4* et l'*opérande–5* donnent des valeurs entières ou à virgule flottante, et la valeur entière ou à virgule flottante résultante est le modulo ou le reste de la division des deux valeurs.

L'opération modulo est définie de telle manière que *i* **MOD** *j* donne l'entier unique *k*, $0 \le k < j$ tel qu'il existe une valeur entière *n* tel que $i = n * j + k$; *j* doit être supérieur à 0.

L'opération quotient se définit de telle sorte que toutes les relations:

ABS (*x*/*y*) = *ABS* (*x*) / *ABS* (*y*) et

signe (*x*/*y*) = signe (*x*) / signe (*y*) et

ABS (*x*) – (*ABS* (*x*) / *ABS* (*y*)) ∗ *ABS* (*y*) = *ABS* (*x*) **MOD** *ABS* (*y*)

donnent *TRUE* pour toutes les valeurs entières de *x* et *y*, où le signe*(x) = –1* si $x < 0$, sinon le signe*(x)* = 1.

L'opération de reste est définie de telle manière que *x* **REM** $y = x – (x/y) * y$ donne *TRUE* pour toutes les valeurs entières de *x* et *y*.

Si la classe de l'*opérande–4* a un mode **racine** à virgule flottante, le résultat est la valeur à virgule flottante qui se rapproche le plus, selon le même critère que celui utilisé pour la conversion de représentation, du résultat de l'opération mathématique exacte.

propriétés statiques: si l'*opérande–4* est un *opérande–5*, la classe de l'*opérande–4* est la classe de l'*opérande–5*; sinon, la classe de l'*opérande–4* est la **classe résultante** des classes du *sous-opérande–4* et de l'*opérande–5*.

Un *opérande–4* est **constant** (**littéral**) si et seulement s'il est soit un *opérande–5* qui est **constant** (**littéral**), soit s'il est construit à partir d'un *opérande–4* et d'un *opérande–5* qui sont tous deux **constants** (**littéraux**).

conditions statiques: si un *opérateur arithmétique multiplicatif* est spécifié entre des opérandes entier ou à virgule flottante, les classes de l'*opérande–5* et du *sous-opérande–4* doivent être **compatibles** et toutes deux doivent avoir respectivement un mode **racine** entier ou un mode **racine** à virgule flottante. Par ailleurs, si l'*opérande–4* n'est pas **constant**, le mode **racine** de la classe d'*opérande–4* doit être un mode entier **prédéfini** ou un mode virgule flottante **prédéfini**.

conditions dynamiques: si dans le cas d'un *opérande–4* qui n'est pas **constant**, un *opérateur arithmétique multiplicatif* est spécifié, une exception *OVERFLOW* est causée si une multiplication (∗) ou une division (/) ou un modulo (**MOD**) ou un reste (**REM**) donne une valeur qui n'est pas l'une des valeurs définies par le mode **racine** de la classe de l'*opérande–4* ou s'effectue sur des valeurs d'opérandes pour lesquelles l'opérateur n'est pas défini mathématiquement, c'est-à-dire division ou reste avec un *opérande–5* donnant 0 ou une opération modulo avec un *opérande–5* donnant une valeur entière non positive, ou bien un ou les deux opérandes n'appartiennent pas à l'ensemble de valeurs du mode **racine** de l'*opérande–4*.

Dans le cas d'un *opérande–4* qui n'est pas **constant**, une exception *UNDERVERFLOW* est causée si la classe de l'opérande–4 a un mode **racine** et que la multiplication (∗) ou la division (/) exacte donne une valeur qui est plus grande que la **limite supérieure négative** et plus petite que la **limite inférieure positive** du mode **racine** de l'*opérande–4* et différente de zéro.

exemples:

6.15 *1_461* *(1.1)*

6.15 *(4 ∗ d + 3) / 1_461* *(1.2)*

5.3.8 Opérande–5

syntaxe:

<opérande–5> ::= *(1)*
<opérande–6> *(1.1)*
| *<sous-opérande–5> <opérateur d'exponentiation> <opérande–6>* *(1.2)*

<sous-opérande–5> ::= *(2)*
<opérande–5> *(2.1)*

<opérateur d'exponentiation> ::= *(3)*
**** *(3.1)*

sémantique: si l'*opérateur d'exponentiation* est spécifié, le *sous-opérande–5* et l'*opérande–6* donnent une valeur à virgule flottante ou une valeur entière. La valeur en question est obtenue en élevant la valeur du *sous-opérande–5* à la puissance de la valeur de l'*opérande–6*.

Si la classe de l'opérande–5 a un mode **racine** à virgule flottante et que l'on applique le même critère que pour la conversion de représentation, le résultat est une valeur à virgule flottante approchée du résultat de l'opération mathématique exacte.

propriétés statiques: si l'*opérande–5* est un *opérande–6*, la classe de l'*opérande–5* est la classe de l'*opérande–6*.

Si l'*opérateur d'exponentiation* est spécifié, la classe de l'*opérande–5* est celle du *sous-opérande–5*.

Un *opérande–5* est **constant** (**littéral**) si, et seulement si, il est un *opérande–6* **constant** (**littéral**) ou s'il est formé d'un *opérande–5* et d'un *opérande–6* qui sont tous deux **constants** (**littéraux**).

conditions statiques: si un *opérateur d'exponentiation* est spécifié:

- et que la classe du *sous-opérande–5* a un mode **racine** à virgule flottante, la classe de *l'opérande–6* doit avoir un mode **racine** entier ou un mode **racine** à virgule flottante;
- sinon la classe du *sous-opérande–5* doit avoir un mode **racine** entier et la classe de l'*opérande–6* un mode **racine** entier.

conditions dynamiques: dans le cas d'un *opérande–5* qui n'est pas **constant**, une exception *OVERFLOW* se produit si une opération d'exponentiation donne une valeur qui se situe hors de l'intervalle du mode **racine** de la classe de l'*opérande–5*.

Dans le cas d'un *opérande–5* qui n'est pas **constant**, une exception *UNDERFLOW* se produit si la classe de l'*opérande–5* a un mode **racine** à virgule flottante et que l'élévation exacte à la puissance donne une valeur qui est plus petite que la **limite inférieure positive** du mode **racine** de l'*opérande–5*.

Si un *opérateur d'exponentiation* est spécifié et que la classe de l'*opérande–5* a un mode **racine** entier, et si l'*opérande–6* n'est pas **constant**, sa valeur doit être supérieure ou égale à zéro.

exemple:

*r ** 4* *(1.2)*

5.3.9 Opérande–6

syntaxe:

<opérande–6> ::= *(1)*
[*<opérateur unaire>*] *<opérande–7>* *(1.1)*
| *<littéral signé entier>* *(1.2)*
| *<littéral signé à virgule flottante>* *(1.3)*

<opérateur unaire> ::= *(2)*
– | **NOT** *(2.1)*
| *<opérateur de répétition de chaîne>* *(2.2)*

<opérateur de répétition de chaîne> ::= *(3)*
(*<expression de littéral entier>*) *(3.1)*

NOTE – Si l'*opérateur unaire* est l'opérateur de changement de signe (–) et que l'*opérande–7* est un *littéral non signé entier* ou un *littéral non signé à virgule flottante*, la construction syntactique est ambiguë et sera interprétée respectivement comme un *littéral signé entier* ou un *littéral signé à virgule flottante.*

sémantique: si l'opérateur unaire est l'opérateur changer-le-signe (–), l'*opérande–7* donne une valeur entière ou une valeur à virgule flottante et la valeur entière ou à virgule flottante résultante est la valeur entière ou à virgule flottante précédente changée de signe.

Si l'opérateur unaire est **NOT**, l'*opérande–7* donne soit une valeur booléenne soit une valeur chaîne binaire, soit une valeur ensembliste. Dans les deux premiers cas, la négation logique de la valeur booléenne ou chaîne binaire est donnée; dans le dernier cas, la valeur ensembliste complémentaire, c'est-à-dire l'ensemble de ces valeurs primitives qui ne sont pas dans la valeur ensembliste opérande, est donnée.

Si l'opérateur unaire est l'opérateur de répétition de chaîne, l'*opérande–7* est un *littéral chaîne de caractères* ou un *littéral chaîne binaire*. Si *l'expression de littéral entier* donne 0, le résultat est la valeur chaîne vide, sinon la valeur chaîne

formée en concaténant la chaîne avec elle-même autant de fois que spécifié par la valeur donnée par l'*expression de littéral entier* moins 1.

propriétés statiques: si *l'opérande–6* est un *opérande–7*, la classe de *l'opérande–6* est la classe de *l'opérande–7*.

Si un *opérateur unaire* est spécifié, la classe de *l'opérande–6* est:

- si *l'opérateur unaire* est – ou **NOT**, alors la **classe résultante** de celle de l'*opérande–7*;
- si *l'opérateur unaire* est *l'opérateur de répétition de chaîne*, alors c'est la **CHARS** *(n)*- ou **BOOLS** *(n)*-classe par dérivation (selon que le littéral était un *littéral chaîne de caractères* ou un *littéral chaîne binaire*) où $n = r * l$, où r est la valeur donnée par *l'expression de littéral entier* et l est la **longueur de chaîne** du littéral chaîne.

Un *opérande–6* est **constant** si et seulement si *l'opérande–7* est **constant**. Un *opérande–6* est **littéral** si et seulement si *l'opérande–7* est **littéral** et que *l'opérateur unaire* est – ou **NOT**.

conditions statiques: si *l'opérateur unaire* est –, la classe de l'*opérande–7* doit avoir un mode **racine** entier ou un mode **racine** à virgule flottante. Par ailleurs, si *l'opérande–6* n'est pas **constant**, le mode **racine** de la classe de *l'opérande–6* doit être un mode entier **prédéfini** ou un mode virgule flottante **prédéfini**.

Si *l'opérateur unaire* est **NOT**, la classe de *l'opérande–7* doit avoir un mode **racine** booléen, chaîne **binaire** ou ensembliste.

Si *l'opérateur unaire* est *l'opérateur de répétition de chaîne*, *l'opérande–7* doit être un *littéral chaîne de caractères* ou un *littéral chaîne binaire*. L'*expression de littéral entier* doit donner une valeur entière non négative.

conditions dynamiques: si *l'opérande–6* n'est pas **constant**, une exception *OVERFLOW* est causée si l'opération changer-de-signe (–) donne une valeur qui n'est pas dans l'une des valeurs définies par le mode **racine** de la classe de *l'opérande–6*.

Si *l'opérande–6* n'est pas **constant**, une exception *UNDERFLOW* se produit si la classe de *l'opérande–6* a un mode **racine** à virgule flottante et que l'opération mathématiquement exacte de changement de signe (–) donne une valeur qui est supérieure à la **limite supérieure négative** et inférieure à la **limite inférieure positive** du mode **racine** de *l'opérande–6*, et qui est différente de zéro.

exemples:

5.10 **NOT** *k2* *(1.1)*

7.54 *(6)" "* *(1.1)*

7.54 *(6)* *(2.2)*

5.3.10 Opérande–7

syntaxe:

<opérande–7> ::= *(1)*
<locus référencé> *(1.1)*
| *<valeur primitive>* *(1.2)*

<locus référencé> ::= *(2)*
–> <locus> *(2.1)*

sémantique: un locus référencé donne une référence au locus spécifié.

propriétés statiques: la classe de *l'opérande–7* est la classe du *locus référencé*, de *l'expression recevoir* ou de la *valeur primitive*, respectivement. La classe du *locus référencé* est la M-classe par référençage, où M est le mode du *locus*.

Un *opérande–7* est **constant** si et seulement si la *valeur primitive* est **constante** ou si le *locus référencé* est **constant**. Un *locus référencé* est **constant** si et seulement si le *locus* est un locus **statique**. Un *opérande–7* est **littéral** si et seulement si la *valeur primitive* est **littérale**.

conditions statiques: le *locus* doit être **référençable**.

exemple:

8.25 *–> c* *(2.1)*

6 Actions

6.1 Généralités

syntaxe:

<énoncé d'action> ::= *(1)*
[*<occurence de définition>* :] *<action>* [*<filet>*] [*<chaîne de nom simple>*]; *(1.1)*
| *<module>* *(1.2)*
| *<module de spec>* *(1.3)*
| *<module de contexte>* *(1.4)*

<action> ::= *(2)*
<action parenthésée> *(2.1)*
| *<action d'affectation>* *(2.2)*
| *<action appeler>* *(2.3)*
| *<action sortir>* *(2.4)*
| *<action revenir>* *(2.5)*
| *<action résulter>* *(2.6)*
| *<action aller>* *(2.7)*
| *<action affirmer>* *(2.8)*
| *<action vide>* *(2.9)*
| *<action démarrer>* *(2.10)*
| *<action arrêter>* *(2.11)*
| *<action mettre en attente>* *(2.12)*
| *<action continuer>* *(2.13)*
| *<action envoyer>* *(2.14)*
| *<action induire>* *(2.15)*

<action parenthésée> ::= *(3)*
<action conditionnelle> *(3.1)*
| *<action de cas>* *(3.2)*
| *<action faire>* *(3.3)*
| *<bloc début-fin>* *(3.4)*
| *<action mettre en attente avec cas>* *(3.5)*
| *<action recevoir avec cas>* *(3.6)*
| *<action temporisation>* *(3.7)*

sémantique: les énoncés d'action constituent la partie algorithmique d'un programme CHILL. Tout énoncé d'action peut être étiqueté et les actions qui ne pourraient jamais causer une exception peuvent ne pas se terminer par un filet.

propriétés statiques: une *occurence de définition* apparaissant dans un *énoncé d'action* définit un nom d'**étiquette**.

conditions statiques: la *chaîne de nom simple* ne peut être donnée qu'après une *action* qui est une *action parenthésée* ou si un *filet* est spécifié et seulement si une *occurence de définition* est spécifiée. La *chaîne de nom simple* doit être la même chaîne que l'*occurence de définition.*

6.2 Action d'affectation

syntaxe:

<action d'affectation> ::= *(1)*
<action d'affectation simple> *(1.1)*
| *<action d'affectation multiple>* *(1.2)*

<action d'affectation simple> ::= *(2)*
<locus> <symbole d'affectation> <valeur> *(2.1)*
| *<locus> <opérateur affectant> <expression>* *(2.2)*

<action d'affectation multiple> ::= *(3)*
<locus> { , *<locus>* }$^{+}$ *<symbole d'affectation> <valeur>* *(3.1)*

<opérateur affectant> ::= *(4)*
<opérateur binaire fermé> <symbole d'affectation> *(4.1)*

<opérateur binaire fermé> ::= *(5)*
OR | **XOR** | **AND** *(5.1)*
| *<opérateur de différence ensembliste>* *(5.2)*
| *<opérateur arithmétique additif>* *(5.3)*
| *<opérateur arithmétique multiplicatif>* *(5.4)*
| *<opérateur de concaténation de chaîne>* *(5.5)*

<symbole d'affectation> ::= *(6)*
:= *(6.1)*

sémantique: une action d'affectation place une valeur dans un ou plusieurs locus.

Si un symbole d'affectation est employé, la valeur donnée par la partie droite est mise dans le ou les locus spécifiés en partie gauche.

Si un opérateur affectant est employé, la valeur contenue dans le locus est combinée avec la valeur partie droite (dans cet ordre) suivant la sémantique de l'opérateur binaire fermé spécifié, et le résultat est remis dans le même locus.

Les évaluations du ou des locus partie gauche et de la valeur partie droite, ainsi que les affectations elles-mêmes sont faites dans un ordre quelconque et peuvent éventuellement se mélanger. Toute affectation peut se faire aussitôt que la valeur et le locus ont été évalués.

Si le locus (ou n'importe lequel des locus) est le champ **étiquette** d'une structure variable, la sémantique des champs récurrents qui en dépendent est définie par l'implémentation.

conditions statiques: les modes de chaque occurrence de *locus* doivent être **équivalents** et ils ne peuvent avoir ni la **propriété de protection**, ni la **propriété de non-valeur**. Chaque mode doit être **compatible** avec la classe de la *valeur*. Les vérifications sont dynamiques dans les cas où il s'agit de locus à mode dynamique et d'une valeur de classe dynamique.

La *valeur* doit être **régionalement sûre** pour chaque *locus* (voir 11.2.2).

Si un *locus* quelconque a un mode chaîne **fixe**, la **longueur de chaîne** du mode et la **longueur effective** de la valeur doivent être les mêmes; sinon, s'il a un mode chaîne **variable**, la **longueur de chaîne** du mode ne doit pas être inférieure à la **longueur effective** de la valeur. Ce contrôle est dynamique si l'un des modes ou les deux sont des modes dynamiques ou des modes chaîne **variable**. Cette condition est appelée condition d'affectation de chaîne.

Si l'une des affectations est de la forme "pvl–> := pvr–>;", où pvl et pvr ont respectivement les modes "REF ML" et "REF MR", et que ML et MR sont des noms de mode moreta, ML et MR doivent être sur le même chemin.

Si l'une des affectations est de la forme "pvl–> := mr;", où pvl a le mode "REF ML", et ML ainsi que le nom de mode de mr sont des noms de mode module, il faut que mr succ ML soit satisfait.

Si l'une des affectations est de la forme "ml := pvr–>;", où pvr a le mode "REF MR", et MR ainsi que le nom de mode de ml sont des noms de mode module, il faut que MR succ ml soit satisfait.

Si le mode de l'un des locus du côté gauche est un mode module, les noms de mode de tous ces modes doivent être synonymes par paires.

conditions dynamiques: l'exception *RANGEFAIL* ou *TAGFAIL* est causée si le mode du locus et/ou celui de la valeur sont des modes dynamiques et si la partie dynamique de la vérification de compatibilité précitée échoue.

L'exception *RANGEFAIL* est causée si le mode du locus et/ou celui de la valeur sont des modes chaîne **variable** et si la partie dynamique de la vérification de compatibilité précitée échoue.

L'exception *RANGEFAIL* est causée si un *locus* quelconque a un mode intervalle discret (mode intervalle virgule flottante) et si la valeur fournie par l'évaluation de *valeur* n'est ni l'une des valeurs définies par le mode intervalle discret (mode intervalle virgule flottante) ni la valeur **indéfinie**.

Si le mode d'un locus L est du type REF MM, où MM est un mode moreta, il faut que soit satisfait ce qui suit: le mode de la valeur effective du rhs doit être un successeur du mode de L; sinon, l'exception *RANGEFAIL* se produit.

Les conditions dynamiques mentionnées ci-dessus avec la condition d'affectation de chaîne sont appelées les conditions d'affectation d'une valeur par rapport à un mode.

Dans le cas d'un *opérateur affectant*, les mêmes exceptions sont causées comme si l'expression:

<locus> *<opérateur binaire fermé>* (*<expression>*)

était évaluée et que la valeur donnée était mise dans le locus spécifié (à noter que le locus n'est évalué qu'une fois).

Si le mode d'un locus L est du type "REF MM", où MM est un mode moreta, il faut que le mode de la valeur effective du rhs soit un successeur du mode de L; sinon, l'exception *RANGEFAIL* se produit.

Si l'une des affectations est de la forme "pvl–> := pvr–>;", "pvl–> := mr;" ou "ml := pvr–>;", où pvl et pvr ont respectivement les modes "REF ML" et "REF MR", et où ML, MR, ml et mr sont des modes module, les modes effectifs du lhs et du rhs doivent satisfaire aux règles d'assignation des modes module.

exemples:

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 Action conditionnelle

syntaxe:

<action conditionnelle> ::= *(1)*
IF *<expression booléenne>* *<clause alors>* [*<clause sinon>*] **FI** *(1.1)*

<clause alors> ::= *(2)*
THEN *<liste d'énoncés d'action>* *(2.1)*

<clause sinon> ::= *(3)*
ELSE *<liste d'énoncés d'action>* *(3.1)*
| **ELSIF** *<expression booléenne>* *<clause alors>* [*<clause sinon>*] *(3.2)*

syntaxe dérivée: la notation:

ELSIF *<expression booléenne>* *<clause alors>* [*<clause sinon>*]

est une syntaxe dérivée pour:

ELSE IF <expression booléenne> <clause alors> [<clause sinon>] **FI**;

sémantique: une action conditionnelle est un branchement conditionnel à deux voies. Si l'*expression booléenne* donne *TRUE*, la liste d'énoncés d'action qui suit **THEN** est entamée; sinon, c'est la liste d'énoncés d'action qui suit **ELSE**, s'il y en a une, qui est entamée.

conditions dynamiques: l'exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemples:

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 Action à cas

syntaxe:

<action à cas> ::= *(1)*
CASE *<liste de sélecteurs de cas>* **OF** [*<liste d'intervalles>*;] { *<cas alternatif>* }$^{+}$
[**ELSE** *<liste d'énoncés d'action>*] **ESAC** *(1.1)*

<liste de sélecteurs de cas> ::= *(2)*
*<expression discrète> { , <expression discrète> }** *(2.1)*

<liste d'intervalles> ::= *(3)*
*<nom de mode discret> { , <nom de mode discret> }** *(3.1)*

<cas alternatif> ::= *(4)*
<spécification d'étiquettes de cas> : <liste d'énoncés d'action> *(4.1)*

sémantique: une action de cas est un branchement multiple. Elle consiste en la spécification d'une ou de plusieurs expressions discrètes (la liste de sélecteurs de cas) et en un certain nombre de listes d'énoncés d'action étiquetées (les cas alternatifs). Cette liste d'énoncés d'action est étiquetée par une spécification d'étiquettes de cas qui consiste en une liste d'étiquettes de cas (une pour chaque sélecteur de cas). Chaque liste d'étiquettes de cas définit un ensemble de valeurs. L'emploi d'une liste d'expressions discrètes dans la liste de sélecteurs de cas permet de choisir un cas suivant plusieurs conditions.

L'action de cas entame la liste d'énoncés d'action pour laquelle les valeurs données dans la spécification d'étiquettes de cas correspondent aux valeurs de la liste des sélecteurs de cas; sinon, la *liste d'énoncés d'action* qui suivent **ELSE** est entamée.

Les expressions figurant dans la liste de sélecteurs de cas sont évaluées dans un ordre indéfini et éventuellement mélangé. Il n'est nécessaire de les évaluer que jusqu'au point où un cas alternatif est déterminé univoquement.

conditions statiques: pour la liste d'occurrences de *spécification d'étiquettes de cas*, les conditions de sélection de cas sont à respecter (voir 12.3).

Le nombre d'occurrences d'*expression discrète* dans la *liste de sélecteurs de cas* doit être égal au nombre de classes dans la **liste résultante des classes** de la liste d'occurrences de *liste d'étiquettes de cas* et, si présente, au nombre d'occurrences de *nom de mode discret* dans la *liste d'intervalles*.

La classe de chaque *expression discrète* dans la *liste de sélecteurs de cas*, doit être **compatible** avec la classe correspondante (par position) de la **liste résultante des classes** des occurrences de *liste d'étiquettes de cas* et, si présente, **compatible** avec le *nom de mode discret* correspondant (par position) de la *liste d'intervalles*. Ce dernier mode doit aussi être **compatible** avec la classe correspondante de la **liste résultante des classes**.

Toute valeur donnée par une *expression de littérale discrète* ou définie par un *intervalle littéral* ou un *nom de mode discret* dans une étiquette de cas (voir 12.3) doit résider dans l'intervalle du *nom de mode discret* correspondant de la *liste d'intervalles*, si présente, et aussi dans l'intervalle défini par le mode de l'*expression discrète* correspondante dans la *liste de sélecteurs de cas*, si c'est une *expression discrète* **forte**. Dans ce dernier cas, les valeurs définies par le *nom de mode discret* correspondant dans la *liste d'intervalles*, si présente, doivent aussi résider dans cet intervalle.

La partie optionnelle **ELSE** selon la syntaxe ne peut s'omettre que si la liste d'occurrences de *liste d'étiquettes de cas* est **complète** (voir 12.3).

conditions dynamiques: l'exception *RANGEFAIL* est causée si une *liste d'intervalles* est spécifiée et que la valeur donnée par une *expression discrète* dans la *liste de sélecteurs de cas* ne se trouve pas entre les bornes spécifiées par le *nom de mode discret* correspondant de la *liste d'intervalles*.

L'exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemples:

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 Action faire

6.5.1 Généralités

syntaxe:

<action faire> ::= *(1)*
DO [*<partie commande>*;] *<liste d'énoncés d'action>* **OD** *(1.1)*

<partie commande> ::= *(2)*
<commande pour> [*<commande tandis>*] *(2.1)*
| *<commande tandis>* *(2.2)*
| *<partie avec>* *(2.3)*

sémantique: une action faire a trois formes différentes: les versions faire-pour et faire-tandis, toutes deux pour boucler, et la version faire-avec comme abréviation adéquate pour accéder à des champs de structure d'une manière efficace. Si aucune partie de commande n'est spécifiée, la liste d'énoncés d'action est entamée une fois, chaque fois que l'action faire est entamée.

Quand la commande pour et la commande tandis sont combinées, la commande tandis est évaluée après la commande pour, et seulement si l'action faire n'est pas terminée par la commande pour.

Si la partie de commande spécifiée concerne une commande pour et/ou une commande tandis, tant que la commande reste à l'intérieur de l'action faire, la liste des énoncés d'action est entamée conformément à la partie de commande, mais le domaine faire n'est pas entamé de nouveau pour chaque exécution de la liste d'énoncés d'action.

conditions dynamiques: l'exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemples:

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 Commande pour

syntaxe:

<commande pour> ::= *(1)*
FOR { *<itération>* {, *<itération>*}* | **EVER** } *(1.1)*

<itération> ::= *(2)*
<énumération de valeur> *(2.1)*
| *<énumération de locus>* *(2.2)*

<énumération de valeur> ::= *(3)*
<énumération par pas> *(3.1)*
| *<énumération par intervalle>* *(3.2)*
| *<énumération ensembliste>* *(3.3)*

<énumération par pas> ::= *(4)*
<compteur de boucle> <symbole d'affectation>
<valeur initiale> [*<valeur de pas>*] [**DOWN**] *<valeur finale>* *(4.1)*

<compteur de boucle> ::= *(5)*
<occurence de définition> *(5.1)*

<valeur initiale> ::= *(6)*
<expression discrète> *(6.1)*

<valeur de pas> ::= *(7)*
BY *<expression entière>* *(7.1)*

<*valeur finale*> *::=* *(8)*
TO <*expression discrète*> *(8.1)*

<*énumération par intervalle*> *::=* *(9)*
<*compteur de boucle*> [**DOWN**] **IN** <*nom de mode discret*> *(9.1)*

<*énumération ensembliste*> *::=* *(10)*
<*compteur de boucle*> [**DOWN**] **IN** <*expression ensembliste*> *(10.1)*

<*énumération de locus*> *::=* *(11)*
<*compteur de boucle*> [**DOWN**] **IN** <*objet composite*> *(11.1)*

<*objet composite*> *::=* *(12)*
<*locus matrice*> *(12.1)*
| <*expression matrice*> *(12.2)*
| <*locus chaîne*> *(12.3)*
| <*expression chaîne*> *(12.4)*

NOTE – Si l'*objet composite* et un *locus* de *chaîne ou un locus de matrice*, on résout l'ambiguïté syntactique en interprétant l'*objet composite* comme un *locus* et non comme une *expression*.

sémantique: la commande pour peut consister en plusieurs compteurs de boucle. Les compteurs de boucle sont évalués chaque fois dans un ordre non spécifié avant d'entamer la liste d'énoncés d'action et il ne faut les évaluer que jusqu'au point où il devient possible de décider de terminer l'action faire. L'action faire est terminée si au moins un des compteurs de boucle indique la terminaison.

1) **do for ever**:

la liste d'énoncés d'action est répétée un nombre indéfini de fois. L'action faire ne peut se terminer que par un transfert de commande en dehors d'elle.

2) **énumération de valeur**:

la liste d'énoncés d'action est entamée de façon répétée pour l'ensemble des valeurs spécifiées des compteurs de boucle. L'ensemble des valeurs est soit spécifié par *un nom de mode discret* (énumération par intervalle), soit par une valeur ensembliste (énumération ensembliste), soit par une valeur initiale, une valeur de pas et une valeur finale (énumération par pas).

Le compteur de boucle définit implicitement un nom qui désigne sa valeur ou locus dans la liste d'énoncés d'action.

énumération par intervalle:

dans le cas d'une énumération par intervalle sans (avec) spécification de **DOWN**, la valeur initiale du compteur de boucle est la plus petite (la plus grande) valeur dans l'ensemble de valeurs défini par le *nom de mode discret*. Pour les exécutions suivantes de la liste d'énoncés d'action, la *valeur suivante* sera évaluée comme:

SUCC(valeur précédente) (PRED(valeur précédente)).

La terminaison normale se produit si la liste d'énoncés d'action a été exécutée pour la plus grande (plus petite) valeur définie par le *nom de mode discret*.

énumération ensembliste:

dans le cas d'une énumération ensembliste sans (avec) spécification de **DOWN**, la valeur initiale du compteur de boucle est la plus petite (la plus grande) valeur primitive dans la valeur ensembliste dénotée. Si la valeur ensembliste est vide, la liste d'énoncés d'action ne sera pas exécutée. Pour les exécutions suivantes de la liste d'énoncés d'action, la valeur suivante sera la valeur primitive suivante la plus grande (la plus petite) dans la valeur ensembliste. L'action faire se termine normalement quand la liste d'énoncés d'action a été exécutée pour la plus grande (plus petite) valeur. Quand l'action faire est exécutée, l'expression **ensembliste** n'est évaluée qu'une fois.

énumération par pas:

dans le cas d'une énumération par pas sans (avec) spécification de **DOWN**, l'ensemble de valeurs du compteur de boucle est déterminé par une valeur initiale, une valeur finale, et, éventuellement, une valeur de pas. Quand l'action faire est exécutée, ces expressions ne sont évaluées qu'une fois dans un ordre non spécifié, et éventuellement mélangé. La valeur de pas est toujours positive. La vérification de terminaison est faite avant chaque exécution de la liste d'énoncés d'action. Initialement, on vérifie que la valeur initiale du compteur de boucle est plus grande (plus petite) que la valeur finale. Pour les exécutions suivantes, *valeur suivante* sera évaluée comme:

valeur précédente + valeur de pas (valeur précédente – valeur de pas)

dans le cas d'une spécification de *valeur de pas*, sinon comme:

SUCC(valeur précédente) (PRED(valeur précédente)).

La terminaison normale se produit si l'évaluation donne une valeur qui est plus grande (plus petite) que la valeur finale, ou aurait causé l'exception *OVERFLOW*.

3) **énumération de locus**:

dans le cas d'une énumération de locus sans (avec) spécification de **DOWN**, la liste d'énoncés d'action est entamée de façon répétée pour un ensemble de locus spécifiés qui sont les éléments du locus matrice dénoté par le *locus matrice* ou les composantes du locus chaîne désigné par le *locus chaîne*. Si une *expression matrice* ou *une expression chaîne* est spécifiée et n'est pas un locus, un locus contenant la valeur spécifiée sera implicitement créé. La durée de vie du locus créé est l'action faire. Le mode du locus créé est dynamique si la valeur a une classe dynamique. La sémantique est comme si avant chaque exécution de la liste d'énoncés d'action la déclaration d'identité de locus:

DCL <*compteur de boucle*> <*mode*> **LOC** := <*objet composite*> *(<indice>)*;

était rencontrée, où *mode* est le mode élément du mode locus matrice, où *&nom(1)* tel que *&nom* est un nom de **synmode** virtuel **synonyme** du mode locus chaîne, s'il s'agit d'un mode chaîne **fixe**, sinon du mode **composante**, et où *l'indice* est réglé initialement sur la **borne inférieure** (**borne supérieure**) du mode locus et *l'indice* précédent chaque exécution subséquente de la liste d'énoncés d'action est réglé sur *SUCC (indice) (PRED (indice))*. La liste d'énoncés d'action ne sera pas exécutée si la **longueur effective** du *locus chaîne* = 0. L'action faire se termine (terminaison normale) si *l'indice* qui suit immédiatement une exécution de la liste d'énoncés d'action est égal à la **borne supérieure** (**borne inférieure**) du mode *locus*. Quand l'action faire est terminée, *l'objet composite* n'est évalué qu'une seule fois.

propriétés statiques: un compteur de boucle a une chaîne de noms rattachée qui est la chaîne de noms de son *occurence de définition*.

énumération de valeur: le nom défini par le *compteur de boucle* est un nom **d'énumération de valeur**.

énumération par pas: la classe du nom défini par un *compteur de boucle* est la **classe résultante** des classes de *valeur initiale*, *valeur de pas*, si présente, et *valeur finale*.

énumération par intervalle: la classe du nom défini par le *compteur de boucle* est la M-classe par valeur, où M est le *nom de mode discret*.

énumération ensembliste: la classe du nom défini par le *compteur de boucle* est la M-classe par valeur, où M est le mode **primitif** du mode *expression ensembliste* (**forte**).

énumération de locus: le nom défini par le *compteur de boucle* est un nom **d'énumération de locus**. Son mode est le mode **élément** du mode *locus matrice* ou *expression matrice* ou le mode chaîne *&nom(1)*, où *&nom* est un nom de **synmode** virtuel **synonyme** du mode *locus chaîne* ou du *mode* **racine** de *l'expression chaîne*.

Un nom **d'énumération de locus** est **référençable** si l'implantation d'élément du mode *locus matrice* est **NOPACK**.

conditions statiques: les classes de *valeur initiale*, *valeur finale*, et *valeur de pas*, si présentes, doivent être deux à deux **compatibles**.

Le mode **racine** de la classe d'un *compteur de boucle* dans une *énumération de valeur* ne doit pas être un mode ensemble avec **numéros**.

Si le mode **racine** de la classe d'un *compteur de boucle* est un mode entier, il faut qu'il y ait un mode entier **prédéfini** qui contient toutes les valeurs données par *valeur initiale*, *valeur finale* et *valeur de pas*, s'il y en a.

conditions dynamiques: une exception *RANGEFAIL* est causée si la valeur donnée par *valeur de pas* n'est pas supérieure à 0. Cette exception est causée hors du bloc de l'action faire.

exemples:

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 Commande tandis

syntaxe:

<commande tandis> ::= *(1)*
WHILE *<expression booléenne>* *(1.1)*

sémantique: l'expression booléenne est évaluée juste avant d'entamer la liste d'énoncés d'action (après l'évaluation de la commande pour, si présente). Si elle donne *TRUE*, la liste d'énoncés d'action est entamée, sinon l'action faire est terminée.

exemple:

7.35 **WHILE** *n >= 1* *(1.1)*

6.5.4 Partie avec

syntaxe:

<partie avec> ::= *(1)*
WITH *<commande avec>* {, *<commande avec>* }* *(1.1)*

<commande avec> ::= *(2)*
<locus structure> *(2.1)*
| *<valeur primitive structure>* *(2.2)*

NOTE – Si la *commande avec* est un *locus de structure*, on résout l'ambiguïté syntaxique en interprétant la *commande avec* comme un *locus* et non comme une *valeur primitive*.

sémantique: les noms de champ (**visibles**) du mode locus structure ou des valeurs structure spécifiés dans chaque *commande avec* sont rendus disponibles comme accès direct aux champs.

Les règles de visibilité se présentent comme si une occurence de définition de nom de champ était introduite pour chaque nom **de champ** attaché au mode locus ou de la valeur primitive et ayant la même chaîne de nom que le nom de champ.

Si un *locus structure* est spécifié, des noms d'accès ayant la même chaîne de noms que les noms de champ du mode *locus structure* sont implicitement définis, dénotant les sous-locus du locus structure.

Si une *valeur primitive structure* est spécifiée, des noms de valeur ayant la même chaîne de noms que les noms de champ du mode *valeur primitive structure* (**forte**) sont implicitement définis, dénotant les sous-valeurs de la valeur structure.

Quand on entame l'action faire, les locus structure et/ou les valeurs structure spécifiés ne sont évalués qu'une fois en entamant l'action faire, dans un ordre quelconque.

propriétés statiques: l'occurence de définition (virtuelle) introduite pour un nom de **champ** a la même chaîne de noms que *l'occurence de définition de noms de champ* de ce nom de **champ**.

Si une *valeur primitive structure* est spécifiée, une occurence de définition (virtuelle) dans une *partie avec* définit un nom de **valeur faire-avec**. Sa classe est la M-classe par valeur, où M est le mode de ce nom de **champ** du mode structure de la *valeur primitive structure*, qui est rendue disponible comme **valeur** de nom **faire-avec**.

Si un *locus structure* est spécifié, une occurence de définition (virtuelle) dans une *partie avec* définit un nom de **locus faire-avec**. Son mode est le mode de ce nom de **champ** du mode *locus structure*, qui est rendu disponible comme nom de **locus faire-avec**. Un nom de **locus faire-avec** est **référençable** si l'implantation de champ du nom de **champ** associé est **NOPACK**.

exemple:

15.58 **WITH** *each* *(1.1)*

6.6 Action sortir

syntaxe:

<action sortir> ::= *(1)*
EXIT *<nom d'étiquette>* *(1.1)*

sémantique: une action sortir est employée pour quitter un énoncé d'action parenthésé ou un module. L'exécution reprend immédiatement après l'énoncé d'action parenthésé englobant du plus près ou le module étiqueté par le *nom d'étiquette*.

conditions statiques: l'*action sortir* doit résider à l'intérieur de l'énoncé d'action parenthésé ou du module dont *l'occurence de définition* qui précède a la même chaîne de nom que le *nom d'étiquette*.

Si l'*action sortir* se trouve à l'intérieur d'une définition de procédure ou d'une définition de processus, l'énoncé d'action parenthésé ou le module dont on sort doivent aussi résider à l'intérieur de la même définition de procédure ou de processus (c'est-à-dire que l'action sortir ne peut s'employer pour quitter des procédures ou des processus).

Aucun *filet* ne peut terminer une *action sortir*.

exemple:

15.62 **EXIT** *find_counter* *(1.1)*

6.7 Action appeler

syntaxe:

<action appeler> ::= *(1)*
<appel de procédure> *(1.1)*
| *<appel de routine prédéfinie>* *(1.2)*
| *<appel de procédure de composante moreta>* *(1.3)*

<appel de procédure> ::= *(2)*
{ *<nom de procédure>* | *<valeur primitive procédure>* }
([*<liste de paramètres effectifs>*]) *(2.1)*

<liste de paramètres effectifs> ::= *(3)*
<paramètre effectif> { , *<paramètre effectif>* }* *(3.1)*

<paramètre effectif> ::= *(4)*
<valeur> *(4.1)*
| *<locus>* *(4.2)*

<appel de routine prédéfinie> ::= *(5)*
<nom de routine prédéfinie> ([*<liste de paramètres de routine prédéfinie>*]) *(5.1)*

<liste de paramètres de routine prédéfinie> ::= *(6)*
<paramètre de routine prédéfinie> { , *<paramètre de routine prédéfinie>*}* *(6.1)*

<paramètre de routine prédéfinie> ::= *(7)*
<valeur> *(7.1)*
| *<locus>* *(7.2)*
| *<nom non réservé>*[(*<liste de paramètres de routine prédéfinie>*)] *(7.3)*

<appel de procédure de composante moreta> ::= *(8)*
<locus de moreta> **.** *<appel de procédure de composante moreta>* [*<priorité>*] *(8.1)*
| *<valeur primitive de locus moreta à référence liée>* –> **.**
<appel de procédure de composante moreta> [*<priorité>*] *(8.2)*
| *<appel de procédure de composante moreta>* [*<priorité>*] *(8.3)*

NOTE – Si le *paramètre effectif* ou le *paramètre de routine prédéfinie* est un *locus*, on élimine l'ambiguïté syntaxique en l'interprétant comme *un locus* et non comme une *valeur*.

syntaxe dérivée: une action appeler **P(…)** d'une *procédure* **P** *de composante moreta* est une syntaxe dérivée de **SELF.P(…).**

sémantique: une action appeler cause l'appel d'une procédure, d'une routine prédéfinie ou d'une procédure de composante moreta. Un appel de procédure cause un appel de la procédure **générale** indiquée par la valeur donnée par la *valeur primitive procédure* ou la procédure indiquée par le *nom de procédure*. Une appel de procédure de *composante moreta* **L.name(…)** produit cet appel, qui est identifié par le nom dans le mode de **L**. **L** est transmis à l'action en tant que paramètre de locus initial. Les valeurs et les locus effectifs spécifiés dans la liste des paramètres effectifs sont transmis à la procédure.

Un *appel de routine prédéfinie* est soit un *appel de routine prédéfinie CHILL*, soit un *appel de routine prédéfinie* par l'implémentation (voir 6.20 et 13.1 respectivement).

Une valeur, un locus ou tout nom défini de programme qui n'est pas une chaîne de nom simple **réservée** peut être passé comme un *paramètre de routine prédéfinie*. L'appel de routine prédéfinie peut envoyer une valeur ou un locus.

Une routine prédéfinie peut être générique, c'est-à-dire que sa classe (si c'est un appel de routine prédéfinie rendant **valeur**) ou son mode (si c'est un appel de routine prédéfinie rendant **locus**) peut dépendre non seulement du *nom de routine prédéfinie* mais aussi des propriétés statiques des paramètres effectifs passés et du contexte statique de l'appel.

Un appel de procédure de composante moreta a toujours une structure "locus . appel de procédure". Cela est caractérisé par l'expression "l'appel de procédure est appliqué au locus".

Les étapes suivantes sont exécutées pour un appel de procédure de composante moreta:

a) *la procédure appelée est appliquée à un locus de mode module*:
   1) évaluation des paramètres effectifs
   2) vérification de la précondition
   3) vérification de l'ensemble de l'invariant
   4) exécution du corps de la procédure
   5) vérification de l'ensemble de l'invariant
   6) vérification de l'ensemble de la postcondition
   7) retour au point d'appel

b) *la procédure appelée est appliquée à un RL de locus de mode région*:
   1) évaluation des paramètres effectifs
   2) attente de la libération RL, puis verrouillage de celui-ci
   3) vérification de la précondition
   4) vérification de l'ensemble de l'invariant
   5) exécution du corps de la procédure
   6) vérification de l'ensemble de l'invariant
   7) vérification de la postcondition
   8) libération de RL
   9) retour au point d'appel

c) *la procédure appelée est appliquée à un locus TL de mode région:*

   L'appelant exécute les opérations suivantes:
   1) évaluation des paramètres effectifs
   2) envoi au locus TL de la procédure d'identification, des paramètres effectifs et de la priorité
   3) passage à l'étape suivante

   Le locus TL exécute les opérations suivantes:
   1) réception de la procédure d'identification et des paramètres effectifs compte tenu de la priorité
   2) vérification de la précondition
   3) vérification de l'ensemble de l'invariant
   4) exécution du corps de la procédure
   5) vérification de l'ensemble de l'invariant
   6) vérification de l'ensemble de la postcondition

propriétés statiques: un *appel de procédure* a les propriétés suivantes: il a une liste de **specs de paramètre**, éventuellement une **spec de résultat**, un ensemble éventuellement vide de noms d'exception, une **généralité**, une **récursivité**, et il peut être **intrarégional** (cette dernière propriété n'est possible que pour un *nom de procédure*, voir 11.2.2). Ces propriétés sont héritées du *nom de procédure*, *du nom de procédure de composante moreta* ou d'un mode **compatible** avec la classe de la *valeur primitive procédure* (dans le dernier cas, la généralité est toujours **générale**).

Un *appel de procédure* qui a une **spec de résultat**; est un *appel de procédure rendant locus* si et seulement si **LOC** est spécifié dans la **spec de résultat**; sinon c'est un *appel de procédure rendant valeur*.

Un *nom de routine prédéfinie* est un nom défini par le CHILL ou par l'implémentation qui est considéré comme défini dans le domaine de la définition du processus imaginaire le plus externe ou dans un contexte quelconque (voir 10.8).

Un *appel de routine prédéfinie* est un *appel de routine prédéfinie* rendant **locus** s'il livre un locus; c'est un *appel de routine prédéfinie* rendant **valeur** s'il livre une valeur.

conditions statiques: une *priorité* ne peut être utilisée que dans l'appel d'une procédure appliquée à un locus tâche.

Le nombre d'occurrences de *paramètre effectif* dans *l'appel de procédure* doit être le même que le nombre de ses specs de paramètre. Les règles de compatibilité pour un *paramètre effectif* et une spec de paramètre correspondante (par position) de *l'appel de procédure* sont:

- si la spec de paramètre a l'attribut **IN** (ce qui est le cas par défaut), le *paramètre effectif* doit être une *valeur* dont la classe doit être **compatible** avec le mode dans la spec de paramètre correspondante. Ce dernier mode ne doit pas avoir la **propriété de non-valeur**. Le *paramètre effectif* est une *valeur* qui doit être **régionalement sûre** pour *l'appel de procédure*;
- si la spec de paramètre a l'attribut **INOUT** ou **OUT**, le *paramètre effectif* doit être un *locus*, dont le mode doit être **compatible** avec la M-classe par valeur, où M est le mode dans la spec de paramètre correspondante. Le mode du *locus* (effectif) doit être statique et ne doit avoir ni la **propriété de protection** ni la **propriété de non-valeur**. Le *paramètre effectif* est un *locus*. Il peut être considéré comme une *valeur* qui doit être **régionalement sûre** pour *l'appel de procédure*;
- si la spec de paramètre a l'attribut **INOUT**, le mode dans la spec de paramètre doit être **compatible** avec la M-classe par valeur, où M est le mode du *locus*;
- si la spec de paramètre a l'attribut **LOC** spécifié sans **DYNAMIC**, le *paramètre effectif* doit être un *locus* qui est à la fois **référençable** et tel que le mode dans la spec de paramètre soit **compatible en lecture** avec le mode de ce *locus* (effectif), ou bien le *paramètre effectif* doit être une *valeur* qui n'est pas un *locus* mais dont la classe est **compatible** avec le mode dans la spec de paramètre. Si le mode du paramètre formel est un mode moreta, le nom de mode du paramètre formel et celui du paramètre effectif doivent être synonymes. Si le mode du paramètre formel est de la forme "REF MM", où MM est un mode moreta, le mode du paramètre formel et celui du mode effectif doivent être analogues;
- si la spec de paramètre a l'attribut **LOC**, **DYNAMIC** étant spécifié, le *paramètre effectif* doit être un *locus* qui est à la fois **référençable** et tel que le mode dans la spec de paramètre soit **compatible en lecture dynamique** avec le mode de ce *locus* (effectif), ou bien le *paramètre effectif* doit être une *valeur* qui n'est pas un *locus* mais dont la classe est **compatible** avec une version paramétrée de ce mode;
- si la spec de paramètre a l'attribut **LOC**, alors:
  - si le *paramètre effectif* est un *locus*, il doit avoir la même **régionalité** que *l'appel de procédure*;
  - si le *paramètre effectif* est une *valeur*, il doit être **régionalement sûr** pour *l'appel de procédure*.

conditions dynamiques: un *appel de procédure* peut causer toute exception de l'ensemble de noms d'exception qui lui est attaché. Un *appel de procédure* cause l'exception *EMPTY* si la *valeur primitive procédure* donne *NULL*; il cause l'exception *SPACEFAIL* si on ne peut satisfaire les besoins de mémoire. Si la **récursivité** de la procédure est **non récursive**, la procédure ne doit pas s'appeler directement ou indirectement.

Le passage de paramètres peut causer les exceptions suivantes:

- si la spec de paramètre a l'attribut **IN** ou **INOUT**, les conditions d'affectation de la valeur (effective), en tenant compte du mode de la spec de paramètre doivent être respectées à l'endroit de l'appel (voir 6.2) et les exceptions possibles sont causées avant que la procédure ne soit appelée;
- si la spec de paramètre a l'attribut **INOUT** ou **OUT**, les conditions d'affectation de la valeur locale du paramètre formel, en tenant compte du mode du locus (effectif) doivent être respectées au point de retour (voir 6.2) et les exceptions possibles sont causées après le retour de la procédure;
- si la spec de paramètre a l'attribut **LOC** et que le *paramètre effectif* est une *valeur* qui n'est pas un *locus*, les conditions d'affectation de la *valeur* (effective) en tenant compte du mode de la spec de paramètre doivent être respectées à l'endroit de l'appel et les exceptions possibles sont causées avant que la procédure ne soit appelée (voir 6.2).

La vérification d'une assertion peut donner lieux aux exceptions suivantes:

- l'exception *PREFAIL* si la précondition juge l'assertion *FALSE*. La recherche d'un filet approprié commence à la fin du corps de la procédure et se poursuit comme indiqué au 8.3*;*
- l'exception *POSTFAIL* si la postcondition juge l'assertion *FALSE*. La recherche d'un filet approprié commence à la fin du corps de la procédure et se poursuit comme indiqué au 8.3*;*
- l'exception *INVFAIL* si l'invariant juge l'assertion *FALSE*. La recherche d'un filet approprié commence à la fin du mode moreta correspondant et se poursuit comme indiqué au 8.3.

La *valeur primitive procédure* ne doit pas donner une procédure définie dans une définition de processus dont l'activation n'est pas la même que l'activation du processus exécutant l'appel de procédure (autre que le processus imaginaire le plus externe) et la durée de vie de la procédure désignée ne doit pas être terminée.

Si un appel est appliqué à un locus TL de tâche, il y a lieu de ne pas terminer ce locus TL.

exemple:

4.18 op(a,b,d,order–1) *(1.1)*

6.8 Action résulter et action revenir

syntaxe:

<action revenir> ::= *(1)*
RETURN [*<résultat>*] *(1.1)*

<action résulter> ::= *(2)*
RESULT *<résultat>* *(2.1)*

<résultat> ::= *(3)*
<valeur> *(3.1)*
| *<locus>* *(3.2)*

syntaxe dérivée: l'*action revenir* avec *résultat* est dérivée de **DO RESULT** *<résultat>*; **RETURN**; **OD**.

sémantique: une action résulter sert à établir le résultat devant être rendu par un appel de procédure. Ce résultat peut être un locus ou une valeur. Une action revenir cause le retour de l'invocation de la procédure dans la définition de laquelle elle est placée. Si la procédure retourne un résultat, ce résultat est déterminé par la dernière action résulter exécutée. Si aucune action résulter n'a été exécutée, l'appel de procédure donne un locus **indéfini** ou une valeur **indéfinie**.

propriétés statiques: a une *action résulter* et à une *action revenir* est attaché un nom de **procédure**, qui est le nom de la définition de procédure qui les englobe du plus près.

conditions statiques: une *action revenir* et une *action résulter* doivent être textuellement englobées par une définition de procédure. Une *action résulter* ne peut être spécifiée que si son nom de **procédure** a une **spec de résultat**.

Un *filet* ne doit pas terminer une *action revenir* (sans *résultat*).

Si **LOC** (**LOC DYNAMIC**) est spécifié dans la **spec de résultat** du nom de **procédure** de l'*action résulter*, le *résultat* doit être un *locus*, tel que le mode dans la **spec de résultat** soit **compatible en lecture** (**compatible en lecture dynamique**) avec le mode du *locus*. Le *locus* doit être **référençable** si **NONREF** n'est pas spécifié dans **la spec de résultat**. Le *résultat* est un *locus* qui doit avoir la même **régionalité** que le nom **de procédure** attaché à l'*action résulter*.

Si **LOC** n'est pas spécifié dans la **spec de résultat** du nom **de procédure** de l'*action résulter*, le *résultat* doit être une *valeur* dont la classe est **compatible** avec le mode dans la **spec de résultat**. Le *résultat* est une *valeur* qui doit être **régionalement sûre** pour le nom **de procédure** attaché à l'*action résulter*.

conditions dynamiques: si **LOC** n'est pas spécifié dans la **spec de résultat** du nom de **procédure**, les conditions d'affectation de la *valeur* dans l'*action résulter* en tenant compte du mode dans la **spec de résultat** de son nom de **procédure** doivent être respectées.

exemples:

4.21 **RETURN** *(1.1)*

1.6 **RESULT** *i+j* *(2.1)*

5.19 *c* *(3.1)*

6.9 Action aller

syntaxe:

<action aller> ::= *(1)*
GOTO *<nom d'étiquette>* *(1.1)*

sémantique: l'action aller cause un déplacement du point d'exécution. L'exécution continue à l'énoncé d'action étiqueté par le *nom d'étiquette*.

conditions statiques: si l'*action aller* se trouve à l'intérieur d'une définition de procédure ou d'une définition de processus, l'étiquette indiquée par le *nom d'étiquette* doit aussi être définie à l'intérieur de la définition (c'est-à-dire qu'il n'est pas possible de sauter hors d'une invocation de procédure ou de processus).

Un *filet* ne doit pas terminer une *action aller*.

6.10 Action affirmer

syntaxe:

<action affirmer> ::= *(1)*
ASSERT *<expression booléenne>* *(1.1)*

sémantique: une action affirmer fournit une manière de tester une condition.

conditions dynamiques: l'exception *ASSERTFAIL* est causée si l'*expression booléenne* donne *FALSE*.

exemple:

4.7 **ASSERT** *b>0* **AND** *c>0* **AND** *order>0* *(1.1)*

6.11 Action vide

syntaxe:

<action vide> ::= *(1)*
<vide> *(1.1)*

<vide> ::= *(2)*

sémantique: une action vide ne fait rien.

conditions statiques: un *filet* ne doit pas terminer une *action vide*.

6.12 Action induire

syntaxe:

<action induire> ::= *(1)*
CAUSE *<nom d'exception>* *(1.1)*

sémantique: une action induire cause l'exception dont le nom est indiqué par *nom d'exception.*

conditions statiques: un *filet* ne doit pas terminer une *action induire*.

exemple:

4.9 **CAUSE** *wrong_input* *(1.1)*

6.13 Action démarrer

syntaxe:

<action démarrer> ::= *(1)*
<expression démarrer> *(1.1)*

sémantique: une action démarrer évalue l'expression démarrer (voir 5.2.15), éventuellement sans employer la valeur instance donnée par cette expression.

exemple:

14.45 **START** *call_distributor ( )* *(1.1)*

6.14 Action arrêter

syntaxe:

<action arrêter> ::= *(1)*
STOP *(1.1)*

sémantique: l'action arrêter termine le processus qui exécute l'action arrêter (voir 11.1).

conditions statiques: un *filet* ne doit pas terminer une *action arrêter*.

6.15 Action continuer

syntaxe:

<action continuer> ::= *(1)*
CONTINUE *<locus événement>* *(1.1)*

sémantique: une action continuer évalue le *locus événement*.

Si un ensemble non vide de processus mis en attente est attaché au locus événement, l'un de ces processus dont la priorité est la plus élevée sera réactivé. S'il y a plusieurs de ces processus, on en choisira un tel que défini dans l'implémentation. S'il n'y a aucun de ces processus, l'action continuer n'aura pas d'autres effets.

Si un processus est réactivé, il est enlevé de tous les ensembles de processus mis en attente dont il faisait partie.

exemple:

13.25 **CONTINUE** *resource_freed* *(1.1)*

6.16 Action mettre en attente

syntaxe:

<action mettre en attente> ::= *(1)*
DELAY *<locus événement>* [*<priorité>*] *(1.1)*

<priorité> ::= *(2)*
PRIORITY *<expression de littéral entier>* *(2.1)*

sémantique: une action mise en attente évalue le *locus événement*.

Ensuite, une exception *DELAYFAIL* se produit (voir ci-dessous) ou le processus en exécution est mis en attente.

Si le processus en exécution est mis en attente, il devient membre avec une certaine priorité, de l'ensemble des processus mis en attentes associés au locus événement spécifié. La priorité est celle qui est spécifiée, le cas échéant, sinon elle est égale à 0 (c'est-à-dire la plus faible).

propriétés dynamiques: un processus qui exécute une action mettre en attente devient **temporisable** quand il atteint le point d'exécution où il peut être mis en attente. Il cesse d'être **temporisable** quand il quitte ce point.

conditions statiques: l'*expression de littéral entier* ne peut pas donner une valeur négative.

conditions dynamiques: l'exception *DELAYFAIL* est causée si le mode *locus événement* a une **longueur d'événement** associée qui est égale au nombre de processus déjà mis en attente dans ce locus événement.

La durée de vie du *locus événement* ne doit pas se terminer pendant que le processus en exécution est mis en attente sur lui.

exemple:

13.18 **DELAY** *resource_freed* *(1.1)*

6.17 Action attente

syntaxe:

<action attente> ::= *(1)*
DELAY CASE [**SET** *<locus instance>* [*<priorité>*] ; | *<priorité>* ;]
{ *<alternative>* }$^{+}$
ESAC *(1.1)*

<alternative> ::= *(2)*
(*<liste d'événements>*) : *<liste d'énoncés d'action>* *(2.1)*

<liste d'événements> ::= *(3)*
<locus événement> { , *<locus événement>* }* *(3.1)*

sémantique: une action attente évalue, dans un ordre non spécifié ou éventuellement mélangé, un *locus instance*, s'il y en a un, et tous les *locus événement* spécifiés dans *une alternative*.

Ensuite, une exception *DELAYFAIL* se produit (voir ci-dessous) ou le processus exécutant est en attente.

Si le processus exécutant est en attente, il devient membre, avec une certaine priorité, de l'ensemble des processus en attente associés à chacun des locus événement spécifiés. La priorité est celle qui est spécifiée, le cas échéant, sinon elle est de 0 (priorité la plus faible).

Si le processus en attente est réactivé par un autre processus qui exécute une action continuer sur un locus événement, la *liste d'énoncés d'action* correspondante est entamée. Si plusieurs *alternatives* spécifient le même locus événement, le choix entre ces choix n'est pas spécifié. Avant d'entamer, si un *locus instance* est spécifié, la valeur instance identifiant le processus qui a exécuté l'action continuer est mémorisée dans ce locus.

propriétés dynamiques: un processus qui exécute une action attente devient **temporisable** quand il atteint le point d'exécution où il peut être en attente. Il cesse d'être **temporisable** quand il quitte ce point.

conditions statiques: le mode du *locus instance* ne doit pas avoir la **propriété de protection**. L'*expression de littéral entier* dans *priorité* ne doit pas donner une valeur négative.

conditions dynamiques: l'exception *DELAYFAIL* est causée si un *locus événement* quelconque a un mode avec une **longueur d'événement** associée égale au nombre de processus déjà en attente sur le locus événement.

La durée de vie d'aucun des *locus événement* donnés ne doit se terminer pendant que le processus exécutant l'action attente est en attente sur lui.

L'exception *SPACEFAIL* est causée lorsque les besoins de mémoire ne peuvent pas être satisfaits.

exemple:

```
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 Action envoyer

6.18.1 Généralités

syntaxe:

```
<action envoyer> ::=                                            (1)
        <action envoyer signal>                                 (1.1)
    |   <action envoyer tampon>                                 (1.2)
```

sémantique: une action envoyer initie le transfert d'information de synchronisation à partir d'un fil d'exécution envoyant. La sémantique détaillée dépend de la question de savoir si l'objet de synchronisation est un signal ou un tampon.

6.18.2 Action envoyer signal

syntaxe:

```
<action envoyer signal> ::=                                     (1)
        SEND <nom de signal> [ ( <valeur> { , <valeur> } *) ]
        TO <valeur primitive instance> [ <priorité> ]           (1.1)
```

sémantique: une action envoyer signal évalue dans un ordre non spécifié et éventuellement mélangé, la liste de *valeurs*, s'il y en a une, et la *valeur primitive instance*.

Le signal spécifié par *nom de signal* est composé pour transmission à partir des valeurs spécifiées et d'une priorité. La priorité est celle qui est spécifiée, le cas échéant, sinon elle est de 0 (priorité la plus faible).

Si le nom de **signal** a un nom de **processus** qui lui est attaché, seuls les processus ayant ce nom peuvent recevoir le signal; si une *valeur primitive instance* est spécifiée, seul le processus identifié par *valeur primitive instance* peut recevoir le signal.

Si le signal a un ensemble non vide de processus en attente qui lui est attaché, dans lequel un ou plusieurs processus peuvent recevoir le signal, un de ces derniers sera réactivé. S'il y a plusieurs de ces processus, on en choisira un défini par l'implémentation. S'il n'y a pas de tels processus, le signal est mis en instance.

Si un processus est réactivé, il ne fait plus partie de tous les ensembles de processus en attente auxquels il appartenait.

conditions statiques: le nombre d'occurrences de *valeur* doit être égal au nombre de modes *nom de signal*. La classe de chaque *valeur* doit être **compatible** avec le mode correspondant du *nom de signal*. Aucune occurrence de *valeur* ne peut être **intrarégionale** (voir 11.2.2). L'*expression de littéral entier* dans *priorité* ne doit pas donner une valeur négative.

conditions dynamiques: les conditions d'affectation de chaque *valeur* en tenant compte du mode correspondant du *nom de signal* doivent être respectées.

L'exception *EMPTY* est causée si la *valeur primitive instance* donne *NULL*.

La durée de vie du processus indiqué par la valeur donnée par la *valeur primitive instance* ne doit pas être terminée au point d'exécution de l'action envoyer signal.

L'exception *SENDFAIL* est causée si le *nom de signal* a un nom de **processus** qui n'est pas le nom du processus indiqué par la valeur donnée par la *valeur primitive instance*.

exemples:

15.78 **SEND** *ready* **TO** *received_user* *(1.1)*

15.86 **SEND** *readout(count)* **TO** *user* *(1.1)*

6.18.3 Action envoyer tampon

syntaxe:

<action envoyer tampon> ::= *(1)*
SEND <*locus tampon*> (<*valeur*>) [<*priorité*>] *(1.1)*

sémantique: une action envoyer tampon évalue le *locus tampon* et la *valeur* dans un ordre quelconque.

Si le locus tampon a un ensemble non vide de processus qui lui est attaché, l'un de ces derniers sera réactivé. S'il y a plusieurs de ces processus, on en choisira un défini par l'implémentation. S'il n'y a pas de tels processus et si la capacité du locus tampon est dépassée, le processus exécutant est en attente avec une certaine priorité. Sinon, la valeur est stockée avec une certaine priorité. La priorité est celle qui est spécifiée, le cas échéant, sinon elle est de 0 (priorité la plus faible). La capacité du tampon est dépassée si le *locus tampon* a un mode avec une **longueur de tampon** qui lui est attachée égale au nombre de valeurs déjà stockées dans le locus tampon.

Si le processus exécutant est en attente, il devient membre de l'ensemble des processus envoyants en attente associé au locus tampon. Si un processus est réactivé, il ne fait plus partie de tous les ensembles de processus en attente auxquels il appartenait.

propriétés dynamiques: un processus exécutant une action envoyer tampon devient **temporisable** quand il atteint le point d'exécution où il peut être en attente. Il cesse d'être **temporisable** lorsqu'il quitte ce point.

conditions statiques: la classe de *valeur* doit être **compatible** avec le mode **élément tampon** du mode du *locus tampon*. La *valeur* ne doit pas être **intrarégionale** (voir 11.2.2). L'*expression de littéral entier* dans *priorité* ne doit pas donner une valeur négative.

conditions dynamiques: les conditions d'affectation de la *valeur* en tenant compte du mode **élément tampon** du mode du *locus tampon* doivent être respectées. Les exceptions possibles sont causées avant que le processus ne soit en attente.

La durée de vie du locus *tampon* donné ne doit pas se terminer pendant que le processus qui exécute l'action envoyer tampon est en attente sur lui.

exemple:

16.123 **SEND** *user–>* ([*ready, –>counter_buffer*]) ; *(1.1)*

6.19 Action recevoir et choisir

6.19.1 Généralités

syntaxe:

<action recevoir avec cas> ::= *(1)*
<action recevoir signal avec cas> *(1.1)*
| *<action recevoir tampon avec cas>* *(1.2)*

sémantique: une action recevoir avec cas reçoit l'information de synchronisation qui est transmise par l'action envoyer. La sémantique détaillée dépend de l'objet de synchronisation employé, qui est soit un signal soit un tampon. Entamer une action recevoir avec cas ne se traduit pas nécessairement par la mise en attente du fil d'exécution (voir l'article 11 pour plus de détails).

6.19.2 Action recevoir signal avec cas

syntaxe:

<action recevoir signal avec cas> ::= *(1)*
RECEIVE CASE [**SET** *<locus instance>* ;]
{ *<signal reçu possible>* }$^{+}$
[**ELSE** *<liste d'énoncés d'action>*] **ESAC** *(1.1)*
| **RECEIVE** [**SET** *<locus instance>*]
(*<nom de signal>* [**IN** *<liste de locus>*]) *(1.2)*

<liste de locus> ::= *(2)*
<locus> { , *<locus>* }* *(2.1)*

<signal reçu possible> ::= *(3)*
(*<nom de signal>* [**IN** *<liste d'occurence de définitions>*])*:*
<liste d'énoncés d'action> *(3.1)*

syntaxe dérivée: la notation (1.2) est la syntaxe dérivée pour
RECEIVE CASE [**SET** *<locus instance>* ;]
(*<nom de signal>* [**IN** *<&nom>*$_1$, …, *<&nom>*$_n$]):

<locus>$_1$:= *<&nom>*$_1$; … *<locus>*$_n$:= *<&nom>*$_n$; **ESAC,**

où *<&nom>*$_1$, …, *<&nom>*$_n$ sont des noms de **valeur reçue** introduits virtuellement, et

<locus>$_1$, … , *<locus>*$_n$ sont les *locus* de la *liste de locus.*

sémantique: une action recevoir signal avec cas évalue le *locus instance*, s'il y en a un.

Ensuite, le processus exécutant reçoit (immédiatement) un signal ou, si **ELSE** est spécifié, entame *la liste d'énoncés d'action* correspondante, sinon il est en attente. Le processus exécutant reçoit immédiatement un signal si un *nom de signal* spécifié dans un *signal reçu possible* est en instance et peut être reçu par le processus. Si plus d'un signal peut être reçu, celui qui a la priorité la plus élevée sera choisi selon l'implémentation.

Si le processus exécutant est en attente, il devient membre de l'ensemble des processus en attente attachés à chacun des signaux spécifiés. Si le processus en attente est réactivé par un autre processus exécutant une action envoyer signal, il reçoit un signal.

Si le processus exécutant reçoit un signal, la *liste d'énoncés d'action* correspondante est entamée. Avant d'entamer, si un *locus instance* est spécifié, la valeur instance identifiant le processus qui a envoyé le signal reçu est stockée dans ce locus. Si le nom de **signal** du signal reçu a une liste de modes qui lui est attachée, une liste de noms de **valeur reçues** est spécifiée; le signal transporte une liste de valeurs, et les noms de **valeurs reçues** dénotent leurs valeurs correspondantes dans la *liste d'énoncés d'action* entamée.

propriétés statiques: une *occurence de définition* de la *liste d'occurences de définitions* d'un *signal reçu possible* définit un nom de **valeur reçue**. Sa classe est la M-classe par valeur, où M est le mode correspondant dans la liste de modes attachée au *nom de signal* qui le précède.

propriétés dynamiques: un processus qui exécute une action recevoir signal avec cas devient **temporisable** quand il atteint le point d'exécution où il peut être en attente. Il cesse d'être **temporisable** quand il quitte ce point.

conditions statiques: le mode du *locus instance* ne doit pas avoir la **propriété de protection**.

Toutes les occurrences de *nom de signal* doivent être différentes.

Le **IN** facultatif et la *liste d'occurences de définitions* dans le *signal reçu possible* ne doivent être spécifiés que si le *nom de signal* a un ensemble de modes non vide. Le nombre de noms dans la *liste d'occurences de définitions* doit être égal au nombre de modes du *nom de signal*.

Les conditions d'attribution des valeurs données par *&name*$_1$, …, *&name*$_n$ relativement aux modes de *location*$_1$, …, *location*$_n$ s'appliquent.

conditions dynamiques: l'exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemple:

```
15.83   RECEIVE CASE
            (advance): count + := 1;
            (terminate):
                SEND readout(count) TO user;
                EXIT work_loop;
        ESAC                                                    (1.1)
```

6.19.3 Action recevoir tampon avec cas

syntaxe:

```
<action recevoir tampon avec cas> ::=                                   (1)
        RECEIVE CASE [ SET <locus instance> ; ]
        { <alternative réception de tampon> }+
        [ ELSE <liste d'énoncés d'action> ]
        ESAC                                                            (1.1)
    | RECEIVE [ SET <locus instance> ]
        ( <locus tampon> IN <locus> )                                   (1.2)

<alternative réception de tampon> ::=                                   (2)
        ( <locus tampon> IN <occurence de définition> ) : <liste d'énoncés d'action>   (2.1)
```

syntaxe dérivée: la notation (1.2) est la syntaxe dérivée pour

```
RECEIVE CASE [ SET <locus instance>; ]
( <locus tampon> IN <&nom> ) : <locus> := <&nom>;
```

où <*&nom*> est un nom de **valeur reçue** introduit virtuellement.

sémantique: une action recevoir tampon avec cas évalue, dans un ordre non spécifié, le *locus instance*, s'il est présent, et tous les *locus tampon* spécifiés dans une *alternative réception de tampon*.

Ensuite, le processus exécutant reçoit (immédiatement) une valeur ou, si **ELSE** est spécifié, entame la *liste d'énoncés d'action* correspondante, sinon il est en attente. Le processus exécutant reçoit immédiatement une valeur s'il en a une qui est stockée dans, ou un processus envoyer en attente, l'un des locus tampon spécifié. Si plus d'une valeur peut être reçue, celle qui a la priorité la plus élevée sera choisie selon l'implémentation.

Si le processus exécutant est en attente, il devient membre de l'ensemble des processus en attente attachés à chacun des locus tampon spécifiés. Si le processus en attente est réactivé par un autre processus exécutant une action envoyer tampon, il reçoit une valeur.

Si le processus exécutant reçoit une valeur, la *liste d'énoncés d'action* correspondante est entamée. Si plusieurs *alternatives réception de tampon* spécifient le même locus tampon, le choix entre ces choix n'est pas spécifié. Avant d'entamer, si un *locus instance* est spécifié, la valeur instance identifiant le processus qui a envoyé la valeur reçue est stockée dans ce locus. Le nom **de valeur reçue** spécifié dénote la valeur reçue dans la *liste d'énoncés d'action* entamée.

Un autre processus est réactivé si le processus exécutant reçoit une valeur provenant d'un locus tampon, dont l'ensemble de processus envoyant en attente n'est pas vide. Le processus réactivé est celui qui a la priorité la plus élevée, si la valeur reçue a été stockée dans le locus tampon, sinon, c'est celui qui a envoyé la valeur reçue. Dans le premier cas, la valeur que le processus réactivé doit envoyer est mémorisée dans le locus tampon (dont la capacité reste dépassée), et si plusieurs processus peuvent être réactivés, on prendra celui qui a été défini par l'implémentation. Le processus réactivé est enlevé de l'ensemble des processus envoyant en attente attachés au locus tampon.

propriétés statiques: une *occurence de définition* dans une *alternative réception de tampon* définit un nom de **valeur reçue**. Sa classe est la M-classe par valeur, où M est le mode **élément tampon** du mode du *locus tampon* qualifiant l'*alternative réception de tampon*.

propriétés dynamiques: un processus qui exécute une action recevoir tampon avec cas devient **temporisable** quand il atteint le point d'exécution où il peut être en attente. Il cesse d'être **temporisable** quand il quitte ce point.

conditions statiques: le mode de *locus instance* ne doit pas avoir la **propriété de protection**.

Les conditions d'affectation de la valeur que dénote *&nom* en ce qui concerne le mode *locus* s'appliquent.

conditions dynamiques: l'exception *SPACEFAIL* est causée lorsque les besoins de mémoire ne peuvent pas être satisfaits.

La durée de vie d'aucun des *locus tampon* ne doit se terminer pendant que le processus exécutant est en attente sur lui.

6.20 Appels de routine prédéfinie CHILL

syntaxe:

<appel de routine prédéfinie CHILL> ::= *(1)*
<appel de routine prédéfinie simple CHILL> *(1.1)*
| *<appel de routine prédéfinie rendant locus CHILL>* *(1.2)*
| *<appel de routine prédéfinie rendant valeur CHILL>* *(1.3)*

noms prédéfinis: les noms de routine prédéfinie CHILL sont prédéfinis comme des noms **de routine prédéfinie** (voir 6.7).

sémantique: un *appel de routine prédéfinie CHILL* est un *appel de routine prédéfinie simple CHILL* qui ne fournit pas de résultat (voir 6.20.1), ou un *appel de routine prédéfinie rendant locus CHILL* qui donne un locus (voir 6.20.2), ou un *appel de routine prédéfinie rendant valeur CHILL* qui donne une valeur (voir 6.20.3).

propriétés statiques: un *appel de routine prédéfinie CHILL* est un *appel de routine prédéfinie* de **locus** si c'est un *appel de routine prédéfinie rendant locus CHILL*; c'est un *appel de routine prédéfinie* de **valeur** si c'est un *appel de routine prédéfinie rendant valeur CHILL*.

6.20.1 Appels de routine prédéfinie simple CHILL

syntaxe:

<appel de routine prédéfinie simple CHILL> ::= *(1)*
<appel de routine prédéfinie terminer> *(1.1)*
| *<appel de routine prédéfinie simple d'e/s>* *(1.2)*
| *<appel de routine prédéfinie simple de temporisation>* *(1.3)*

sémantique: un *appel de routine prédéfinie simple CHILL* est un *appel de routine prédéfinie* qui ne donne ni valeur ni locus. Les routines prédéfinies simples pour l'entrée-sortie sont décrites au paragraphe 7. Les routines prédéfinies simples pour la temporisation sont décrites au paragraphe 9.

6.20.2 Appels de routine prédéfinie rendant locus CHILL

syntaxe:

<appel de routine prédéfinie rendant locus CHILL> ::= *(1)*
<appel de routine prédéfinie rendant locus d'e/s> *(1.1)*

sémantique: un *appel de routine prédéfinie rendant locus CHILL* est un *appel de routine prédéfinie* qui donne un locus. Les routines prédéfinies rendant locus pour l'entrée-sortie sont décrites à l'article 7.

6.20.3 Appels de routine prédéfinie rendant valeur CHILL

syntaxe:

<appel de routine prédéfinie rendant valeur CHILL> ::= *(1)*
NUM (*<expression discrète>*) *(1.1)*
| *PRED* (*<expression discrète>*) *(1.2)*
| *SUCC* (*<expression discrète>*) *(1.3)*
| *ABS* (*<expression numérique>*) *(1.4)*
| *CARD* (*<expression ensembliste>*) *(1.5)*
| *MAX* (*<expression ensembliste>*) *(1.6)*
| *MIN* (*<expression ensembliste>*) *(1.7)*
| *SIZE* ({*<locus>* | *<argument de mode>*}) *(1.8)*
| *UPPER* (*<argument pour supérieur/inférieur>*) *(1.9)*
| *LOWER* (*<argument pour supérieur/inférieur>*) *(1.10)*
| *LENGTH* (*<argument de longueur>*) *(1.11)*
| *<appel de routine prédéfinie affecter>* *(1.12)*
| *<appel de routine prédéfinie rendant valeur d'e/s>* *(1.13)*
| *<appel de routine prédéfinie de valeur temps>* *(1.14)*
| *SIN* (*<expression à virgule flottante>*) *(1.15)*
| *COS* (*<expression à virgule flottante>*) *(1.16)*
| *TAN* (*<expression à virgule flottante>*) *(1.17)*

| *ARCSIN* (<*expression à virgule flottante*>) *(1.18)*
| *ARCCOS* (<*expression à virgule flottante*>) *(1.19)*
| *ARCTAN* (<*expression à virgule flottante*>) *(1.20)*
| *EXP* (<*expression à virgule flottante*>) *(1.21)*
| *LN* (<*expression à virgule flottante*>) *(1.22)*
| *LOG* (<*expression à virgule flottante*>) *(1.23)*
| *SQRT* (<*expression à virgule flottante*>) *(1.24)*

<*expression numérique*> *::=* *(2)*
<*expression entière*> *(2.1)*
| <*expression à virgule flottante*> *(2.2)*

<*argument de mode*> *::=* *(3)*
<*nom de mode*> *(3.1)*
| <*nom de mode matrice*> (<expression>) *(3.2)*
| <*nom de mode chaîne*> (<expression entière>) *(3.3)*
| <*nom de mode structure variable*> (<*liste d'expressions*>) *(3.4)*

<*argument pour supérieur/inférieur*> *::=* *(4)*
<*locus matrice*> *(4.1)*
| <expression matrice> *(4.2)*
| <*nom de mode matrice*> *(4.3)*
| <*locus chaîne*> *(4.4)*
| <*expression chaîne*> *(4.5)*
| <*nom de mode chaîne*> *(4.6)*
| <*locus discret*> *(4.7)*
| <*expression discrète*> *(4.8)*
| <*nom de mode discret*> *(4.9)*
| <*locus virgule flottante*> *(4.10)*
| <*expression virgule flottante*> *(4.11)*
| <*nom de mode virgule flottante*> *(4.12)*
| <*locus accès*> *(4.13)*
| <*nom de mode accès*> *(4.14)*
| <*locus texte*> *(4.15)*
| <*nom de mode texte*> *(4.16)*

<*argument longueur*> *::=* *(5)*
<*locus chaîne*> *(5.1)*
| <*expression chaîne*> *(5.2)*
| <*nom de mode chaîne*> *(5.3)*
| <*locus événement*> *(5.4)*
| <*nom de mode événement*> *(5.5)*
| <*locus tampon*> *(5.6)*
| <*nom de mode tampon*> *(5.7)*
| <*locus texte*> *(5.8)*
| <*nom de mode texte*> *(5.9)*

NOTE – Si l'*argument pour supérieur/inférieur* est un *locus* (matrice, chaîne, discret, virgule flottante), l'ambiguïté syntaxique est résolue comme suit: on interprète *argument pour supérieur/inférieur* comme un *locus* plutôt que comme une *expression* ou une *valeur primitive*. Si l'argument longueur est un *locus chaîne*, on résout l'ambiguïté syntaxique en interprétant l'*argument longueur* comme un *locus* et non comme une *expression*.

sémantique: un *appel de routine prédéfinie rendant valeur CHILL* est un *appel de routine prédéfinie* qui donne une valeur.

NUM donne une valeur entière qui a la même représentation interne que la valeur donnée par son argument.

PRED et *SUCC* donnent respectivement la valeur discrète immédiatement inférieure ou supérieure de leur argument.

ABS est défini par des valeurs numériques, c'est-à-dire des valeurs entières et des valeurs à virgule flottante, qui représentent la valeur absolue correspondante.

CARD, *MAX* et *MIN* sont définis pour des valeurs ensemblistes. *CARD* donne le nombre de valeurs d'élément dans son argument.

MAX et *MIN* donnent respectivement la plus grande et la plus petite valeurs d'élément dans leur argument.

SIZE est défini pour les locus **référençables** et pour les modes (éventuellement dynamiques). Dans le premier cas, il donne le nombre d'unités de mémoire adressables occupées par ce locus, dans le second cas, le nombre d'unités de mémoire adressables qu'un locus **référençable** de ce mode occuperait. Le mode est statique si *l'argument de mode* est un *nom de mode*, sinon cela en est une version dynamiquement paramétrée avec des paramètres spécifiés dans *l'argument de mode*. Dans le premier cas, le *locus* n'est pas évalué lors de l'exécution.

UPPER et *LOWER* sont définis dans les éventualités ci-après (éventuellement dynamiques):

- locus matrice, chaîne, discret, virgule flottante, accès et texte donnant la **borne supérieure** et la **borne inférieure** du mode du locus;
- expressions matrice et chaîne, donnant la **borne supérieure** et la **borne inférieure** du mode de la classe de valeur;
- expressions discrètes et virgule flottante **fortes**, donnant la **borne supérieure** et la **borne inférieure** du mode de la classe de valeur;
- noms de **mode** matrice, chaîne, discret, virgule flottante, accès et texte donnant la **borne supérieure** et la **borne inférieure** du mode.

LENGTH est défini dans les éventualités ci-après (éventuellement dynamiques):

- locus chaîne et texte et expressions chaîne donnant les valeurs effectives de ceux-ci;
- locus événement donnant la **longueur événement** du mode locus;
- locus tampon donnant la **longueur tampon** du mode locus;
- noms de **mode** chaîne donnant la **longueur chaîne** du mode;
- noms de **mode** texte donnant la **longueur texte** du mode;
- noms de **mode** tampon donnant la **longueur tampon** du mode;
- noms de **mode** événement donnant la **longueur événement** du mode.

SIN donne le sinus de son argument (en radians)

COS donne le cosinus de son argument (en radians)

TAN donne la tangente de son argument (en radians)

ARCSIN donne la fonction $\sin^{-1}$ de son argument dans la gamme $-\pi/2 : \pi/2$

ARCCOS donne la fonction $\cos^{-1}$ de son argument dans la gamme $0 : \pi$

ARCTAN donne la fonction $\tan^{-1}$ de son argument dans la gamme $-\pi/2 : \pi/2$

EXP donne la fonction e^x, l'argument étant x

LN donne le logarithme naturel de son argument

LOG donne le logarithme décimal de son argument

SQRT donne la racine carrée de son argument

Les règles qui s'appliquent pour l'évaluation du résultat de *appel de routine prédéfinie* avec des arguments **constants** sont les mêmes que celles de *expression* **constante** (voir 5.3.1).

propriétés statiques: la classe d'un appel de routine prédéfinie *NUM* est la *&INT*-classe par dérivation. L'appel de routine prédéfinie est **constant** (**littéral)** si et seulement si l'argument est **constant** (**littéral**).

La classe d'un appel de routine prédéfinie *PRED* ou *SUCC* est la **classe résultante** de l'argument. L'appel de routine prédéfinie est **constant** (**littéral**) si et seulement si l'argument est **constant** (**littéral**).

La classe d'un appel de routine prédéfinie *ABS* est la **classe résultante** de l'argument. L'appel de routine prédéfinie est **constant** (**littéral**) si et seulement si l'argument est **constant** (**littéral**).

La classe d'un appel de routine prédéfinie *CARD* est la *&INT*-classe par dérivation. L'appel de routine prédéfinie est **constant** si et seulement si l'argument est **constant**.

La classe d'un appel de routine prédéfinie *MAX* ou *MIN* est la M-classe par valeur, où M est le mode **primitif** du mode de *l'expression ensembliste*. L'appel de routine prédéfinie est **constant** si et seulement si l'argument est **constant**.

La classe d'un appel de routine prédéfinie *SIZE* est la *&INT*-classe par dérivation. L'appel de routine prédéfinie est **constant** si le mode de l'argument est statique.

La classe d'un appel de routine prédéfinie *UPPER* et *LOWER* est:

- la M-classe par valeur si l'*argument* pour *supérieur/inférieur* est un *locus* *matrice*, une *expression* *matrice* ou un *nom de* *mode matrice*, où M est respectivement le mode **indice** du *locus* *matrice*, d'une *expression* *matrice* ou un *nom de* *mode matrice*;
- la *&INT*-classe par dérivation si l'*argument* pour *supérieur/inférieur* est un *locus* *chaîne*, une *expression* *chaîne* ou un *nom de* *mode chaîne*;
- la M-classe par valeur si l'*argument* pour *supérieur/inférieur* est un *locus* *discret*, une *expression* *discrète* ou un *nom de* *mode discret*, où M est respectivement le mode du *locus* *discret*, le mode de l'*expression* *discrète*, ou le nom de *mode discret*.
- la M-classe par valeur si l'*argument* pour *supérieur/inférieur* est un *locus* *virgule flottante*, une *expression à* *virgule flottante* ou un *nom de* *mode virgule flottante*, où M est respectivement le mode du *locus* *virgule flottante*, d'une expression *à virgule flottante* ou du nom de *mode* *virgule flottante*.
- la M-classe par valeur si l'*argument* pour *supérieur/inférieur* est un *locus* *accès* ou un *nom de* *mode accès*, où M est respectivement le mode **indice** du mode de *locus* *accès* ou du *nom de* *mode accès*.
- la M-classe par valeur si *l'argument* pour *supérieur/inférieur* est un *locus* *texte* ou un *nom de* *mode texte*, où M est respectivement le mode **indice** du mode de *locus* *texte* ou du *nom de* *mode texte*.

Un appel de routine prédéfinie *UPPER* ou *LOWER* est **littéral** si *l'argument* pour *supérieur/inférieur* est un *nom de mode* (*matrice*, *chaîne*, *mode discret*, *virgule flottante*, *accès ou texte*), si le mode *locus* *matrice* ou *chaîne* est statique, si *l'expression* *matrice* ou *chaîne* a une classe statique ou si *l'argument* pour *supérieur/inférieur* est un *locus* *discret*, une *expression discrète*, un *locus* *virgule flottante*, une *expression* *virgule flottante*, un *locus* *accès* ou un *locus* *texte*.

La classe d'un appel de routine prédéfinie *LENGTH* est la *&INT*-classe par dérivation. La routine prédéfinie est **littérale** si l'*argument longueur* est un *locus* *chaîne* avec un mode statique, une *expression* *chaîne* avec une classe statique, un *locus* *événement*, ou un *locus* *tampon*, ou s'il est un *nom de* *mode chaîne*, un *nom de* *mode événement*, un *nom de* *mode tampon* ou un *nom de* *mode texte*.

La classe de routine prédéfinie *TAN*, *EXP*, *LN*, *LOG* ou *SQRT* est la **classe résultante** de son argument.

La classe de *SIN*, *COS*, *ARCSIN*, *ARCCOS* et *ARCTAN* est 1. la N-classe par dérivation, 2. la N-classe par valeur si la classe de l'argument est 1. une N-classe par dérivation, 2. une N-classe par valeur, où N est un mode construit de la manière suivante:

- pour *SIN*: &**RANGE** (–1.0 : 1.0, *S*)
- pour *COS*: &**RANGE** (–1.0 : 1.0, *S*)
- pour *ARCSIN*: &**RANGE** (–π/2 : π/2, *S*)
- pour *ARCCOS*: &**RANGE** (0 : π, *S*)
- pour *ARCTAN*: &**RANGE** (–π/2 : π/2, *S*)

où S est la **précision** de N et la **nouveauté** est celle de N.

Une routine prédéfinie *SIN*, *COS*, *TAN*, *ARCSIN*, *ARCCOS*, *ARCTAN*, *EXP*, *LN*, *LOG* ou *SQRT* est **constate** (**littérale**) si, et seulement si, l'argument est **constant** (**littéral**).

conditions statiques: si l'argument d'un appel de routine prédéfinie *PRED* ou *SUCC* est **constant**, il ne doit pas donner, respectivement, la plus petite ou la plus grande valeur discrète définie par le mode **racine** de la classe de l'argument. Le mode **racine** de l'argument d'*expression* *discrète* de *PRED* et *SUCC* ne doit pas être un mode ensemble **avec numéros**.

Si l'argument d'un appel de routine prédéfinie *MAX* ou *MIN* est **constant**, il ne doit pas donner la valeur ensembliste vide.

L'argument pour *locus* de *SIZE* doit être **référençable**.

L'*expression* *discrète* et l'*expression à* *virgule flottante* en tant qu'argument de *UPPER* et *LOWER* doit être **forte**.

Si l'*argument* pour *supérieur/inférieur* est un *nom de* *mode accès* ou un *locus* *accès*, le mode d'accès correspondant doit avoir un mode **indice**.

Si l'*argument* pour *supérieur/inférieur* est un *nom de* *mode texte* ou un *locus* *texte*, le mode d'accès correspondant doit avoir un mode **indice**.

Les conditions de compatibilité suivantes doivent être remplies pour un *argument de mode* qui n'est pas un simple *nom de mode*:

- la classe de *l'expression* doit être **compatible** avec le mode **indice** du mode du *nom de mode matrice*;
- le *nom de mode structure variable* doit être **paramétrable** et il doit y avoir autant d'expressions dans la *liste d'expressions* qu'il y a de classes dans sa liste de classes, et la classe de chaque expression doit être **compatible** avec la classe correspondante de la liste de classes.

conditions dynamiques: *PRED* et *SUCC* qui ne sont pas **constants** causent l'exception *OVERFLOW* s'ils sont appliqués à la plus petite ou à la plus grande valeur discrète définie par le mode **racine** de la classe de leur argument.

NUM et *CARD* qui ne sont pas **constants** causent l'exception *OVERFLOW* si la valeur résultante est en dehors de l'ensemble de valeurs définies par *&INT*.

MAX et *MIN* causent l'exception *EMPTY* s'ils sont appliqués à des valeurs ensembliste vides.

ABS non **constant** cause l'exception *OVERFLOW* si la valeur résultante est en dehors des bornes définies par le mode **racine** de la classe de l'argument.

L'exception *RANGEFAIL* se produit si, dans *l'argument de mode*:

- l'*expression* donne une valeur qui est en dehors de l'ensemble de valeurs définies par le mode **indice** du *nom de mode matrice*;
- l'*expression entière* donne une valeur négative ou une valeur qui est supérieure à la **longueur de chaîne** du *nom de mode chaîne*;
- une expression de la *liste d'expressions* pour laquelle la classe correspondante de la liste de classes du *nom de mode structure variable* est une M-classe par valeur (c'est-à-dire est **forte**), donne une valeur qui est en dehors de l'ensemble de valeurs définies par M.

ARCSIN et *ARCCOS* qui ne sont pas **constants** causent l'exception *OVERFLOW* si l'argument ne se situe pas dans la fourchette *–1,0 : 1,0*.

LN et *LOG* qui ne sont pas **constants** causent l'exception *OVERFLOW* si l'argument n'est pas supérieur à zéro.

SQRT qui n'est pas **constant** cause l'exception *OVERFLOW* si l'argument n'est pas supérieur ou égal à zéro.

SIN, *COS*, *TAN*, *ARCSIN*, *ARCTAN*, *LN* et *LOG* qui ne sont pas **constants** causent l'exception *OVERFLOW* si la valeur résultante est plus grande que la **borne supérieure** ou plus petite que la **borne inférieure** du mode **racine** de la classe de l'argument. Si la valeur mathématique exacte résultante est supérieure à la **borne supérieure négative** et inférieure à la **borne inférieure positive** du mode **racine** de l'argument, et qu'elle est différente de zéro, l'exception *UNDERFLOW* se produit.

ARCCOS, *EXP* et *SQRT* qui ne sont pas **constants** causent l'exception *OVERFLOW* si la valeur résultante est plus grande que la **borne supérieure** ou plus petite que la **borne inférieure** du mode **racine** de la classe de l'argument. Si la valeur mathématique exacte résultante est supérieure à zéro et inférieure à la **borne inférieure positive** du mode **racine** de l'argument, et qu'elle est différente de zéro, l'exception *UNDERFLOW* se produit.

exemples:

9.12 *MIN (sieve)* *(1.7)*

11.47 *PRED (col_1)* *(1.2)*

11.47 *SUCC (col_1)* *(1.3)*

6.20.4 Routines prédéfinies de traitement de mémoire dynamique

syntaxe:

<appel de routine prédéfinie affecter> ::= *(1)*
GETSTACK (*<argument de mode>* [, *<valeur>* |
([*<liste des paramètres effectifs de constructeur>*])]) *(1.1)*
| *ALLOCATE* (*<argument de mode>* [, *<valeur>* |
([*<liste des paramètres effectifs de chonstructeur>*])]) *(1.2)*

<appel de routine prédéfinie terminer> ::= *(2)*
TERMINATE (*<valeur primitive référence>*) *(2.1)*

sémantique: *GETSTACK* et *ALLOCATE* créent un locus du mode spécifié et donnent une valeur référence pour le locus créé. *GETSTACK* crée ce locus sur la pile (voir 10.9). Un locus dont le mode est celui de *l'argument de mode* est créé et

une valeur se référençant à lui est donnée. Le locus créé est initialisé avec la valeur *de valeur*, si présente; sinon, avec la valeur **nondéfinie** (voir 4.1.2) si l'argument de mode n'est pas un *mode moreta*.

Si l'*argument de mode* est un *mode moreta*, toutes les initialisations des composantes sont effectuées en premier lieu, dans l'ordre textuel de leur apparition. Si une liste de paramètres (éventuellement vide) est spécifiée, le **constructeur** correspondant de l'*argument de mode* est appliqué au locus nouvellement créé. Si l'*argument de mode* est un *mode tâche*, la tâche appartenant au locus nouvellement créé est lancée.

TERMINATE met fin à la durée de vie du locus référencé par la valeur donnée par la *valeur primitive référence*. Une implémentation pourrait en conséquence libérer la mémoire occupée par ce locus et, si la *valeur primitive référence* est un locus qui n'est pas **protégé**, affecter la valeur **indéfinie** au locus.

Si la *valeur primitive référence* se réfère à un locus **L** de **région** ou de **tâche**, les opérations suivantes sont exécutées dans l'ordre indiqué:

a) le locus **L** est fermé. Si un locus est fermé, plus aucun appel extérieur des procédures de composante **publique** de **L** n'est accepté;

b) le fil d'exécution de *TERMINATE* attend que **L** soit vide;

c) si le mode de **L** contient un **destructeur**, celui-ci est appliqué à **L**.

propriétés statiques: la classe d'un appel de routine prédéfinie *GETSTACK* ou *ALLOCATE* est la M-classe par référence, dans laquelle M est le mode de l'*argument de mode*. M est soit le *nom de mode*, soit un mode **paramétré** construit ainsi:

& <nom de mode matrice> (<expression>) ou

& <nom de mode chaîne> (<expression entière>) ou

& <nom de mode structure variable> (<liste d'expressions>),

respectivement.

Un appel de routine prédéfinie *GETSTACK* ou *ALLOCATE* est **intrarégional** s'il est englobé dans une région, sinon il est **extrarégional**.

conditions statiques: la classe de la *valeur*, si elle est présente, dans l'appel de routine prédéfinie *GETSTACK* et *ALLOCATE*, doit être **compatible** avec le mode *de l'argument de mode*; cette vérification est dynamique dans le cas où le mode de l'*argument de mode* est un mode dynamique.

Si le mode de l'*argument de mode* a la **propriété de protection**, le deuxième argument doit être présent.

La *valeur*, si elle est présente, dans l'appel de routine prédéfinie *GETSTACK* et *ALLOCATE*, doit être **régionalement sûre** pour le locus créé.

propriétés dynamiques: une valeur référence est une valeur référence **affectée** si, et seulement si elle est envoyée par un appel de routine prédéfinie *ALLOCATE*.

conditions dynamiques: *GETSTACK* cause l'exception *SPACEFAIL* si les besoins de mémoire ne peuvent pas être satisfaits.

ALLOCATE cause l'exception *ALLOCATEFAIL* si les besoins de mémoire ne peuvent pas être satisfaits.

Pour *GETSTACK* et *ALLOCATE*, les conditions d'affectation de la valeur donnée par *valeur* par rapport au mode de l'*argument de mode* sont applicables.

TERMINATE cause l'exception *EMPTY* si *la valeur primitive référence* donne la valeur *NULL*.

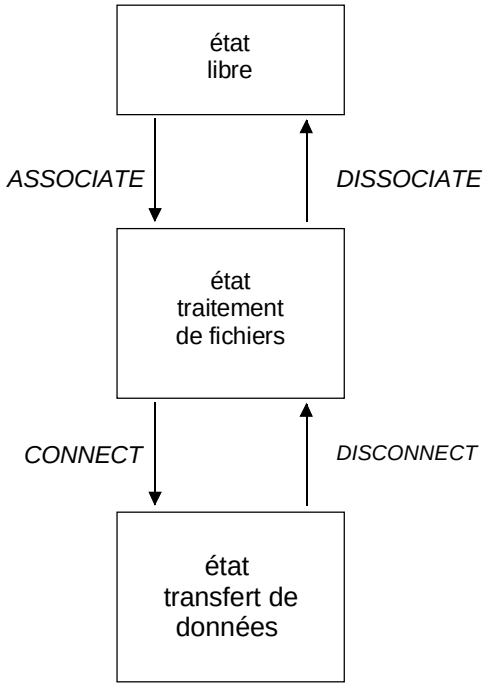
La *valeur primitive référence* doit donner une valeur référence **affectée**. La durée de vie du locus référencé ne doit pas être terminée.

7 Entrée et sortie

7.1 Modèle de référence E/S

Un modèle est utilisé pour la description des facilités d'entrée-sortie, d'une manière indépendante de l'implémentation; on distingue trois états pour un locus association donné: un état libre, un état traitement de fichiers et un état transfert de données.

Le diagramme ci-après représente ces trois états et les transitions possibles entre ceux-ci.



Le locus asociation ne contient pas de valeur.
Aucune relation avec un objet du monde extérieur.

Le locus association contient une association.
Des opérations créent et suppriment un fichier,
ou en modifient les propriétés.

Le locus accès est connecté au locus association.
Transfert de données vers un fichier ou à partir
de celui-ci: opérations lire et écrire.

Le modèle est fondé sur l'hypothèse que des objets, qui dans des implémentations, sont souvent appelés ensembles de données, fichiers, ou dispositifs, existent dans le monde extérieur, c'est-à-dire dans un milieu extérieur à un programme CHILL. Dans le modèle, cet objet du monde extérieur est appelé fichier. Un fichier peut être un dispositif matériel, une ligne de communication ou simplement un fichier d'un système de gestion de fichiers. En général, un fichier est un objet qui peut produire et utiliser des données.

En CHILL, l'utilisation d'un fichier nécessite une association; une association est créée par l'opération "associate" et elle identifie un fichier. Une association a des attributs; ces attributs décrivent les propriétés d'un fichier qui est ou peut être lié à l'association.

Dans l'état libre, il n'y a ni interaction ni relation entre le programme CHILL et des objets du monde extérieur. L'opération "associate" modifie l'état du modèle, qui passe de l'état libre à l'état traitement de fichiers. Cette opération prend pour argument un locus association et une dénotation définie par l'implémentation pour un objet du monde extérieur pour lequel une association doit être créée; on peut utiliser des arguments supplémentaires pour indiquer le type d'association de l'objet et les valeurs initiales des attributs de l'association. En outre, une association particulière implique un ensemble d'opérations (dépendant de l'implémentation) qui peut être appliqué au fichier attaché à cette association.

Dans l'état traitement de fichiers, il est possible de manipuler un fichier et ses propriétés par l'intermédiaire d'une association, à condition que l'association permette cette opération particulière; pour des opérations qui modifient les propriétés d'un fichier, une association réservée exclusivement au fichier sera normalement nécessaire.

Dans le modèle, on admet que les associations sont généralement exclusives, c'est-à-dire qu'une seule association existe au même moment pour un objet donné du monde extérieur. Toutefois, des implémentations peuvent admettre la création de plusieurs associations pour le même objet, à condition que cet objet puisse être partagé par différents utilisateurs (programmes) et différentes associations dans le même programme. Toutes les opérations effectuées dans l'état traitement de fichiers prennent une association pour argument.

L'opération de dissociation est utilisée pour mettre fin à une association relative à un objet du monde extérieur; cette opération fait que le locus revient de l'état traitement de fichiers à l'état libre.

Le transfert de données vers un fichier ou à partir de celui-ci n'est possible que dans l'état transfert de données; les opérations de transfert exigent qu'un locus accès soit connecté à une association relative à ce fichier. L'opération de connexion relie un locus accès à une association et modifie l'état du modèle, qui passe à l'état transfert de données. L'opération prend pour arguments un locus association et un locus accès; le locus association contient une association

avec un fichier dans lequel ou à partir duquel les données peuvent être transférées par l'intermédiaire du locus accès. Des arguments supplémentaires de l'opération de connexion indiquent pour quel type d'opération de transfert le locus accès doit être connecté et dans quel registre le fichier doit être classé. Un locus accès au plus peut être connecté avec un locus association à un moment donné.

L'opération de déconnexion prend pour argument un locus accès et le déconnecte de l'association à laquelle il est relié; elle modifie l'état du modèle, qui revient à l'état traitement de fichiers.

Dans l'état transfert de données, un locus accès doit être utilisé comme argument d'une opération de transfert; deux opérations de transfert sont possibles, à savoir une opération lire, pour transférer des données d'un fichier au programme, et une opération écrire, pour transférer des données du programme à un fichier. Les opérations de transfert utilisent le mode enregistrement du locus accès pour transformer des valeurs CHILL en enregistrement de fichiers, et vice versa.

Dans le modèle, un fichier se présente comme une matrice de valeurs; chaque élément de cette matrice se rapporte à un enregistrement du fichier. Le mode élément de cette matrice est déterminé par l'opération de connexion qui est le mode enregistrement du locus accès qui est connecté. Une valeur d'indice est affectée à chaque enregistrement du fichier; cette valeur identifie de manière unique chaque enregistrement du fichier. Dans la description des opérations de connexion et de transfert, trois valeurs d'indice spéciales seront utilisées, à savoir un indice de **base**, un indice **courant** et un indice de **transfert**. L'indice de **base** est fixé par l'opération de connexion et reste inchangé jusqu'à une opération de connexion suivante. Il est utilisé pour calculer l'indice de **transfert** dans des opérations de transfert et l'indice **courant** dans une opération de connexion. L'indice de **transfert** indique dans le fichier la position où un transfert aura lieu; l'indice **courant** désigne l'enregistrement sur lequel le fichier est actuellement placé.

7.2 Valeurs d'association

7.2.1 Généralités

Une valeur d'association reflète les propriétés d'un fichier qui est ou qui peut lui être rattaché. Une valeur d'association déterminée implique en outre un ensemble d'opérations (dépendant de l'implémentation) sur le fichier qui lui est éventuellement rattaché.

Les valeurs d'association ne possèdent pas de dénotation mais elles sont contenues dans des locus de mode association; il n'existe pas d'expression désignant une valeur de mode association. Les valeurs d'association ne peuvent être manipulées que par des routines prédéfinies qui prennent un locus d'association pour paramètre.

7.2.2 Attributs des valeurs d'association

Une valeur d'association a des attributs, qui décrivent les propriétés de l'association et le fichier qui peut ou qui pourrait y être rattaché.

Les attributs suivants sont définis par le langage:

- **existant**: un fichier (éventuellement vide) est rattaché à l'association;
- **lisible**: les opérations lire sont possibles pour le fichier lorsqu'il est rattaché à l'association;
- **écrivable**: les opérations écrire sont possibles pour le fichier lorsqu'il est rattaché à l'association;
- **indexable**: lorsqu'il est rattaché à l'association, le fichier permet l'accès aléatoire à ses enregistrements;
- **séquençable**: lorsqu'il est rattaché à l'association, le fichier permet l'accès séquentiel à ses enregistrements;
- **variable**: la **taille** des enregistrements du fichier, lorsque celui-ci est rattaché à l'association, peut varier à l'intérieur du fichier.

Ces attributs ont une valeur booléenne; les attributs sont initialisés lorsque l'association est créée et peuvent être mis à jour à la suite d'opérations particulières sur l'association. Cette liste ne comprend que des attributs définis par le langage; des implémentations peuvent ajouter des attributs selon leurs propres besoins.

7.3 Valeurs d'accès

7.3.1 Généralités

Des valeurs d'accès sont contenues dans des locus de mode accès. Il faut un locus accès pour transférer des données d'un fichier au monde extérieur ou vice versa.

Les valeurs d'accès n'ont pas de dénotation mais sont contenues dans des locus de mode accès; il n'existe pas d'expression désignant une valeur de mode accès. Les valeurs d'accès ne peuvent être manipulées que par des routines prédéfinies qui prennent pour paramètre un locus d'accès.

7.3.2 Attributs des valeurs d'accès

Les valeurs d'accès ont des attributs qui décrivent leurs propriétés dynamiques, la sémantique des opérations de transfert et les conditions dans lesquelles des exceptions peuvent se produire.

Le CHILL définit les attributs suivants:

- **usage**: indiquant pour quelle ou quelles opérations de transfert le locus accès est connecté à une association; l'attribut est fixé par l'opération de connexion;
- **hors du fichier**: indiquant si l'indice de **transfert** calculé par la dernière opération lire était ou non dans le fichier; l'attribut est initialisé sur *FALSE* par l'opération de connexion et fixé par chaque opération lire.

7.4 Routines prédéfinies pour entrée-sortie

7.4.1 Généralités

Les routines prédéfinies par le langage sont définies pour des opérations sur des locus association et des locus accès ainsi que pour examiner et modifier les attributs de leurs valeurs.

Les routines prédéfinies sont décrites dans les paragraphes ci-après:

syntaxe:

<appel de routine prédéfinie rendant valeur d'e/s> ::= *(1)*
<appel de routine prédéfinie attribut d'association> *(1.1)*
| *<appel de routine prédéfinie est associé>* *(1.2)*
| *<appel de routine prédéfinie attribut d'accès>* *(1.3)*
| *<appel de routine prédéfinie lire article>* *(1.4)*
| *<appel de routine prédéfinie obtenir texte>* *(1.5)*

<appel de routine prédéfinie simple d'e/s> ::= *(2)*
<appel de routine prédéfinie dissocier> *(2.1)*
| *<appel de routine prédéfinie modification>* *(2.2)*
| *<appel de routine prédéfinie connecter>* *(2.3)*
| *<appel de routine prédéfinie déconnecter>* *(2.4)*
| *<appel de routine prédéfinie écrire article>* *(2.5)*
| *<appel de routine prédéfinie texte>* *(2.6)*
| *<appel de routine prédéfinie fixer texte>* *(2.7)*

<appel de routine prédéfinie rendant locus d'e/s> ::= *(3)*
<appel de routine prédéfinie associer> *(3.1)*

conditions statiques: Un *paramètre de routine prédéfinie* dans une routine prédéfinie d'e/s qui est un locus *association*, *accès* ou *texte* doit être **référençable**.

7.4.2 Association avec un objet du monde extérieur

syntaxe:

<appel de routine prédéfinie associer> ::= *(1)*
ASSOCIATE (*<locus association>* [, *<liste de paramètres pour associer>*]) *(1.1)*

<appel de routine prédéfinie est associé> ::= *(2)*
ISASSOCIATED (*<locus association>*) *(2.1)*

<liste de paramètres pour associer> ::= *(3)*
<paramètre pour associer> { , *<paramètre pour associer>* }* *(3.1)*

<paramètre pour associer> ::= *(4)*
<locus> *(4.1)*
| *<valeur>* *(4.2)*

sémantique: *ASSOCIATE* crée une association avec un objet du monde extérieur. Il initialise le *locus association* avec l'association créée. Il initialise les attributs de l'association créée. En outre, le locus association est renvoyé comme résultat de l'appel. L'association particulière qui est créée est déterminée par les locus et les valeurs qui apparaissent dans la *liste de paramètres pour associer*; les modes (classes) et la sémantique de ces locus (valeurs) sont définis par l'implémentation.

ISASSOCIATED renvoie *TRUE* si le *locus association* contient une association et, sinon, *FALSE*.

propriétés statiques: la classe d'un appel de routine prédéfinie *ISASSOCIATED* est la *BOOL*-classe par dérivation. Le mode de l'appel de routine prédéfinie *ASSOCIATE* est le mode du *locus association*.

La **régionalité** d'un appel de routine prédéfinie *ASSOCIATION* est celle du *locus association*.

conditions statiques: le mode et la classe de chaque *paramètre pour associer* sont définis par l'implémentation.

conditions dynamiques: *ASSOCIATE* cause l'exception *ASSOCIATEFAIL* si le *locus association* contient déjà une association ou si l'association ne peut être créée pour des raisons définies par l'implémentation.

exemple:

20.21 ASSOCIATE (file_association,"DSK:RECORDS.DAT"); *(1.1)*

7.4.3 Dissociation d'un objet du monde extérieur

syntaxe:

<appel de routine prédéfinie dissocier> ::= *(1)*
DISSOCIATE (*<locus association>*) *(1.1)*

sémantique: *DISSOCIATE* met fin à une association avec un objet du monde extérieur. Si un locus accès est encore connecté à l'association contenue dans un locus association, il est déconnecté avant que l'association ne soit terminée.

conditions dynamiques: *DISSOCIATE* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

exemple:

22.38 DISSOCIATE (association); *(1.1)*

7.4.4 Accès aux attributs association

syntaxe:

<appel de routine prédéfinie attribut d'association> ::= *(1)*
EXISTING (*<locus association>*) *(1.1)*
| *READABLE* (*<locus association>*) *(1.2)*
| *WRITEABLE* (*<locus association>*) *(1.3)*
| *INDEXABLE* (*<locus association>*) *(1.4)*
| *SEQUENCIBLE* (*<locus association>*) *(1.5)*
| *VARIABLE* (*<locus association>*) *(1.6)*

sémantique: *EXISTING*, *READABLE*, *WRITEABLE*, *INDEXABLE*, *SEQUENCIBLE* et *VARIABLE* rendent respectivement la valeur de l'attribut **existant**, **lisible**, **écrivable**, **indexable**, **séquençable** et **variable**, de l'association contenue dans le *locus association*.

propriétés statiques: la classe de *l'appel de routine prédéfinie attribut d'association* est la *BOOL*-classe par dérivation.

conditions dynamiques: *l'appel de routine prédéfinie attribut d'association* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

7.4.5 Modification des attributs association

syntaxe:

<appel de routine prédéfinie modification> ::= *(1)*
CREATE (*<locus association>*) *(1.1)*
| *DELETE* (*<locus association>*) *(1.2)*
| *MODIFY* (*<locus association>* [, *<liste de paramètres pour modifier>*]) *(1.3)*

<liste de paramètres pour modifier> ::= *(2)*
<paramètre pour modifier> { , *<paramètre pour modifier>* }* *(2.1)*

<paramètre pour modifier> ::= *(3)*
<valeur> *(3.1)*
| *<locus>* *(3.2)*

sémantique: *CREATE* crée un fichier vide et le rattache à l'association désignée par le *locus association*. L'attribut **existant** de l'association indiquée donne *TRUE* si l'opération réussit.

DELETE détache un fichier de l'association désignée par le *locus association* et supprime le fichier. L'attribut **existant** de l'association indiquée donne *FALSE* si l'opération réussit.

MODIFY fournit les moyens de changer les propriétés d'un objet du monde extérieur pour lequel il existe une association et qui est désigné par *locus association*; les locus et les valeurs qui apparaissent dans la *liste de paramètres pour modifier* indiquent comment modifier les propriétés. Les modes (classes) et la sémantique de ces locus (valeurs) sont définis par l'implémentation.

conditions dynamiques: *CREATE*, *DELETE* et *MODIFY* causent l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

CREATE cause l'exception *CREATEFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *TRUE*;
- la création du fichier échoue (définie par l'implémentation).

DELETE cause l'exception *DELETEFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *FALSE*;
- la suppression du fichier échoue (définie par l'implémentation).

MODIFY cause l'exception *MODIFYFAIL* si les propriétés, définies par la *liste de paramètres pour modifier* peuvent ou ne peuvent pas être modifiées; les conditions dans lesquelles cette exception peut se produire sont définies par l'implémentation.

exemples:

21.39 *CREATE (outassoc)*; *(1.1)*

21.69 *DELETE (curassoc)*; *(1.2)*

7.4.6 Connexion d'un locus accès

syntaxe:

<appel de routine prédéfinie connecter> ::= *(1)*
CONNECT (*<locus transfert>*, *<locus association>*,
<expression usage> [, *<expression positionnement>* [, *<expression indice>*]]) *(1.1)*

<locus transfert> ::= *(2)*
<locus accès> *(2.1)*
| *<locus texte>* *(2.2)*

<expression usage> ::= *(3)*
<expression> *(3.1)*

<expression positionnement> ::= *(4)*
<expression> *(4.1)*

<expression indice> ::= *(5)*
<expression> *(5.1)*

noms prédéfinis: pour commander l'opération de connexion exécutée par la routine prédéfinie *CONNECT*, deux noms de **synmode** sont prédéfinis dans le langage, à savoir *USAGE* et *WHERE*; leurs modes **définissants** sont: **SET** (*READONLY*, *WRITEONLY*, *READWRITE*) et **SET** (*FIRST*, *SAME*, *LAST*), respectivement.

Les valeurs du mode *USAGE* indiquent pour quel type d'opération de transfert le locus accès doit être connecté à une association et les valeurs du mode *WHERE* indiquent comment le fichier qui est rattaché à une association doit être placé par l'opération de connexion.

sémantique: *CONNECT* rattache le locus accès dénoté par le *locus transfert* à l'association qui est contenue dans le *locus association*; il doit y avoir un fichier rattaché à l'association désignée, c'est-à-dire que l'attribut **existant** doit être *TRUE*.

Le locus accès dénoté par le *locus transfert* est le locus lui-même s'il est un *locus accès*; sinon, c'est le sous-locus **accès** du *locus texte*.

La valeur donnée par l'*expression usage* indique pour quel type d'opérations de transfert le locus accès doit être connecté à un fichier. Si l'expression donne *READONLY*, la connexion n'est établie que pour des opérations lire; si elle donne *WRITEONLY*, la connexion n'est établie que pour des opérations écrire; si elle donne *READWRITE*, la connexion est établie pour des opérations lire et écrire.

L'attribut **indexable** de l'association désignée doit être *TRUE* si le locus accès a un mode **indice**, tandis que l'attribut **séquençable** doit être *TRUE* si le locus n'a pas de mode **indice**.

CONNECT (re)place le fichier qui est rattaché à l'association désignée, c'est-à-dire qu'il établit un indice de **base** et un indice **courant** (nouveaux) dans le fichier. Le (nouvel) indice de **base** dépend de la valeur donnée par *l'expression positionnement*:

- si *l'expression positionnement* donne *FIRST* ou n'est pas spécifiée, l'indice de **base** est réglé sur 0, c'est-à-dire que le fichier est positionné avant le premier enregistrement;
- si *l'expression positionnement* donne *SAME*, l'indice de **base** est réglé sur l'indice **courant** du fichier, c'est-à-dire que la position du fichier n'est pas modifiée;
- si *l'expression positionnement* donne *LAST*, l'indice de **base** est réglé sur N, où N désigne le nombre d'enregistrements dans le fichier, c'est-à-dire que le fichier est positionné après le dernier enregistrement.

Une fois fixé l'indice de **base**, un indice **courant** sera établi par *CONNECT*. Cet indice **courant** dépend de la spécification facultative d'une *expression indice*:

- si aucune *expression indice* n'est spécifiée, l'indice **courant** est fixé sur le (nouvel) indice de **base**;
- si une *expression indice* est spécifiée, l'indice **courant** est fixé sur

  indice de **base** + *NUM (v) – NUM (l)*

  où *l* désigne la **borne inférieure** du mode **indice** du locus accès et *v* désigne la valeur donnée par l'*expression indice*.

Si le locus accès est connecté pour les opérations écrire séquentielles (c'est-à-dire que le locus accès n'a pas de mode **indice** et que *l'expression usage* donne *WRITEONLY*), alors les enregistrements du fichier qui ont un indice supérieur à l'indice **courant** (nouveau) sont supprimés du fichier, c'est-à-dire que le fichier peut être tronqué ou vidé par *CONNECT*.

Un locus accès qui n'a pas de mode indice ne peut être connecté simultanément à une association pour des opérations lire et écrire.

Tout locus accès auquel l'association désignée peut être connectée sera déconnecté implicitement avant la connexion de l'association au locus désigné par le *locus transfert*.

CONNECT initialise l'attribut **hors du fichier** du locus d'accès sur *FALSE* et fixe l'attribut **usage** conformément à la valeur donnée par l'*expression usage*.

propriétés statiques: le mode rattaché à un *locus transfert* est le mode du *locus accès* ou le mode **accès** du *locus texte*, respectivement.

conditions statiques: le mode du *locus transfert* doit avoir un mode **indice** si une *expression indice* est spécifiée; la classe de la valeur donnée par *l'expression indice* doit être **compatible** avec ce mode **indice**. Le *locus transfert* doit avoir la même **régionalité** que le *locus association*.

La classe de la valeur donnée par l'*expression usage* doit être **compatible** avec la *USAGE*-classe par dérivation.

La classe de la valeur donnée par l'*expression positionnement* doit être **compatible** avec la *WHERE*-classe par dérivation.

conditions dynamiques: *CONNECT* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

CONNECT cause l'exception *CONNECTFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *FALSE*;
- l'attribut **lisible** de l'association est *FALSE* et l'*expression usage* donne *READONLY* ou *READWRITE*;
- l'attribut **écrivable** de l'association est *FALSE* et l'*expression usage* donne *WRITEONLY* ou *READWRITE*;
- l'attribut **indexable** de l'association est *FALSE* et le locus accès a un mode **indice**;
- l'attribut **séquençable** de l'association est *FALSE* et le locus accès n'a pas de mode **indice**;
- l'*expression positionnement* donne *SAME*, tandis que l'association contenue dans le *locus association* n'est pas connectée à un locus accès;
- l'attribut **variable** de l'association est *FALSE* et le locus accès a un mode **enregistrement dynamique**, tandis que l'*expression usage* donne *WRITEONLY* ou *READWRITE*;
- l'attribut **variable** de l'association est *TRUE* et le locus accès a un mode **enregistrement statique**, tandis que l'*expression usage* donne *READONLY* ou *READWRITE*;
- le locus accès n'a pas de mode **indice**, tandis que l'*expression usage* donne *READWRITE*;
- l'association contenue dans le *locus association* ne peut être connectée au locus accès, en raison de conditions définies dans l'implémentation.

CONNECT cause l'exception *RANGEFAIL* si le mode **indice** du locus accès est un mode intervalle discret et que l'*expression indice* donne une valeur extérieure aux bornes de ce mode intervalle discret.

L'exception *EMPTY* est causée si la **référence accès** du *locus texte* donne la valeur *NULL*.

exemples:

20.22 CONNECT (record_file, file_association, READWRITE); *(1.1)*

20.22 READWRITE *(3.1)*

7.4.7 Déconnexion d'un locus accès

syntaxe:

<appel de routine prédéfinie déconnecter> ::= *(1)*
DISCONNECT (*<locus transfert>*) *(1.1)*

sémantique: *DISCONNECT* déconnecte le locus accès dénoté par le *locus transfert* de l'association à laquelle il était connecté.

conditions dynamiques: *DISCONNECT* cause l'exception *NOTCONNECTED* si le locus accès dénoté par le *locus transfert* n'est pas connecté à une association.

7.4.8 Attributs d'accès de locus accès

syntaxe:

<appel de routine prédéfinie attribut d'accès> ::= *(1)*
GETASSOCIATION (*<locus transfert>*) *(1.1)*
| *GETUSAGE* (*<locus transfert>*) *(1.2)*
| *OUTOFFILE* (*<locus transfert>*) *(1.3)*

sémantique: *GETASSOCIATION* renvoie une valeur référence au locus association auquel le locus accès dénoté par le *locus transfert* est connecté; il renvoie *NULL* si le locus accès n'est pas connecté à une association.

GETUSAGE renvoie la valeur de l'attribut **usage**, c'est-à-dire *READONLY (WRITEONLY)* si le locus accès n'est connecté que pour des opérations lire (écrire), ou *READWRITE* si le locus accès est connecté pour des opérations lire et écrire.

OUTOFFILE renvoie la valeur de l'attribut **hors du fichier** du locus accès, c'est-à-dire *TRUE* si la dernière opération lire a calculé un indice de **transfert** qui n'était pas dans le fichier, sinon *FALSE*.

propriétés statiques: la classe d'un appel de routine prédéfinie *GETASSOCIATION* est la *ASSOCIATION*-classe par référence. La **régionalité** d'un appel de routine prédéfinie *GETASSOCIATION* est celle du *locus transfert*.

La classe d'un appel de routine prédéfinie *OUTOFFILE* est la *BOOL*-classe par dérivation.

La classe d'un appel de routine prédéfinie *GETUSAGE* est la *USAGE*-classe par dérivation.

conditions dynamiques: *GETUSAGE* et *OUTOFFILE* causent l'exception *NOTCONNECTED* si le locus accès n'est pas connecté à une association.

exemple:

21.47 OUTOFFILE (infiles (FALSE)) *(1.3)*

7.4.9 Opérations de transfert de données

syntaxe:

<appel de routine prédéfinie lire article> ::= *(1)*
READRECORD (*<locus accès>* [, *<expression indice>*]
[, *<locus de lecture>*]) *(1.1)*

<appel de routine prédéfinie écrire article> ::= *(2)*
WRITERECORD (*<locus accès>* [, *<expression indice>*],
<expression écrire>) *(2.1)*

<locus de lecture> ::= *(3)*
<locus de mode statique> *(3.1)*

<expression écrire> ::= *(4)*
<expression> *(4.1)*

NOTE – Si le *locus accès* a un mode **indice**, on résout l'ambiguïté syntaxique en interprétant le second argument comme une *expression indice* et non comme un *locus de lecture*.

sémantique: pour le transfert de données à un fichier ou à partir de celui-ci, les routines prédéfinies *WRITERECORD* et *READRECORD* sont définies. Le *locus accès* doit avoir un mode **enregistrement** et il doit être connecté à une association pour transférer des données à un fichier rattaché à cette association ou à partir de celui-ci. La direction du transfert ne doit pas être en contradiction avec la valeur de l'attribut **usage** du *locus accès*.

L'indice de **transfert**, c'est-à-dire la position dans le fichier de l'enregistrement à transférer, est calculé avant que le transfert soit effectué. Si le *locus accès* n'a pas de mode **indice**, l'indice de **transfert** est l'indice **actuel** augmenté de 1; si le *locus accès* a un mode **indice**, l'indice de **transfert** est calculé comme suit:

indice de **transfert** := indice de **base** + *NUM (v)* – *NUM (l)* + 1

où *l* est la **borne inférieure** du mode **indice** du *locus accès* et *v* dénote la valeur donnée par *l'expression indice*. Si le transfert de l'enregistrement portant l'indice de **transfert** calculé a été exécuté avec succès, l'indice **actuel** devient l'indice de **transfert**.

Les opérations lire:

READRECORD transfère au programme CHILL des données d'un fichier du monde extérieur.

Si l'indice de **transfert** calculé n'est pas dans le fichier, l'attribut **hors du fichier** est fixé sur *TRUE*, sinon, le fichier est positionné, l'enregistrement mis en lecture et l'attribut **hors du fichier** est fixé sur *FALSE*.

L'enregistrement en lecture ne doit pas donner une valeur **indéfinie**; l'effet de l'opération lire est défini par l'implémentation si l'enregistrement du fichier mis en lecture n'a pas une valeur correcte conformément au mode **enregistrement** du *locus accès*.

Si le *locus de lecture* est spécifié, alors, la valeur de l'enregistrement qui a été lu est affectée à ce locus. Si aucun *locus de lecture* n'est spécifié, la valeur sera affectée à un locus créé implicitement; la durée de vie de ce locus se termine lorsque le *locus accès* est déconnecté ou reconnecté. On ne précise pas si le locus référencé est créé une seule fois par l'opération de connexion ou chaque fois qu'une opération lire est exécutée.

READRECORD renvoie dans les deux cas une valeur référence qui se rapporte au locus (de mode éventuellement dynamique) auquel la valeur est affectée.

Si l'attribut **hors du fichier** est fixé sur *TRUE* comme résultat de l'appel de routine prédéfinie, la valeur *NULL* est envoyée comme résultat de l'appel.

Les opérations écrire:

WRITERECORD transfère des données d'un programme CHILL à un fichier du monde extérieur. Le fichier est positionné sur l'enregistrement portant l'indice calculé et l'enregistrement est écrit.

Une fois l'enregistrement écrit, le nombre d'enregistrements est fixé sur l'indice de **transfert**, si ce dernier est supérieur au nombre effectif d'enregistrements.

L'enregistrement écrit par *WRITERECORD* est la valeur donnée par *l'expression écrire*.

propriétés statiques: la classe de la valeur lue par *READRECORD* est la M-classe par valeur, où M est le mode **enregistrement** du *locus accès*, s'il a un mode **enregistrement statique**, ou une version paramétrée dynamiquement, si le locus a un mode **enregistrement dynamique**; les paramètres de cet enregistrement paramétré dynamiquement sont les suivants:

- la **longueur de chaîne** dynamique de la valeur chaîne lue dans le cas d'un mode chaîne;
- la **borne supérieure** dynamique de la valeur matrice lue dans le cas d'un mode matrice;
- la liste de valeurs (étiquette) associées au mode de la valeur structure lue dans le cas d'un mode structure **variable**.

La classe de l'appel de routine prédéfinie *READRECORD* est la M-classe par référence si le *locus de lecture* n'est pas spécifié, sinon c'est la S-classe par référence où S est le mode du *locus de lecture*.

La **régionalité** d'un appel de routine prédéfinie *READRECORD* est celle du *locus de lecture* s'il est spécifié, sinon celle du *locus accès*.

conditions statiques: le *locus accès* doit avoir un mode **enregistrement**.

Une *expression indice* peut ne pas être spécifiée si le *locus accès* n'a pas de mode **indice** et doit être spécifiée si le *locus accès* a un mode **indice**. La classe de la valeur donnée par *l'expression indice* doit être **compatible** avec ce mode **indice**.

Le *locus de lecture* doit être **référençable**.

Le mode du *locus de lecture* ne doit pas avoir la **propriété de protection**.

Si le *locus de lecture* est spécifié, le mode du *locus de lecture* doit être l'**équivalent** du mode **enregistrement** du *locus accès*, s'il a un mode **enregistrement statique** ou un mode **enregistrement** chaîne **variable**, sinon d'une version paramétrée dynamiquement de celui-ci; les paramètres d'un tel mode paramétré dynamiquement sont ceux de la valeur qui a été lue.

La classe de la valeur donnée par *l'expression écrire* doit être **compatible** avec le mode **enregistrement** du *locus accès*, s'il a un mode **enregistrement statique** ou un mode **enregistrement** chaîne **variable**; sinon il doit y avoir une version paramétrée dynamiquement du mode **enregistrement** qui soit **compatible** avec la classe de l'*expression écrire*. Les conditions d'affectation de la valeur de l'*expression écrire* par rapport au mode précité s'appliquent.

conditions dynamiques: les exceptions *RANGEFAIL* ou *TAGFAIL* ont lieu si la partie dynamique de la vérification de compatibilité précitée échoue.

Les appels de routine prédéfinie *READRECORD* et *WRITERECORD* causent l'exception *NOTCONNECTED* si le *locus accès* n'est pas connecté à une association.

Les appels de routine prédéfinie *READRECORD* ou *WRITERECORD* causent l'exception *RANGEFAIL* si le mode **indice** du *locus accès* est un mode intervalle et si *l'expression indice* donne une valeur extérieure aux bornes de ce mode intervalle.

L'appel de routine prédéfinie *READRECORD* cause l'exception *READFAIL* si l'une des conditions suivantes se vérifie:

- la valeur de l'attribut **usage** est *WRITEONLY*;
- la valeur de l'attribut **hors du fichier** est *TRUE* et le *locus accès* est connecté pour les opérations de lecture séquentielle;
- la lecture de l'enregistrement ayant l'indice calculé échoue en raison de conditions du monde extérieur.

L'appel d'opération prédéfinie *WRITERECORD* cause l'exception *WRITEFAIL* si et seulement si l'une des conditions suivantes se vérifie:

- la valeur de l'attribut **usage** est *READONLY*;
- l'opération d'écriture de l'enregistrement portant l'indice calculé échoue, en raison de conditions du monde extérieur.

Si l'exception *RANGEFAIL* ou l'exception *NOTCONNECTED* se produit, alors, elle se présente avant que la valeur de tout attribut soit modifiée et avant que le fichier soit positionné.

exemples:

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 Entrée/sortie de texte

7.5.1 Généralités

Les opérations de sortie de texte permettent de représenter les valeurs CHILL sous une forme accessible en lecture par l'homme: les facilités d'entrée de texte assurent les conversions inverses.

Les opérations de transfert de texte sont définies en plus du modèle d'entrée/sortie CHILL de base et s'appliquent à des fichiers dont l'accès peut se faire de façon séquentielle ou aléatoire et dont les enregistrements peuvent avoir une longueur fixe ou variable.

Le modèle suppose qu'à chaque enregistrement est attachée une information de positionnement (éventuellement vide), souvent référencée dans les implémentations comme des caractères de commande de chariot ou de commande.

La manipulation d'un fichier de texte dans le CHILL exige une association; le transfert de données à partir d'un fichier de texte et vers celui-ci exige qu'un locus **texte** soit connecté à une association pour ce fichier.

Les opérations de transfert de texte peuvent s'appliquer aux valeurs CHILL qui peuvent devenir des enregistrements d'un fichier de texte, ainsi qu'aux locus CHILL qui ne sont pas nécessairement liés à une activité d'entrée/sortie quelconque du programme.

La possibilité de retrouver dans un morceau de texte les valeurs CHILL d'origine ne peut pas être garantie d'une manière générale; elle dépend de la représentation qui a été utilisée.

Les valeurs de texte sont contenues dans des locus du mode texte. Un locus texte est nécessaire au transfert de données sous une forme accessible en lecture par l'homme.

Les valeurs de texte n'ont pas de dénotation mais sont contenues dans des locus du mode texte; il n'y a pas d'expression dénotant une valeur du mode texte. Les valeurs de texte peuvent seulement être manipulées par des routines prédéfinies qui prennent un locus texte pour paramètre.

7.5.2 Attributs des valeurs de texte

Les valeurs de texte ont des attributs qui en décrivent les propriétés dynamiques. Les attributs suivants sont définis:

- **indice effectif**: indique la prochaine position de caractère de l'**enregistrement de texte** à lire ou à écrire. Il a un mode qui est ***RANGE*** *(0:L–1)*, où *L* est la **longueur de texte** du mode de la valeur. Il est initialisé à 0 quand un locus texte est créé;
- **référence d'enregistrement de texte**: indique une valeur référence au sous-locus de l'**enregistrement de texte** du locus texte. Il a un mode qui est **REF** *M*, où *M* est le mode **enregistrement de texte** du mode de la valeur;
- **référence d'accès**: indique une valeur référence au sous-locus **accès** du locus texte. Il a un mode qui est **REF** *M*, où *M* est le mode **accès** du mode de la valeur.

7.5.3 Opérations de transfert de texte

syntaxe:

<appel de routine prédéfinie de texte> ::= *(1)*
READTEXT (*<liste d'arguments d'e/s de texte>*) *(1.1)*
| *WRITETEXT* (*<liste d'arguments d'e/s de texte>*) *(1.2)*

<liste d'arguments d'e/s de texte> ::= *(2)*
<argument de texte> [, *<expression indice>*] ,
<argument de format> [, *<liste d'e/s>*] *(2.1)*

<*argument de texte*> ::= (3)
<*locus texte*> (3.1)
| <*locus chaîne de caractères*> (3.2)
| <*expression chaîne de caractères*> (3.3)

<*argument de format*> ::= (4)
<*expression chaîne de caractères*> (4.1)

<*liste d'e/s*> ::= (5)
<*élément de liste d'e/s*> { , <*élément de liste d'e/s*> }* (5.1)

<*élément de liste d'e/s*> ::= (6)
<*argument de valeur*> (6.1)
| <*argument de locus*> (6.2)

<*argument de locus*> ::= (7)
<*locus discret*> (7.1)
<*locus virgule flottante*> (7.2)
| <*locus chaîne*> (7.3)

<*argument de valeur*> ::= (8)
<*expression discrète*> (8.1)
<*expression à virgule flottante*> (8.2)
| <*expression chaîne*> (8.3)

NOTE – Si l'*élément de liste d'e/s* est un locus, on résout l'ambiguïté en interprétant l'*élément de liste d'e/s* comme un *argument de locus* et non comme un *argument de valeur*.

sémantique: *READTEXT* applique les fonctions de conversion, d'édition et de commande d'e/s contenues dans l'*argument de format* à l'**enregistrement de texte** désigné par l'*argument de texte*; ceci (éventuellement) produit une liste de valeurs qui sont attribuées aux éléments de la *liste d'e/s* dans l'ordre dans lequel elles sont spécifiées. *WRITETEXT* est l'opération inverse. Aucune opération implicite d'e/s n'est effectuée.

Si l'*argument de texte* est un *locus chaîne de caractères* ou une *expression chaîne de caractères*, les fonctions d'édition et de conversion sont appliquées sans aucune relation avec le monde extérieur. Dans ce cas, l'**indice effectif** désigne un locus qui est implicitement créé au début de l'appel de routine prédéfinie et initialisé à 0. L'**enregistrement de texte** est la chaîne de caractères désignée par le *locus chaîne de caractères* ou *l'expression chaîne de caractères* et la **longueur de texte** est sa **longueur de chaîne**.

Les éléments de la *liste d'e/s* peuvent être:

- des *arguments de valeur* et des *arguments de locus*;
- des *largeurs de clause* **variables** comme décrit ci-dessous.

Relations entre un argument de format et une liste d'e/s

La valeur donnée par un *argument de format* doit avoir la forme d'une *chaîne de commande de format* (voir 7.5.4).

Pendant l'exécution d'un appel de routine prédéfinie d'e/s, la *chaîne de commande de format* (voir 7.5.4) désignée par l'*argument de format* et la *liste d'e/s* sont explorées de gauche à droite. Chaque occurrence d'un *texte de format* et d'une *spécification de format* est interprétée et l'action appropriée est accomplie ainsi:

a) *texte de format*

Dans *READTEXT*, l'**enregistrement de texte** doit contenir à la position d'**indice effective** un segment de chaîne égal à la chaîne livrée par le *texte de format*. Dans *WRITETEXT*, la chaîne livrée par le *texte de format* est transférée à l'**enregistrement de texte**. La sémantique est la même qu'en cas de rencontre d'une *spécification de format* qui est *%C* et d'un *élément de liste d'e/s* qui livre la même valeur de chaîne que celle livrée par le *texte de format*.

b) *spécification de format*

Si la *spécification de format* contient un *facteur de répétition*, elle est équivalente à une séquence d'un nombre d'occurrences d'*éléments de format* égal au nombre désigné par le *facteur de répétition*.

Si la *spécification de format* est une *clause de format*, elle contient un *code de commande*. Si le *code de commande* est une *clause de conversion*, un *élément de liste d'e/s* est extrait de la *liste d'e/s* et la fonction de conversion choisie par le *code de conversion*, les *qualificateurs de conversion* et la *largeur de clause* lui est appliquée (voir 7.5.5). Si le *code de commande* est une *clause d'édition* ou une *clause d'e/s*, la fonction d'édition ou d'e/s choisie par le *code d'édition* ou le *code d'e/s* et la *largeur de clause* est appliquée à l'*argument de texte* sans référence à la *liste d'e/s* (voir 7.5.6 et 7.5.7).

Si la *largeur de clause* est **variable**, une valeur est extraite de la liste, ce qui désigne le paramètre **largeur** de la fonction de commande de conversion ou d'édition.

Si la *spécification de format* est une *clause parenthésée*, la *chaîne de commande de format* qui y est contenue est explorée.

L'interprétation de la *chaîne de commande de format* prend fin quand est atteinte la fin de la chaîne livrée par la *chaîne de commande de format*.

Les *éléments de la liste d'e/s* sont explorés dans l'ordre dans lequel ils sont spécifiés.

conditions statiques: si l'*argument de texte* est un *locus chaîne*, son mode doit être un mode chaîne **variable**.

Une *expression indice* peut ne pas être spécifiée si l'*argument de texte* n'est pas un *locus texte* ou s'il l'est et son mode **accès** n'a pas de mode **indice** et doit être spécifié si le mode **accès** a un mode **indice**; la classe de la valeur livrée par l'*expression indice* doit être **compatible** avec ce mode **indice**.

Un *argument de texte* dans un appel de routine prédéfinie *WRITETEXT* doit être un locus.

Un *locus chaîne* dans un *argument de texte* doit être **référençable**.

conditions dynamiques: l'exception *TEXTFAIL* se produit si:

- la valeur chaîne livrée par l'*argument de format* ne peut pas être obtenue comme une production terminale de la *chaîne de commande de format;*
- intervient une tentative d'affecter à l'**indice effectif** une valeur inférieure à 0 ou supérieure à la **longueur de texte**;
- pendant l'interprétation, la fin de la *chaîne de commande de format* a été atteinte et si la *liste d'e/s* n'est pas complètement explorée, ou si d'autres éléments ne peuvent plus être extraits de la *liste d'e/s* et la *chaîne de commande de format* contient d'autres *codes de conversion* ou *largeurs de clause* **variables**;
- une *clause d'e/s* est rencontrée et si l'*argument de texte* n'est pas un *locus texte*;
- un *texte de format* est rencontré dans *READTEXT* et **l'enregistrement de texte** ne contient pas à la position **d'indice effective** une chaîne égale à celle qui est livrée par le *texte de format*.

Une exception définie pour l'appel de routine prédéfinie *READRECORD* et *WRITERECORD* peut se produire si une fonction de commande d'e/s est exécutée et si l'une des conditions dynamiques définies n'est pas respectée.

exemple:

26.18 WRITETEXT (output," % B%/",10) *(1.2)*

7.5.4 Chaîne de commande de format

syntaxe:

<chaîne de commande de format> ::= *(1)*
[*<texte de format>*] { *<spécification de format>* [*<texte de format>*] }* *(1.1)*

<texte de format> ::= *(2)*
{ *<caractère non-pour cent>* | *<pour cent>* }* *(2.1)*

<pour cent> ::= *(3)*
% % *(3.1)*

<spécification de format> ::= *(4)*
% [*<facteur de répétition>*] *<élément de format>* *(4.1)*

<facteur de répétition> ::= *(5)*
{ *<chiffre>* }$^{+}$ *(5.1)*

<élément de format> ::= *(6)*
<clause de format> *(6.1)*
| *<clause parenthésée>* *(6.2)*

<clause de format> ::= *(7)*
<code de commande> [*%.*] *(7.1)*

<code de commande> ::= *(8)*
<clause de conversion> *(8.1)*
| *<clause d'édition>* *(8.2)*
| *<clause d'e/s>* *(8.3)*

<clause parenthésée> ::= *(9)*
(<chaîne de commande de format> %) *(9.1)*

NOTE – Le premier caractère qui ne peut pas faire partie de *l'élément de format* met fin à la *spécification de format*. Les espaces et les commandes de mise en page peuvent ne pas être utilisés dans les *éléments de format*. On peut utiliser un point (.) pour mettre fin à une *clause de format*. Il appartient à la *clause de format* et n'a qu'un effet de délimitation. Pour représenter le caractère de pourcentage (%) dans un *texte de format*, il doit être écrit deux fois (*%%*).

sémantique: une *chaîne de commande de format* spécifie la forme externe des valeurs transférées et l'implantation des données dans les enregistrements. Une *chaîne de commande de format* se compose d'occurrences de *texte de format*, qui désignent les parties fixes des enregistrements et d'occurrences de *spécification de format*, qui désignent les représentations externes des valeurs CHILL, permettant l'édition de l'**enregistrement de texte** ou la commande des opérations d'e/s effectives.

Une *spécification de format* qui contient un *facteur de répétition* et une *clause de format* équivaut à autant d'occurrences de *spécification de format* identiques pour la *clause de format* que le *facteur de répétition*. Un *facteur de répétition* peut être égal à 0, auquel cas la *spécification de format* n'est pas examinée. Par exemple "*%3C4*" équivaut à "*%C4%C4%C4*".

La notation décimale est supposée pour les *chiffres* d'un *facteur de répétition*.

Une *chaîne de commande de format* dans une *clause parenthésée* est explorée à plusieurs reprises en fonction du *facteur de répétition*. Si aucun n'est spécifié, *1* est supposé par défaut.

exemple:

26.20 size = %C%/ *(1.1)*

7.5.5 Conversion

syntaxe:

<clause de conversion> ::= *(1)*
<code de conversion> { *<qualificatif de conversion>* }*
[*<largeur de clause>*] *(1.1)*

<code de conversion> ::= *(2)*
B | O | H | C | F *(2.1)*

<qualificatif de conversion> ::= *(3)*
L | E | P <caractère> *(3.1)*

<largeur de clause> ::= *(4)*
{ {*<chiffre>* }* | *V* } [*<largeur fractionnaire>*] [*<largeur d'exposant>*] *4.1)*

<largeur fractionnaire> ::= *(5)*
. { *<chiffre>* }+ *(5.1)*

<largeur d'exposant> ::= *(6)*
: { *<chiffre>* }+ *(6.1)*

syntaxe dérivée: une *clause de conversion* dans laquelle une *largeur de clause* n'est pas présente est une syntaxe dérivée pour une *clause de conversion* dans laquelle une *largeur de clause* qui est 0 est spécifiée.

sémantique: une conversion dans un appel de routine prédéfinie *READTEXT* transforme une chaîne qui est une représentation externe en une valeur CHILL. Une conversion dans un appel de routine prédéfinie *WRITETEXT* accomplit la transformation inverse. Le *code de conversion* avec le *qualificatif de conversion* spécifient le type de conversion et les détails de l'opération demandée, comme la justification, le traitement de débordement et le remplissage.

La représentation externe est une chaîne dont la longueur dépend en général de la valeur à convertir. Cette chaîne peut contenir le nombre minimal de caractères nécessaires pour représenter la valeur CHILL (format libre) ou peut avoir une longueur donnée (format fixe).

Dans le format fixe, un segment de taille **largeur** commençant à la position d'**indice effective** est lu à partir d'un **enregistrement de texte** ou écrit dans un tel enregistrement selon la justification et le remplissage choisis par les *qualificatifs de conversion*, comme suit:

- dans *READTEXT*: tous les caractères de remplissage (à gauche ou à droite, selon la justification) s'il y en a, sont enlevés. Néanmoins, quand des caractères ou des chaînes de caractères **fixes** sont lus, le nombre maximal *N* de caractères de remplissage qui sont enlevés est *L* – **largeur**, où *L* est 1 ou la **longueur de chaîne**, respectivement. Aucun caractère n'est enlevé si $N < 0$. Les caractères restants sont pris comme la représentation externe;
- dans *WRITETEXT*: si la longueur de la représentation externe est inférieure ou égale à **largeur**, les caractères sont justifiés vers la gauche ou vers la droite dans le segment (selon la justification). Les éléments de chaîne inutilisés, le cas échéant, sont remplis avec le caractère de remplissage. Sinon, la chaîne est tronquée (à gauche si la justification à droite a été choisie, sinon à droite), ou des caractères indiquant le "débordement" de **largeur** (∗) sont transférés, si le qualificatif *E* est présent. La troncation est appliquée à la représentation externe, y compris le signe moins, le point (.) et le qualificatif *E* (notation scientifique) le cas échéant.

Dans le format libre, on a:

- dans *READTEXT*: les caractères de remplissage, s'il y en a, sont omis, sauf quand un caractère ou une chaîne de caractères est lu et que le *qualificatif de conversion P* n'est pas spécifié. Ensuite, la représentation externe est prise comme le plus long segment de caractères qui commence à l'**indice effectif** et qui est constitué de tous les caractères subséquents qui peuvent lexicalement lui appartenir comme défini ci-dessous;
- dans *WRITETEXT*: la chaîne livrée par la conversion est insérée en commençant par la position d'**indice effective**.

Dans *WRITETEXT*, la chaîne qui est la représentation externe est transférée à l'**enregistrement de texte** sans considération de sa **longueur effective**. Après le transfert, l'**indice effectif** est automatiquement avancé à la prochaine position de caractère disponible et la **longueur effective** est fixée à la valeur maximale entre l'**indice effectif** et (l'ancienne) **longueur effective**.

Une *largeur de clause* est **constante** si elle est composée de *chiffres*. On suppose une notation décimale. Sinon, elle est **variable**.

Si la **largeur** est zéro, le format libre est choisi, sinon la **largeur** est la longueur du format fixe.

Si la **largeur** est trop petite pour contenir la chaîne, l'action appropriée est accomplie en fonction du *qualificatif de conversion*.

Dans un *READTEXT*, la représentation externe qui est appliquée est celle définie ci-dessous pour le mode de *l'argument de locus*.

Dans un *WRITETEXT*, la représentation externe qui est appliquée est celle définie ci-dessous pour le mode M de la M-classe par valeur ou par dérivation de la valeur donnée par *l'argument de valeur*.

Codes de conversion

Les *codes de conversion* sont représentés comme des lettres simples. Les *codes de conversion* suivants sont définis:

B: représentation binaire.

O: représentation octale.

H: représentation hexadécimale.

C: conversion: indique la représentation externe par défaut des valeurs CHILL, qui dépend du mode de la valeur à convertir (voir plus loin).

F: représentation scientifique, c'est-à-dire la représentation de valeurs à virgule flottante au moyen d'une mantisse et d'un exposant.

La représentation externe dépend du *code de conversion* et du mode de la valeur à convertir.

Qualificatifs de conversion

Les *qualificatifs de conversion* sont représentés comme des lettres simples. Les *qualificatifs de conversion* suivants sont définis:

L: justification à gauche. La justification à droite est supposée si le qualificatif n'est pas présent. Dans le format libre, le qualificatif n'a aucun effet.

E: preuve de débordement. Dans *WRITETEXT*, l'indication de débordement est choisie: si le qualificatif n'est pas présent, la troncation a lieu. Dans *READTEXT* ou dans le format libre, ce qualificatif n'a aucun effet.

P: remplissage. Le caractère qui suit le qualificatif spécifie le caractère de remplissage. Si *P* n'est pas présent, le caractère de remplissage est supposé être espace par défaut. Dans *READTEXT*, si le format libre est choisi, les espaces et HT (tabulation horizontale) sont considérés comme étant le même caractère pour les besoins de l'omission, en cas de spécification après le qualificatif ou d'application par défaut.

Représentation externe

La représentation externe des valeurs CHILL est définie ainsi:

a) *entiers*

Les valeurs entières sont représentées lexicalement comme un ou plusieurs chiffres dans une base décimale par défaut non précédés de zéros et précédés d'un signe si négatifs. Le signe plus et les zéros qui précèdent sont ignorés dans *READTEXT*. Les *codes de conversion* suivants sont disponibles: *B, O, C* et *H*. Le *code de conversion C* choisit la représentation décimale. Les chiffres qui peuvent appartenir à la représentation sont seulement ceux qui sont choisis par le code de conversion.

b) *valeurs à virgule flottante*

Les valeurs à virgule flottante peuvent être représentées de deux façons:

- en virgule fixe (sélectionnée par *code de conversion C*);
- en notation scientifique (sélectionnée par *code de conversion F*).

La représentation en virgule fixe de la valeur à virgule flottante est lexicalement représentée par une séquence constituée d'un ou de plusieurs chiffres (la partie entière) suivie d'une séquence facultative d'un ou de plusieurs chiffres (la partie fractionnaire) séparée de la partie entière par une virgule (,). Si la valeur est négative, elle est précédée du signe "moins".

En notation scientifique, la valeur à virgule flottante est représentée par une mantisse et un exposant. La mantisse est lexicalement représentée comme une valeur à virgule fixe dont la partie entière est constituée d'un seul chiffre supérieur à zéro. L'exposant est lexicalement représenté par la lettre *E* suivie d'un signe éventuel et d'une séquence d'un ou de plusieurs chiffres. Dans les deux représentations, le signe "plus" et les zéros qui précèdent la valeur sont ignorés par *READTEXT*.

En présence de *largeur fractionnaire*, la valeur donnée par les *chiffres* qu'elle contient est la longueur de la partie fractionnaire complétée des zéros à droite si nécessaire; sinon la partie fractionnaire contient le nombre minimal de chiffres nécessaires pour la représenter.

En présence de *largeur d'exposant*, la valeur donnée par les *chiffres* qu'elle contient indique le nombre minimal de chiffres à utiliser pour représenter l'exposant, y compris les zéros à gauche si nécessaire; sinon, la valeur par défaut "3" est adoptée.

Les *codes de conversion* suivants sont disponibles: *C, F*.

c) *booléens*

Les valeurs booléennes sont représentées lexicalement comme une *chaîne de nom simple*, qui sont *TRUE* et *FALSE* [en majuscules (par exemple, *TRUE*) ou minuscules (par exemple, *true*) selon la représentation choisie par l'implémentation pour les chaînes de nom simple **spéciales**]. Le *code de conversion* suivant est disponible: *C*.

d) *caractères*

Les valeurs de caractère sont représentées lexicalement comme des chaînes de longueur *1*. Le *code de conversion* suivant est disponible: *C*.

e) *ensembles*

Les valeurs de mode ensemble sont représentées lexicalement comme des chaînes de nom simple, qui sont les littéraux d'ensemble. Le *code de conversion* suivant est disponible: *C*.

f) *intervalles*

Les valeurs d'intervalle ont la même représentation que les valeurs de leur mode **racine**. Cependant, seules les représentations des valeurs définies par le mode intervalle discret ou le mode intervalle en virgule flottante appartiennent à l'ensemble de représentations externes associées au mode intervalle discret ou au mode intervalle en virgule flottante.

g) *chaînes de caractères*

Les valeurs de chaîne de caractères sont représentées lexicalement comme des chaînes de caractères de longueur *L*. Dans *WRITETEXT*, *L* est la **longueur effective**. Dans *READTEXT*, *L* est la **longueur de chaîne** si la chaîne est une chaîne **fixe**, sinon c'est une chaîne **variable** et *L* est la **longueur de chaîne**, sauf s'il y a moins de caractères disponibles dans (le segment) d'**enregistrement de texte** à la position d'**indice effective**, auquel cas *L* est le nombre de caractères disponibles. Le *code de conversion* suivant est disponible: *C*.

h) *chaînes binaires*

Les valeurs de chaîne binaires sont représentées lexicalement comme des chaînes de chiffres binaires. Les mêmes règles que pour les chaînes de caractères sont appliquées pour déterminer le nombre de chiffres. Le *code de conversion* suivant est disponible: *C*.

propriétés dynamiques: une *largeur de clause* a une **largeur**, qui est la valeur livrée par *chiffre* ou par une valeur de la *liste d'e/s* si la *largeur de clause* est **variable**; en l'absence de spécification, cette largeur est nulle.

conditions dynamiques: l'exception *TEXTFAIL* se produit si:

- dans *READTEXT*, l'**enregistrement de texte** ne contient pas de segment de chaîne commençant à l'**indice effectif** qui peut (après enlèvement ou omission des caractères de remplissage, voir ci-dessus) être interprété comme une représentation externe de l'une des valeurs du mode de l'*argument de locus* actuel (y compris une tentative de lire une représentation externe non vide à partir d'un **enregistrement de texte** quand **indice effectif** = **longueur effective**),
- dans *WRITETEXT*, un segment de chaîne qui est la représentation externe de l'*argument de valeur* actuel ne peut pas être transféré à l'**enregistrement de texte** commençant à l'**indice effectif**;
- dans *READTEXT*, un *code de conversion* est rencontré et l'élément actuel dans la *liste d'e/s* n'est pas un locus, ou le mode du locus a la **propriété de protection**;
- le même *qualificateur de conversion* est spécifié plus d'une fois;
- une *largeur de clause* **variable** est rencontrée et l'*élément de liste d'e/s* correspondant dans la *liste d'e/s* n'a pas une classe entière ou est inférieur à 0;
- une *largeur de clause* a une *largeur fractionnaire* ou une *largeur d'exposant*; l'*élément de liste d'e/s* correspondant de la *liste d'e/s* n'a pas de classe à virgule flottante ou a une *largeur d'exposant* et un *code de conversion* autre que *F*.

exemple:

26.21 CL6 *(1.1)*

7.5.6 Edition

syntaxe:

<clause d'édition> ::= *(1)*
<code d'édition> [*<largeur de clause>*] *(1.1)*

<code d'édition> ::= *(2)*
X | < | > | *T* *(2.1)*

syntaxe dérivée: une *clause d'édition* dans laquelle une *largeur de clause* n'est pas présente est la syntaxe dérivée pour une *clause d'édition* dans laquelle une *largeur de clause* qui est *1* est spécifiée si le *code d'édition* n'est pas *T*, sinon 0 respectivement.

sémantique: les fonctions d'édition suivantes sont définies:

X: espace: des espaces de **largeur** de clause sont insérés ou omis.

>: omettre à droite: l'**indice effectif** est déplacé vers la droite de la position **largeur** de clause.

<: omettre à gauche: l'**indice effectif** est déplacé vers la gauche de la position **largeur** de clause.

T: tabulation: l'**indice effectif** est déplacé sur la position **largeur** de clause.

Dans *WRITETEXT*, si l'**indice effectif** est déplacé sur une position plus grande que la **longueur effective**, une chaîne de *N* caractères espace, où *N* est la différence entre l'**indice effectif** et l'(ancienne) **longueur effective** est ajoutée à l'**enregistrement de texte**. La **longueur effective** est fixée à la valeur maximale entre l'**indice effectif** et l'(ancienne) **longueur effective**.

conditions dynamiques: l'exception *TEXTFAIL* se produit si:

- l'**indice effectif** est déplacé sur une position inférieure à 0 ou supérieure à la **longueur de texte**;
- dans *READTEXT*, l'**indice effectif** est déplacé sur une position qui est supérieure à la **longueur effective**;
- dans *READTEXT*, le *code d'édition X* est spécifié et une chaîne de largeur espaces ou de caractères HT (tabulation horizontale) n'est pas présente dans l'**enregistrement de texte** à la position d'**indice effective**.

exemple:

26.22 X *(1.1)*

7.5.7 Commande d'e/s

syntaxe:

<clause d'e/s> ::= *(1)*
<code d'e/s> *(1.1)*

<code d'e/s> ::= *(2)*
/ | – | + | ? | ! | = *(2.1)*

sémantique: les fonctions de commande d'e/s (sauf %=) effectuent une opération d'e/s. Elles permettent un contrôle précis du transfert de l'**enregistrement de texte**. Dans *READTEXT*, toutes les fonctions ont le même effet, lire le prochain enregistrement du fichier. Dans *WRITETEXT*, l'**enregistrement de texte** et la représentation appropriée de l'information de commande chariot sont transférés. La position initiale du chariot au moment où le *locus texte* est connecté est telle que le premier caractère du premier **enregistrement de texte** est imprimé au début de la première ligne inoccupée (indépendamment de toute information éventuelle de positionnement rattachée à l'**enregistrement de texte**).

La manière de placer le chariot est décrite au moyen des opérations abstraites suivantes sur la colonne ligne et page actuelles (x, y, z), les colonnes étant considérées comme étant numérotées à partir de zéro et à partir de la marge de gauche et les lignes à partir de zéro et à partir de la marge supérieure.

nl(w): le chariot est déplacé w lignes plus bas, au début de la ligne (nouvelle position: $(0, (y + w) \bmod p, z + (y + w)/p$, où p est le nombre de lignes par page));

np(w): le chariot est déplacé w pages plus bas, au début de la ligne (nouvelle position: $(0, 0, z + w)$).

Les fonctions de commande suivantes sont fournies:

/: prochain enregistrement: l'enregistrement est imprimé sur la prochaine ligne (nl(1), imprimer enregistrement, nl(0));

+: prochaine page: l'enregistrement est imprimé en haut de la prochaine page (np(1), imprimer enregistrement, nl(0));

–: ligne actuelle: l'enregistrement est imprimé sur la ligne actuelle (imprimer l'enregistrement, nl(0));

?: incitation: l'enregistrement est imprimé sur la prochaine ligne. Le chariot est laissé à la fin de la ligne (nl(1), imprimer enregistrement);

!: émettre: aucune commande de chariot n'est effectuée (imprimer enregistrement);

=: fin de page: définit la position du prochain enregistrement, s'il y en a un, en haut de la prochaine page (cela a priorité sur le positionnement effectué avant l'impression de l'enregistrement). Cela ne cause aucune opération d'e/s.

Le transfert d'E/S est effectué comme suit:

- dans *READTEXT*, la sémantique est comme si était exécuté un *READRECORD* (*A, I, R*), où *A* est le sous-locus **accès** du *locus texte*, *I* est l'*expression indice* (le cas échéant) et *R* désigne l'**enregistrement de texte**. Après le transfert d'E/S, l'**indice effectif** est mis sur 0 et la **longueur effective** sur la **longueur de chaîne** de la valeur de chaîne qui a été lue;
- dans *WRITETEXT*, la sémantique est comme si était exécuté un *WRITERECORD* (*A, I, R*), où *A* est le sous-locus **accès** du *locus texte*, *I* est l'*expression indice* (le cas échéant) et *R* désigne l'**enregistrement de texte**. L'information de position associée est également transférée. Si le mode **enregistrement** de l'accès n'est pas **dynamique**, l'**enregistrement de texte** est rempli à la fin avec des caractères espace et sa **longueur effective** est fixée sur la **longueur de texte** avant que le transfert n'ait lieu. Après le transfert d'E/S, l'**indice effectif** et la **longueur effective** sont mis sur 0.

exemple:

26.21 / *(1.1)*

7.5.8 Accès aux attributs d'un locus texte

syntaxe:

<appel de routine prédéfinie obtenir texte> ::= *(1)*
GETTEXTRECORD (*<locus texte>*) *(1.1)*
| *GETTEXTINDEX* (*<locus texte>*) *(1.2)*
| *GETTEXTACCESS* (*<locus texte>*) *(1.3)*
| *EOLN* (*<locus texte>*) *(1.4)*

<appel de routine prédéfinie fixer texte> ::= *(2)*
SETTEXTRECORD (*<locus texte>* , *<locus chaîne de caractères>*) *(2.1)*
| *SETTEXTINDEX* (*<locus texte>* ,*<expression entière>*) *(2.2)*
| *SETTEXTACCESS* (*<locus texte>* , *<locus accès>*) *(2.3)*

sémantique: *GETTEXTRECORD* renvoie la **référence d'enregistrement de texte** de *locus texte*.

GETTEXTINDEX renvoie l'**indice effectif** du *locus texte*.

GETTEXTACCESS renvoie la **référence d'accès** du *locus texte*.

EOLN donne *TRUE* s'il n'y a plus de caractères disponibles dans l'**enregistrement de texte** (c'est-à-dire si l'**indice effectif** est égal à la **longueur effective**).

SETTEXTRECORD met en mémoire une référence pour le locus donné par le *locus chaîne de caractères* dans la **référence d'enregistrement de texte** du *locus texte*.

SETTEXTINDEX a la même sémantique qu'une *clause d'édition* dans *WRITETEXT* dans laquelle le *code d'édition* est *T* et la *largeur de clause* donne la même valeur que l'*expression entière*, appliquée à l'**enregistrement de texte** désigné par le *locus texte*.

SETTEXTACCESS met en mémoire une référence du locus donnée par le *locus accès* dans la **référence accès** du *locus texte*.

propriétés statiques: la classe de l'appel de routine prédéfinie *GETTEXTRECORD* est la M-classe par référence, où M est le mode **enregistrement de texte** du *locus texte*.

La classe de l'appel de routine prédéfinie *GETTEXTINDEX* est la *&INT*-classe par dérivation.

La classe de routine prédéfinie *GETTEXTACCESS* est la M-classe par référence, où M est le mode **accès** du *locus texte*.

La classe de l'appel de routine prédéfinie *EOLN* est la *BOOL*-classe par dérivation.

Un appel de routine *GETTEXTRECORD* ou *GETTEXTACCESS* a la même **régionalité** que le *locus texte*.

conditions statiques: le mode de l'argument *locus chaîne de caractères* de *SETTEXTRECORD* doit être **compatible en lecture** avec le mode **enregistrement de texte** du *locus texte*.

Le mode de l'argument *locus accès* de *SETTEXTACCESS* doit être **compatible en lecture** avec le mode **accès** du *locus texte*.

L'argument *locus* dans *SETTEXTRECORD* et *SETTEXTACCESS* doit avoir la même **régionalité** que le *locus texte*.

conditions dynamiques: L'exception *TEXTFAIL* se produit si l'argument *expression entière* de *SETTEXTINDEX* donne une valeur inférieure à 0 ou supérieure à la **longueur de texte** du *locus texte*.

exemple:

26.23 GETTEXTINDEX (output) *(1.2)*

8 Filets d'exception

8.1 Généralités

Une exception est soit une exception définie par le langage, auquel cas elle a un nom défini par le langage, une exception définie par l'utilisateur, soit une exception définie par l'implémentation. Une exception définie par le langage sera causée par la violation dynamique d'une condition dynamique. Toute exception nommée peut être causée par l'exécution d'une action routine.

Quand une exception est causée, elle peut être traitée, c'est-à-dire qu'une liste d'énoncés d'action d'un filet qui convient sera exécutée.

Le traitement des exceptions est défini de telle manière que pour tout énoncé, on connaît statiquement les exceptions qui pourraient arriver (c'est-à-dire qu'on sait statiquement quelles exceptions ne peuvent pas arriver) et les exceptions pour lesquelles un filet approprié peut être trouvé, ou les exceptions qui peuvent être passées au point d'appel d'une procédure. Si une exception arrive et qu'aucun filet ne peut être trouvé pour la traiter, le programme est en erreur.

Quand il se produit une exception à un énoncé d'action ou à un énoncé de déclaration, l'exécution de l'énoncé a lieu jusqu'à un point non spécifié, sauf indication contraire dans la section pertinente.

8.2 Filets

syntaxe:

<filet> ::= *(1)*
ON { *<alternative d'exception>* }* [**ELSE** *<liste d'énoncés d'action>*] **END** *(1.1)*

<alternative d'exception> ::= *(2)*
(*<liste d'exceptions>*) : *<liste d'énoncés d'action>* *(2.1)*

sémantique: un filet est entamé s'il convient pour une exception E conformément au 8.3. Si E est mentionné dans une *liste d'exceptions* dans une *alternative d'exception* dans le *filet*, la *liste d'énoncés d'action* correspondante est entamée; sinon, **ELSE** est spécifié et la *liste d'énoncés d'action* correspondante est entamée.

Quand la fin d'une *liste d'énoncés d'action* choisie est atteinte, le *filet* et l'énoncé auquel il est attaché sont terminés.

conditions statiques: tous les *noms d'exception* dans toutes les occurrences de *liste d'exceptions* doivent être différents.

conditions dynamiques: l'exception *SPACEFAIL* arrive si on entame une liste d'énoncés d'action et que les besoins de mémoire ne peuvent pas être satisfaits.

exemple:

10.47 **ON**

(*ALLOCATEFAIL*): **CAUSE** *overflow*;

END *(1.1)*

8.3 Identification de filet

Quand une exception E arrive dans une action ou un module A, ou un énoncé informatif ou une région D, l'exception peut être traitée par le filet qui convient: une liste d'énoncés d'action dans le filet sera exécutée, ou l'exception peut être passée au point d'appel de la procédure ou, si rien n'est possible, le programme est en erreur.

Pour toute action ou module A, ou énoncé informatif ou région D, on peut déterminer statiquement si pour une exception E dans A ou D, un filet qui convient peut être trouvé ou si l'exception peut être passée au point d'appel.

Le filet qui convient pour A ou D par rapport à E est déterminé comme suit:

1) si un filet qui mentionne E dans une *liste d'exceptions* ou qui spécifie **ELSE**, termine A ou D ou est inclus dans l'une ou l'autre de ces lettres, et si E se produit dans le domaine immédiatement englobant le filet, celui-ci est alors le filet qui convient par rapport à E;

2) sinon, si A ou D est immédiatement englobé par une action, un module ou une région parenthésé, le filet qui convient (si présent) est le filet qui convient pour l'action, le module ou la région parenthésé par rapport à E;

3) sinon, si A ou D est placé dans le domaine d'une définition de procédure, alors:

   - si un filet qui mentionne E dans une liste d'exceptions ou qui spécifie **ELSE** termine la définition de procédure, ce filet est alors le filet qui convient,
   - sinon, si E est mentionné dans la liste d'exceptions de la définition de procédure, alors E sera causée au point d'appel,
   - sinon il n'y a pas de filet établi par l'utilisateur; toutefois, dans ce cas, un filet établi par l'implémentation peut convenir (voir 13.5);

4) sinon, si A ou D est placé dans le domaine d'une définition de processus, alors:

   - si un filet qui mentionne E dans une liste d'exceptions ou qui spécifie **ELSE** termine la définition de procédure, ce filet est alors le filet qui convient,
   - sinon, il n'y a pas de filet établi par l'utilisateur; cependant, dans ce cas, un filet défini par l'implémentation peut être utilisé (voir 13.5);

5) sinon, si A est une action ou une liste d'énoncés d'action dans un filet, alors le filet qui convient est celui qui convient pour l'action A' ou l'énoncé informatif ou la région D' par rapport à E que le filet termine ou dans lequel il est inclus mais considéré comme si ce filet n'était pas spécifié.

Si une exception est causée et que le transfert au filet qui convient implique la sortie de blocs, la mémoire locale sera libérée quand les blocs sont quittés.

9 Temporisation

9.1 Généralités

On suppose qu'un concept de temps existe à l'extérieur du programme (système CHILL). Le CHILL ne spécifie pas des propriétés de temporisation précises mais il fournit des mécanismes permettant à un programme d'avoir une interaction avec la vision du temps du monde extérieur.

9.2 Processus temporisables

Le concept de processus **temporisable** existe pour identifier les points précis, pendant l'exécution du programme, où une interruption peut se produire, c'est-à-dire le moment où une temporisation peut perturber l'exécution normale d'un processus.

Un processus devient **temporisable** quand il atteint un point bien défini dans l'exécution de certaines actions. Le CHILL définit qu'un processus devient **temporisable** pendant l'exécution d'actions spécifiques; une implémentation peut définir qu'un processus devient **temporisable** pendant l'exécution d'actions autres.

9.3 Actions de temporisation

syntaxe:

<action de temporisation> ::= *(1)*
<action de temporisation relative> *(1.1)*
| *<action de temporisation absolue>* *(1.2)*
| *<action de temporisation cyclique>* *(1.3)*

sémantique: une action de temporisation spécifie les temporisations du processus en cours d'exécution. Une temporisation peut être déclenchée, expirer et cesser d'exister. Souvent les temporisations peuvent être associées à un même processus en raison de l'action de temporisation cyclique et parce qu'une action de temporisation peut elle-même contenir d'autres actions dont l'exécution peut déclencher des temporisations.

Une interruption se produit quand un processus est **temporisable** et qu'une au moins des temporisations associées a expiré. L'occurrence d'une interruption implique que la première temporisation expirée cesse d'exister; de plus, elle aboutit au transfert de commande associé à cette temporisation dans le processus temporisé. Si le processus est en attente, il devient réactivé.

Les temporisations cessent aussi d'exister quand la commande quitte l'action de temporisation qui les a déclenchées.

NOTE – Si le transfert de commande a pour effet que le processus quitte la région, celle-ci sera libérée (voir 11.2.1).

9.3.1 Action de temporisation relative

syntaxe:

<action de temporisation relative> ::= *(1)*
AFTER *<valeur primitive durée>* [**DELAY**] **IN**
<liste d'énoncés d'action> *<filet de temporisation>* **END** *(1.1)*

<filet de temporisation> ::= *(2)*
TIMEOUT *<liste d'énoncés d'action>* *(2.1)*

sémantique: la *valeur primitive durée* est évaluée, une temporisation est déclenchée, puis la *liste d'énoncés d'action* est entamée.

Si **DELAY** est spécifié, la temporisation est déclenchée quand le processus d'exécution devient **temporisable** au point d'exécution spécifié par l'*énoncé d'action* de la *liste d'énoncés d'action;* sinon, elle est déclenchée avant l'entrée de la *liste d'énoncés d'action*.

Si **DELAY** est spécifié, la temporisation cesse d'exister si elle a été déclenchée et si le processus d'exécution cesse d'être **temporisable**.

La temporisation expire si elle n'a pas cessé d'exister quand la période spécifiée a pris fin depuis le déclenchement.

Le transfert de commande associé à la temporisation consiste à aller à la *liste d'énoncés d'action* du *filet de temporisation*.

conditions statiques: si **DELAY** est spécifié, la *liste d'énoncés d'action* doit se composer d'un *énoncé d'action* précis qui peut à son tour avoir pour effet que le processus d'exécution devient **temporisable**.

conditions dynamiques: l'exception *TIMERFAIL* intervient si le déclenchement de la temporisation échoue pour une raison définie par l'implémentation.

9.3.2 Action de temporisation absolue

syntaxe:

<action de temporisation absolue> ::= *(1)*
AT *<valeur primitive temps absolu>* **IN**
<liste d'énoncés d'action> <filet de temporisation> **END** *(1.1)*

sémantique: la *valeur primitive temps absolu* est évaluée, une temporisation est déclenchée, puis la *liste d'énoncés d'action* est entamée.

La temporisation expire si elle n'a pas cessé d'exister au moment spécifié (ou après).

Le transfert de commande associé à la temporisation consiste à aller à la *liste d'énoncés d'action* du *filet de temporisation*.

conditions dynamiques: l'exception *TIMERFAIL* intervient si le déclenchement de la temporisation échoue pour une raison définie par l'implémentation.

9.3.3 Action de temporisation cyclique

syntaxe:

<action de temporisation cyclique> ::= *(1)*
CYCLE *<valeur primitive durée>* **IN**
<liste d'énoncés d'action> **END** *(1.1)*

sémantique: l'action de temporisation cyclique vise à ce que le processus en cours d'exécution entame la liste d'énoncés d'action à intervalles précis sans décalages cumulés (ceci implique que le temps d'exécution pour la *liste d'énoncés d'action* soit en moyenne inférieur à la durée spécifiée). La *valeur primitive durée* est évaluée, une temporisation relative est déclenchée, puis la *liste d'énoncés d'action* est entamée.

La temporisation expire si elle n'a pas cessé d'exister quand la période spécifiée a pris fin depuis le déclenchement. Indivisiblement de l'expiration, une nouvelle temporisation de même durée est déclenchée.

Le transfert de commande associé à la supervision du temps consiste à aller au début de la *liste d'énoncés d'action*.

On notera que l'action de temporisation cyclique ne peut prendre fin que par un transfert de commande hors de cette action.

propriétés dynamiques: le processus en cours d'exécution devient **temporisable** si et quand la commande atteint la fin de la *liste d'énoncés d'action*.

conditions dynamiques: l'exception *TIMERFAIL* intervient si un déclenchement de temporisation quelconque échoue pour une raison définie par l'implémentation.

9.4 Routines prédéfinies pour le temps

syntaxe:

<appel de routine prédéfinie de valeur temps> ::= *(1)*
<appel de routine prédéfinie de durée> *(1.1)*
| *<appel de routine prédéfinie de temps absolu>* *(1.2)*

sémantique: les conditions requises et les capacités seront sans doute très différentes en matière de précision des valeurs de temps et des intervalles de temps selon les implémentations. Les opérations prédéfinies ci-dessous sont destinées à tenir compte de ces différences d'une manière portable.

9.4.1 Routines prédéfinies de durée

syntaxe:

<appel de routine prédéfinie de durée> ::= *(1)*
MILLISECS (*<expression entière>*) *(1.1)*
| *SECS* (*<expression entière>*) *(1.2)*
| *MINUTES* (*<expression entière>*) *(1.3)*
| *HOURS* (*<expression entière>*) *(1.4)*
| *DAYS* (*<expression entière>*) *(1.5)*

sémantique: un appel de routine prédéfinie de durée livre une valeur avec une précision prédéfinie par l'implémentation et éventuellement variable (c'est-à-dire que *MILLISECS (1000)* et *SECS (1)* peuvent donner des valeurs de durée différentes); cette valeur est l'approximation la plus proche dans la précision choisie pour la période indiquée. L'argument de *MILLISECS*, *SECS*, *MINUTES*, *HOURS* et *DAYS* indique un point dans le temps exprimé en millisecondes, secondes, minutes, heures et jours, respectivement.

propriétés statiques: la classe d'un appel de routine prédéfinie de durée est la *DURATION*-classe par dérivation.

conditions dynamiques: l'exception *RANGEFAIL* intervient si l'implémentation ne peut pas livrer une valeur de durée désignant la période indiquée.

9.4.2 Routine prédéfinie de temps absolu

syntaxe:

<appel de routine prédéfinie de temps absolu> ::= *(1)*
ABSTIME ([[[[[[*<expression année>* ,] *<expression mois>* ,]
<expression jour> ,] *<expression heure>* ,]
<expression minute> ,] *<expression seconde>*]) *(1.1)*

<expression année> ::= *(2)*
<expression entière> *(2.1)*

<expression mois> ::= *(3)*
<expression entière> *(3.1)*

<expression jour> ::= *(4)*
<expression entière> *(4.1)*

<expression heure> ::= *(5)*
<expression entière> *(5.1)*

<expression minute> ::= *(6)*
<expression entière> *(6.1)*

<expression seconde> ::= *(7)*
<expression entière> *(7.1)*

sémantique: l'appel de routine prédéfinie *ABSTIME* livre une valeur de temps absolu désignant le moment dans le calendrier grégorien indiqué dans la liste de paramètres. Les paramètres indiquent les composantes du temps dans l'ordre suivant: l'année, le mois, le jour, l'heure, la minute et la seconde. Quand des paramètres d'ordre supérieur sont omis, le moment indiqué est le prochain moment qui s'accorde avec les paramètres d'ordre inférieur présents (par exemple, *ABSTIME (15,12,00,00)* désigne midi le 15 du présent mois ou du mois prochain).

Quand aucun paramètre n'est spécifié, une valeur de temps absolu dénotant le moment présent est livrée.

propriétés statiques: la classe de l'appel de routine prédéfinie de temps absolu est la *TIME*-classe par dérivation.

conditions dynamiques: l'exception *RANGEFAIL* est causée si l'implémentation ne peut pas livrer une valeur de temps absolu désignant le moment indiqué.

9.4.3 Appel de routine prédéfinie de temporisation

syntaxe:

<appel de routine prédéfinie simple de temporisation> ::= *(1)*
WAIT ( ) *(1.1)*
| *EXPIRED* ( ) *(1.2)*
| *INTTIME* (*<valeur primitive temps absolu>* , [[[[*<locus année>*
<locus mois> ,] *<locus jour>* ,]
<locus heure> ,] *<locus minute>* ,]
<locus seconde>) *(1.3)*

<locus année> ::= *(2)*
<locus entier> *(2.1)*

<locus mois> ::= *(3)*
locus entier> *(3.1)*

<locus jour> ::= *(4)*
<locus entier> *(4.1)*

<locus heure> ::= *(5)*
<locus entier> *(5.1)*

<locus minute> ::= *(6)*
<locus entier> *(6.1)*

<locus seconde> ::= *(7)*
<locus entier> *(7.1)*

sémantique: *WAIT* rend le processus en cours d'exécution inconditionnellement **temporisable**: son exécution peut seulement prendre fin par une interruption (on notera que le processus reste actif au sens du langage CHILL).

EXPIRED rend le processus en cours d'exécution **temporisable** si l'une de ses temporisations associées a expiré; sinon, il n'a aucun effet.

INTTIME affecte aux locus entiers spécifiés une représentation entière du moment spécifié par la *valeur primitive temps absolu* dans le calendrier grégorien. Les locus transférés comme des arguments reçoivent les composantes de temps dans l'ordre suivant: l'année, le mois, le jour, l'heure, la minute et la seconde.

conditions statiques: tous les locus entiers spécifiés doivent être **référençables** et leurs modes ne peuvent pas avoir **la propriété de protection**.

propriétés dynamiques: *WAIT* rend le processus qui l'exécute **temporisable**.

EXPIRED rend le processus qui l'exécute **temporisable** si une temporisation expirée est associée à ce processus.

10 Structure de programme

10.1 Généralités

Les *actions conditionnelles*, *de cas*, *faire*, *attente*, *bloc début-fin*, *module*, *région*, *module de spec*, *région de spec*, *contexte*, *recevoir avec cas*, *définition de procédure* et *définition de processus* déterminent la structure du programme, c'est-à-dire qu'elles déterminent la portée des noms et la durée de vie des locus qui y sont créés.

- Le mot bloc sera employé pour dénoter:
  - la *liste d'énoncés d'action* dans une *action faire*, y compris le *compteur de boucle* et la *commande tandis*;
  - la *liste d'énoncés d'action* dans une *clause alors* dans une *action conditionnelle*;
  - la *liste d'énoncés d'action* dans un *cas alternatif* dans une *action à cas*;
  - la *liste d'énoncés d'action* dans une *alternative* dans une *action attente*;
  - le *bloc début-fin*;
  - la *définition de procédure*, en excluant la *spec de résultat* et la *spec de paramètre* de tous les *paramètres formels* de la *liste de paramètres formels*;
  - la *définition de processus*, en excluant la *spec de paramètre* de tous les *paramètres formels* de la *liste de paramètres formels*;
  - la *liste d'énoncés d'action* dans une *alternative réception de tampon* ou dans un *signal reçu possible*, y compris une *occurrence de définition* ou *liste d'occurrences de définitions* après **IN**;
  - la *liste d'énoncés d'action* après **ELSE** dans une *action conditionnelle*, *à cas*, *recevoir avec cas* ou un *filet*;
  - l'*alternative d'exception* dans un *filet*;
  - la *liste d'énoncés d'action* dans une *action de temporisation relative*, une *action de temporisation absolue*, une *action de temporisation cyclique* ou dans un *filet de temporisation*.
- Le mot modulion sera employé pour dénoter:
  - un *module* ou une *région*, en excluant les *listes de contextes* et les *occurrences de définitions*, s'il en existe;
  - un *module de spec* ou une *région de spec*, en excluant les *listes de contextes*, s'il en existe;
  - un *contexte*;
  - la spécification accompagnée du corps correspondant d'un *mode moreta*.
  - un *gabarit* accompagné du corps correspondant.
- Le mot groupe sera employé pour dénoter soit un bloc soit un modulion.
- Le mot domaine ou domaine d'un groupe sera employé pour dénoter la partie du groupe qui n'est pas entourée (voir 10.2) d'un groupe interne. Si BM est un mode moreta et DM un successeur direct de BM, $BM_P - BM_{CD} \cup DM_P$ forme un domaine. Pour des raisons de visibilité des composantes internes des modes moreta, le domaine d'un successeur est imbriqué directement dans la partie spécification de son prédécesseur direct; cette imbrication se produit à la fin de la partie spécification.

Un groupe influence la portée de chacun des noms créés dans son domaine. Des noms peuvent être créés par des *occurrences de définitions*:

- Une *occurrence de définition* qui apparaît dans la *liste d'occurrences de définitions* d'une *déclaration*, d'une *définition de mode*, ou d'une *définition de synonyme*, ou qui apparaît dans une *définition de signal* crée un *nom* dans le domaine dans lequel, respectivement, la *déclaration*, la *définition de mode*, la *définition de synonyme* ou la *définition du signal* apparaît.
- Une *occurrence de définition* qui apparaît dans un *mode ensemble* crée un nom dans le domaine qui englobe immédiatement le *mode ensemble*.
- Une *occurrence de définition* qui apparaît dans la *liste d'ocurrences de définitions* dans une *liste de paramètres formels* crée un nom dans le domaine de la *définition de procédure* ou de la *définition de processus* correspondante.
- Une *occurrence de définition*, devant un deux points, suivie par une *action*, par une *région*, par une *définition de procédure* ou par une *définition de processus* crée un nom dans le domaine dans lequel, respectivement, l'*action*, la *région*, la *définition de procédure* et la *définition de processus* apparaît.
- Une *occurrence de définition* (virtuelle) introduite par une *partie-avec* ou dans un *compteur de boucle* crée un nom dans le domaine du bloc de l'*action faire* correspondante.
- Une *occurrence de définition* de la *liste d'occurrences de définitions* d'une *alternative réception de tampon* ou d'un *signal* crée un nom, respectivement, dans le domaine du bloc de l'*alternative réception de tampon à choisir* ou du *signal reçu possible* correspondants.
- Une *occurrence de définition* (virtuelle) relative à un nom prédéfini par le langage ou à un nom défini par l'implémentation crée un nom dans le domaine du processus imaginaire le plus externe (voir 10.8).

Les endroits où le nom est employé sont appelés occurrences d'utilisation du nom. Les règles d'identification associent une *définition unique* à chaque occurrence d'utilisation du nom (voir 12.2.2).

Un nom a une certaine portée, c'est-à-dire la partie du programme où sa définition ou déclaration peut être vue et, en conséquence, où il peut être utilisé librement. Le nom est dit être **visible** dans cette partie. Les locus et les procédures ont une certaine durée de vie, c'est-à-dire la partie de programme où ils existent. Les blocs déterminent à la fois la visibilité des noms et la durée de vie des locus qui y sont créés. Les modulions ne déterminent que la visibilité; la durée de vie des locus créés dans le domaine d'un modulion sera la même que s'ils avaient été créés dans le domaine du bloc englobant du plus près. Les modulions permettent de restreindre la visibilité des noms. Par exemple, un nom créé dans le domaine d'un modulion ne sera pas automatiquement **visible** dans les modules englobés ou englobants, bien que la durée de vie le permette.

10.2 Domaines et imbrication

syntaxe:

<corps début-fin> ::= *(1)*
<liste d'énoncés informatifs> <liste d'énoncés d'action> *(1.1)*

<corps de procédure> ::= *(2)*
<liste d'énoncés informatifs><liste d'énoncés d'action> *(2.1)*

<corps de processus> ::= *(3)*
<liste d'énoncés informatifs> <liste d'énoncés d'action> *(3.1)*

<corps de module> ::= *(4)*
{ *<énoncé informatif>* | *<énoncé de visibilité>* | *<région>* |
<région de spec> }* *<liste d'énoncés d'action>* *(4.1)*

<corps de région> ::= *(5)*
{ *<énoncé informatif>* | *<énoncé de visibilité>* }* *(5.1)*

<corps de module de spec> ::= *(6)*
{ *<quasi-énoncé informatif>* | *<énoncé de visibilité>* |
<module de spec> | *<région de spec>* }* *(6.1)*

<corps de région de spec> ::= *(7)*
{*<quasi-énoncé informatif>* | *<énoncé de visibilité>* }* *(7.1)*

<corps de contexte> ::= *(8)*
{ *<quasi-énoncé informatif>* | *<énoncé de visibilité>* |
<module de spec> | *<région de spec>* }* *(8.1)*

<liste d'énoncés d'action> ::= *(9)*
{ *<énoncé d'action>* }* *(9.1)*

<liste d'énoncés informatifs> ::= *(10)*
{ *<énoncé informatif>* }* *(10.1)*

<énoncé informatif> ::= *(11)*
<énoncé déclaratif> *(11.1)*
| *<énoncé définissant>* *(11.2)*

<énoncé définissant> ::= *(12)*
<énoncé de définition de synmode> *(12.1)*
| *<énoncé de définition de néomode>* *(12.2)*
| *<énoncé de définition de synonyme>* *(12.3)*
| *<énoncé de définition de procédure>* *(12.4)*
| *<énoncé de définition de processus>* *(12.5)*
| *<énoncé de définition de signal>* *(12.6)*
| *<gabarit>* *(12.7)*
| *<vide>*; *(12.8)*

sémantique: quand on entame le domaine d'un bloc, toutes les initialisations viagères des locus créés en entamant le bloc sont faites. Après, les initialisations domaniales dans le domaine du bloc, éventuellement les évaluations dynamiques des déclarations d'identité de locus, l'initialisation domaniale dans les régions et les actions sont faites dans l'ordre dans lequel elles sont spécifiées textuellement.

Quand on entame le domaine d'un modulion, les initialisations domaniales, éventuellement les évaluations dynamiques des déclarations d'identité de locus, l'initialisation domaniale dans les régions et les actions (si le modulion est un module) dans le domaine du modulion sont faites dans l'ordre dans lequel elles sont spécifiées textuellement.

Un énoncé informatif, un énoncé relatif à une action, un module ou une région est terminé soit en terminant l'exécution de l'énoncé en question, soit en terminant un filet qui lui est ajouté.

Lorsqu'il faut terminer un groupe G, tous les locus TASK et REGION (RTL) qui *dépendent* de G (voir 12.2.6) sont préalablement *fermés*. Le groupe G est réellement terminé lorsque tous ces locus RTL sont *arrivés à conclusion* (voir 11.6).

Lorsqu'une initialisation domaniale, un énoncé d'identité de locus, une action, un module, une région, une procédure ou un processus d'initialisation domaniale est terminé, l'exécution reprend de la manière suivante selon l'énoncé ou le type d'arrêt:

- si l'énoncé est terminé par la fin de l'exécution d'un filet, l'exécution reprend avec l'énoncé suivant;
- sinon, s'il s'agit d'une action qui implique un transfert de commande, l'exécution reprend avec l'énoncé défini pour cette action (voir 6.5, 6.6, 6.8 et 6.9);
- sinon, s'il s'agit d'une procédure, la commande est renvoyée au point d'appel (voir 10.4);
- sinon, s'il s'agit d'un processus, l'exécution de ce processus (ou du programme, s'il s'agit du tout dernier processus) se termine (voir 11.1) et l'exécution reprend (éventuellement) avec un autre processus;
- sinon, c'est l'énoncé suivant qui reprend le contrôle.

propriétés statiques: tout domaine est immédiatement englobé comme suit par zéro, un ou plusieurs groupes:

- si le domaine est le domaine d'une *action faire*, d'un *bloc début-fin*, d'une *définition de procédure*, d'une *définition de processus*, il est alors immédiatement englobé respectivement par le groupe dans le domaine duquel se trouve placé l'*action faire*, le *bloc début-fin*, la *définition de procédure* ou la *définition de processus*, et seulement par ce groupe;
- si le domaine est la *liste d'énoncés d'action* d'une *action de temporisation* ou d'un *filet de temporisation*, ou une des *listes d'énoncés d'action* d'une *action conditionnelle*, d'une *action à cas* ou d'une *action attente*, il est alors immédiatement englobé par le groupe dans le domaine duquel se trouve placé l'*action de temporisation*, le *filet de temporisation*, l'*action conditionnelle*, l'*action à cas* ou l'*action attente*, et seulement par ce groupe;
- si le domaine est la *liste d'énoncés d'action* ou une *alternative réception de tampon* ou un *signal reçu possible*, ou la *liste d'énoncés d'action* suivant **ELSE** dans une *action recevoir tampon avec ces* ou une *action recevoir signal avec cas*, il est alors immédiatement englobé par le groupe dans le domaine duquel se trouve placé l'*action recevoir tampon avec cas* ou l'*action recevoir signal avec cas*, et seulement par ce groupe;
- si le domaine est la *liste d'énoncés d'action* d'une liste d'alternatives *d'exceptions* ou la *liste d'énoncés d'action* suivant **ELSE** dans un *filet* qui ne termine pas un groupe, il est alors immédiatement englobé par le groupe dans le domaine duquel se trouve placé l'énoncé terminé par le *filet*, et seulement par ce groupe;
- si le domaine est une *alternative d'exception* ou une *liste d'énoncés d'action* suivant **ELSE** dans un *filet* qui termine un groupe, il est alors immédiatement englobé par le groupe terminé par le *filet*, et seulement par ce groupe;
- si le domaine est un *module*, une *région*, un *module de spec* ou une *région de spec*, il est alors immédiatement englobé par le groupe dans le domaine duquel il se trouve placé et aussi englobé dans le *contexte* qui précède immédiatement le *module*, la *région*, le *module de spec* ou la *région de spec*, s'il en existe. C'est le seul cas où un domaine a plus d'un groupe immédiatement englobant;
- si le domaine est un *contexte*, il est alors immédiatement englobé dans le *contexte* qui le précède immédiatement. Si un tel *contexte* n'existe pas, il n'a pas de groupe immédiatement englobant.

Un domaine a des domaines immédiatement englobants qui sont les domaines des groupes immédiatement englobants. Un énoncé a un groupe immédiatement englobant unique, qui est le groupe dans le domaine duquel l'énoncé se trouve placé. Un domaine est dit englober immédiatement un groupe (domaine) si et seulement si le domaine est le domaine immédiatement englobant le groupe (domaine).

Un énoncé (domaine) est dit être englobé par un groupe, si et seulement si, soit le groupe est le groupe immédiatement englobant l'énoncé (domaine), soit le domaine immédiatement englobant est englobé par le groupe.

Un domaine est dit être entamé quand:

- domaine module: le module est exécuté comme une action (c'est-à-dire le module n'est pas dit être entamé quand une action aller transfère la commande à un nom d'**étiquette** défini à l'intérieur du module);
- domaine début-fin: le bloc début-fin est exécuté comme une action;
- domaine région: la région est rencontrée (c'est-à-dire la région n'est pas dite être entamée quand une de ses procédures **critiques** est appelée);
- domaine procédure: la procédure est entamée via son appel de procédure;
- domaine processus: le processus est activé via l'évaluation d'une expression démarrer;
- domaine faire: l'action faire est exécutée comme une action après l'évaluation des expressions ou locus dans la partie de commande;
- domaine alternative réception de tampon, domaine signal reçu possible: le choix est exécutée à la réception d'une valeur tampon ou d'un signal;
- domaine alternative d'exception, l'alternative d'exception est exécuté à cause d'une exception;
- autres domaines de bloc: la liste d'énoncés d'action est entamée.

Une liste d'énoncés d'action est dite être entamée quand et seulement quand sa première action, si présente, reçoit le contrôle depuis l'extérieur de la liste d'énoncés d'action.

Un domaine est un **quasi-**domaine si c'est celui d'un *module de spec*, d'une *région de spec* ou un *contexte*, sinon c'est un domaine **réel**.

Une *occurrence de définition* est une *occurrence de* **quasi**-*définition* si:

- elle est englobée par un *contexte* et non par un module ou une région;
- ou si elle est englobée par un *module de spec simple* ou une *région de spec simple*;
- ou si elle est englobée par l'un des domaines ci-dessus et par une *spec de module* ou une *spec de région* et contenue dans une *quasi-déclaration*, un *énoncé de définition de quasi-procédure* ou un *énoncé de définition de quasi-processus*;

sinon, c'est une *occurrence de définition* **réelle**.

10.3 Blocs début-fin

syntaxe:

<bloc début-fin> ::= *(1)*
BEGIN *<corps début-fin>* **END** *(1.1)*

sémantique: un bloc début-fin est une action contenant éventuellement des déclarations locales et des définitions. Il détermine à la fois la visibilité des noms créés localement et la durée de vie des locus créés localement (voir 10.9 et 12.2).

conditions dynamiques: une exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemples: voir 15.73-15.90

10.4 Définitions de procédure

syntaxe:

<énoncé de définition de procédure> ::= *(1)*
< occurrence de définition> : *<définition de procédure>*
[*<filet>*] [*<chaîne de nom simple>*] ; *(1.1)*
| *<instanciation de procédure générique>* *(1.2)*

<définition de procédure> ::= *(2)*
PROC ([*<liste de paramètres formels>*]) [*<spec de résultat>*]
[**EXCEPTIONS** (*<liste d'exceptions>*)] *<liste d'attributs de procédure>* ;
<corps de procédure> **END** *(2.1)*

<liste de paramètres formels> ::= *(3)*
<paramètre formel> { , *<paramètre formel>* }* *(3.1)*

<paramètre formel> ::= *(4)*
<liste d'occurrences de définitions> <spec de paramètre> *(4.1)*

<liste d'attributs de procédure> ::= *(5)*
[*<généralité>*] *(5.1)*

<généralité> ::= *(6)*
GENERAL *(6.1)*
| **SIMPLE** *(6.2)*
| **INLINE** *(6.3)*

<énoncé de signature de procédure protégée> ::= *(7)*
<occurrence de définition>:
<signature de procédure protégée> [*<chaîne de nom simple>*] ; *(7.1)*

<signature de procédure protégée> ::= *(8)*
PROC ([*<liste de paramètres>*]) [*<spec de résultat>*]
[**EXCEPTIONS** (*<liste d'exceptions>*)]
<liste d'attributs de procédure protégée> **END** *(8.1)*

<énoncé de définition de procédure protégée> ::= *(9)*
<occurrence de définition>: <définition de procédure protégée>
[*<filet>*] [*<chaîne de nom simple>*] ; *(9.1)*

<définition de procédure protégée> ::= *(10)*
PROC ([*<liste de paramètres>*]) [*<spec de résultat>*]
[**EXCEPTIONS** (*<liste d'exceptions>*)] *<liste d'attributs de procédure protégée>*
<corps de procédure> **END** *(10.1)*

<liste d'attributs de procédure protégée> ::= *(11)*
[**GENERAL**] *(11.1)*
| [**SIMPLE**] [*<liste d'attributs de procédure à composante simple>*]
<partie assertion> *(11.2)*
| [**INLINE**] [*<liste d'attributs de procédure à composante en ligne>*] *(11.3)*

<liste d'attributs de procédure à composante simple> ::= *(12)*
<liste d'attributs de procédure à composante en ligne> *(12.1)*
| **DESTR** *(12.2)*
| **INCOMPLETE** *(12.3)*
| [**REIMPLEMENT**] [**FINAL**] *(12.4)*

<liste d'attributs de procédure à composante en ligne> ::= *(13)*
CONSTR *(13.1)*
| **FINAL** *(13.2)*

<partie assertion> ::= *(14)*
[**PRE** (*expression booléenne*)>]
[**POST** (*<expression booléenne>*] *(14.1)*

syntaxe dérivée: un *paramètre formel* où la *liste d'occurrences de définitions* comporte plus d'une *occurence de définition* est dérivé de plusieurs occurrences de *paramètre formel* séparées par des virgules, une pour chaque *occurence de définition* et chacune avec la même *spec de paramètre*. Par exemple: *i, j INT* **LOC** est dérivé de: *i INT* **LOC**, *j INT* **LOC**.

sémantique: un énoncé de définition de procédure définit une séquence d'actions (éventuellement) paramétrée qui peut être appelée de différents endroits du programme. La procédure est terminée et le contrôle revient au point d'appel soit en exécutant une action revenir, soit en atteignant la fin du *corps de procédure*, soit en mettant fin au filet qui termine la définition de procédure (passant les bornes). Différents degrés de complexité de procédure peuvent se spécifier comme suit:

a) les procédures **simples** (**SIMPLE**) sont les procédures qui ne peuvent être manipulées dynamiquement. Elles ne peuvent pas être traitées comme des valeurs, c'est-à-dire qu'elles ne peuvent pas être mises dans des locus procédure, ni être passées comme paramètres à un résultat d'un appel de procédure ou être retournées comme résultat d'un appel de procédure;

b) les procédures **générales** (**GENERAL**) n'ont pas les restrictions des procédures **simples** et peuvent être traitées comme des valeurs procédure;

c) les procédures **en ligne** (**INLINE**) ont les mêmes restrictions que les procédures **simples** et elle ne peuvent être **récursives**. Elles ont la même sémantique que les procédures normales, mais le compilateur insérera le code engendré à l'endroit de l'invocation au lieu d'engendrer le code pour appeler effectivement la procédure.

Seules les procédures **simples** et **générales** sont **récursives**.

Un énoncé de définition de procédure protégée définit une (éventuelle) séquence paramétrée d'actions qui peut être appelée depuis des endroits différents du programme. La procédure est achevée et la commande est renvoyée au point appelant par une *action retour*, par la fin de *corps de procédure* ou par la conclusion d'un *filet* joint à la définition de procédure (passant les bornes).

Lorsque la procédure est définie dans un *mode moreta*, elle est appelée **procédure de composante**. Différentes procédures de composante **simple** et **en ligne** définies en mode moreta peuvent être spécifiées de la manière suivante:

a) une procédure de composante **constr** (**CONSTR**) est un constructeur qui peut être utilisé pour initialiser automatiquement des locus moreta lorsqu'ils sont créés statiquement ou dynamiquement;

b) une procédure de composante **destr** (**DESTR**) est un destructeur qui peut être utilisé pour finaliser des locus moreta lorsqu'ils sont détruits statiquement ou dynamiquement;

c) une procédure de composante **incomplète** (**INCOMPLETE**) a uniquement une signature, pas de corps;

d) une procédure de composante **reimplémentation** (**REIMPLEMENT**) est une procédure à laquelle est accordé un nouveau corps et éventuellement de nouvelles insertions;

e) une procédure de composante **finale** (**FINAL**) est une procédure qui ne peut être réimplémentée dans un mode moreta dérivé.

Différents types de *partie assertion* peuvent être spécifiés pour des procédures de composante **simple**:

a) une partie **pré**assertion (**PRE**) qui est automatiquement vérifiée avant l'exécution du corps de la procédure correspondante;

b) une partie **post**assertion (**POST**) qui est vérifiée automatiquement après l'exécution du corps de la procédure correspondante et avant le retour au point appelant.

Seules les procédures **simples** (sauf les *procédures de composante* ayant l'attribut **constr** ou **destr**, ou ayant une visibilité **publique** dans un mode **région**) et **générales** sont **récursives**.

Une procédure peut retourner une valeur ou elle peut retourner un locus (indiqué par l'attribut **LOC** dans la spec de résultat).

L'*occurence de définition* devant la définition de procédure définit le nom de la procédure.

passage de paramètres

Il y a fondamentalement deux mécanismes de passage de paramètres: le "passage par valeur" (**IN**, **OUT** et **INOUT**) et le "passage par locus" (**LOC**).

passage par valeur

Dans le passage de paramètres par valeur, une valeur est passée comme paramètre à la procédure et mise dans un locus local du mode du paramètre spécifié. Tout se passe comme si, au début de l'appel de procédure, la déclaration de locus:

DCL <*occurence de définition*> <*mode*> *:=* <*paramètre effectif*>;

était rencontrée pour les *occurences de définitions* du *paramètre formel*. Cependant, la procédure est entamée après évaluation des paramètres effectifs. Optionnellement, le nom réservé **IN** peut être spécifié pour indiquer le passage par valeur explicitement.

Si l'attribut **INOUT** est spécifié, la valeur du paramètre effectif est obtenue d'un locus, et juste avant le retour, la valeur courante du paramètre formel est replacée dans le locus effectif.

Les effets de **OUT** sont les mêmes que pour **INOUT**, si ce n'est que la valeur initiale du locus effectif n'est pas copiée dans le locus paramètre formel à l'entrée de la procédure; ainsi, le paramètre formel a une valeur initiale **non définie**. L'opération de recopie ne doit pas se faire si la procédure cause une exception au point d'appel.

passage par locus

Dans le passage de paramètres par locus, un locus (de mode éventuellement dynamique) est passé comme paramètre au corps de procédure. Seuls des locus **référençables** peuvent être passés de cette manière. Tout se passe comme si, au point d'entrée de la procédure, la déclaration identité de locus:

DCL <*occurence de définition*> <*mode*>

LOC [**DYNAMIC**] *:=* <*paramètre effectif*> ;

était rencontrée pour les *occurences de définitions* du *paramètre formel*. Cependant, la procédure est entamée après évaluation des paramètres effectifs.

Si une *valeur* est spécifiée, qui n'est pas un *locus*, un locus contenant la valeur spécifiée sera un locus créé implicite et passé à l'endroit de l'appel. La durée de vie du locus créé est celle de l'appel de procédure. Le mode du locus créé est dynamique si la valeur a une classe dynamique.

transmission de résultat

Une valeur ou un locus peut être retourné par la procédure. Dans le premier cas, une *valeur* est spécifiée dans toute *action résulter*; dans le dernier cas, un *locus* (voir 6.8). Si l'attribut **NONREF** n'est pas donné dans la *spec de résultat*, le *locus* doit être **référençable**. La valeur ou le locus retournés sont déterminés par l'action de résultat la plus récemment exécutée avant de revenir. Si une procédure avec une spec de résultat revient sans avoir exécuté d'action résulter, la procédure retourne une valeur **indéfinie** ou un locus **indéfini**. Dans ce cas, l'appel de procédure ne peut être employé comme appel de procédure rendant locus (voir 4.2.11), ni comme appel de procédure rendant valeur (voir 5.2.13), mais seulement comme action appeler (6.7).

propriétés statiques: une *occurence de définition* dans un *énoncé de définition de procédure* définit un nom de **procédure**.

Un nom de **procédure** possède une *définition de procédure* qui est la *définition de procédure* dans l'énoncé dans lequel le nom de **procédure** est défini.

Un nom de **procédure** possède les propriétés suivantes, définies par sa *définition de procédure*:

- il a une liste de **specs de paramètre**, qui sont définies par les occurrences de *spec de paramètre* dans la *liste de paramètres formels*, chaque paramètre consistant en un mode et éventuellement un attribut de paramètre;
- il a éventuellement une **spec de résultat**, consistant en un mode et un attribut facultatif résulter;
- il a une liste éventuellement vide de noms d'exception qui sont les noms mentionnés dans la *liste d'exceptions*;
- il a une **généralité** qui est, si *généralité* est spécifiée, soit **général**, soit **simple**, soit **en ligne**, selon que **GENERAL**, **SIMPLE** ou **INLINE** est spécifié; sinon, un défaut défini par l'implémentation spécifie **général** ou **simple**. Si le nom de **procédure** est défini à l'intérieur d'un bloc ou d'une région, sa **généralité** est **simple**. Si une procédure est définie dans un mode moreta et que sa visibilité est **publique**, sa généralité est **simple** ou **en ligne**;
- il a une **récursivité** qui est **récursive**. Cependant, si la **généralité** est **en ligne**, ou si le nom de **procédure** est **critique** (voir 11.2.1) la **récursivité** est **non récursive**;
- une procédure de composante a la généralité **en ligne** si l'attribut **INLINE** est spécifié; sinon elle a, par défaut, la généralité **SIMPLE**.

Un nom de **procédure** qui est **général**, est un nom de **procédure général**. Un nom de **procédure général** a un mode procédure qui est construit comme:

PROC ([<*liste de paramètres*>]) [<*spec de résultat*>]

[**EXCEPTIONS** (<*liste d'exceptions*>)]

où <*spec de résultat*>, si présent, et <*liste d'exceptions*> sont les mêmes que dans sa *définition de procédure* et <*liste de paramètres*> est la séquence d'occurrences de <*spec de paramètre*> dans la *liste de paramètres formels*, séparées par des virgules.

Un nom défini dans une *liste d'occurences de définitions* dans le *paramètre formel* est un nom de **locus** si et seulement si la *spec de paramètre* dans le *paramètre formel* ne contient pas l'attribut **LOC**. S'il le contient, c'est un nom d'**identité de locus**. De tels noms de **locus** ou noms d'**identité de locus** sont **référençables**.

Une procédure de composante de mode moreta d'un mode moreta M a une postcondition CPM complète qui est définie de la manière suivante:

a) si M n'a pas de mode de base direct, CPM = partie **post**

b) si M a le mode de base direct B, CPM = partie **post** CPB ∧ , où CPB est la postcondition complète de B.

conditions statiques: si un nom de procédure est **intrarégional** (voir 11.2.2) ou est une procédure publique d'un mode moreta, sa définition de procédure ne doit pas spécifier **GENERAL.**

Si un nom de **procédure** est une procédure **critique** (voir 11.2.1), sa définition ne peut spécifier **GENERAL**.

Si une procédure de composante **simple** a une *partie assertion*, le nom de la procédure doit avoir une visibilité **publique**.

Si une procédure de composante protégée **simple** ou **en ligne** a l'attribut **FINAL**, le nom de la procédure ne doit pas avoir de visibilité **privée**.

L'occurence de définition d'une procédure de composante **constr** doit être la même que celle du mode **moreta** joint. Une procédure de composante **constr** ne doit pas spécifier de *spec de résultat* et doit être **non récursive**.

L'occurence de définition d'une procédure de composante **destr** doit être la même que celle du mode **moreta** joint. Une procédure de composante **destr** ne doit spécifier ni de *liste de paramètres formels*, ni de *spec de résultat* et doit être **non récursive**.

Si elle est spécifiée, la *chaîne de nom simple* doit être la même que la chaîne de *l'occurence de définition* devant la *définition de procédure*.

Ce n'est que si **LOC** est spécifié dans *spec de paramètre* ou *spec de résultat* que le mode qu'il contient a la *propriété de* **non-valeur**.

Tous les noms d'exception mentionnés dans la *liste d'exceptions* doivent être différents.

Si P1 et P2 sont des procédures de composante ou des processus de composante, P1 correspond à P2 si et seulement si:

a) P1 et P2 sont de même nature;

b) P1 et P2 ont la même chaîne de nom simple;

c) les listes de paramètres formels de P1 et P2 sont syntactiquement et sémantiquement équivalentes;

d) les specs de résultat de P1 et P2 sont syntactiquement et sémantiquement équivalentes.

Si P est une procédure de composante pour un processus de composante, P_B *correspond* à P_S si et seulement si:

a) P_B correspond à P_S;

b) les listes d'exceptions de P_S et P_B sont syntactiquement et sémantiquement équivalentes;

c) les listes d'attributs de P_S et P_B sont syntactiquement et sémantiquement équivalentes.

Deux procédures P1 et P2 sont *conformes* l'une à l'autre si et seulement si:

a) les deux ont le même nombre de paramètres et que les noms des modes des paramètres correspondants sont conformes les uns aux autres;

b) (les deux ont un mode résultat et que les noms de ces modes résultats sont conformes les uns aux autres, ou tous les deux n'ont pas de mode résultat).

exemple:

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 Définitions de processus et de spécifications

syntaxe:

<énoncé de définition de processus> ::= *(1)*
<occurence de définition> : *<définition de processus>*
[*<filet>*] [*<chaîne de nom simple>*] ; *(1.1)*
| *<instanciation de processus générique>* ; *(1.2)*

<définition de processus> ::= *(2)*
PROCESS ([*<liste de paramètres formels>*]) *<corps de processus>* **END** *(2.1)*

sémantique: un énoncé de définition de processus définit une séquence d'actions éventuellement paramétrée qui peut être déclenchée pour exécution simultanée à partir de différents endroits du programme (voir l'article 11).

propriétés statiques: une *occurence de définition* dans un *énoncé de définition de processus* définit un nom de **processus**.

Est attachée à un nom de **processus** la propriété suivante définie par sa *définition de processus*:

- il a une liste de **spec de paramètre** définie par les *occurences de specs de paramètre* dans la *liste de paramètres formels*, chaque paramètre comportant un mode et éventuellement un attribut de paramètre.

conditions statiques: si spécifiée, la *chaîne de nom simple* doit être égale à la chaîne de nom de *l'occurence de définition* devant la *définition de processus*.

Un *énoncé de définition de processus* ne doit pas être englobé par une région, ni par un bloc autre que la définition de processus imaginaire le plus externe (voir 10.8).

Les attributs de paramètres dans la *liste de paramètres formels* ne doivent pas être **INOUT** ou **OUT**.

Seulement si **LOC** est spécifié dans la *spec de paramètre* d'un *paramètre formel* de la *liste de paramètres formels*, le mode contenu peut avoir la **propriété de non-valeur**.

exemple:

```
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 Modules

syntaxe:

<module> ::= *(1)*
[*<liste de contextes>*] [*<occurence de définition>*:]
MODULE [***BODY***] *<corps de module>* ***END***
[*<filet>*] [*<chaîne de nom simple>*]; *(1.1)*
| *<modulion distant>* *(1.2)*
| *<instanciation de module générique >* *(1.3)*

sémantique: un module est un énoncé d'action qui peut éventuellement contenir des déclarations et des définitions locales. Un module est un moyen de restreindre la visibilité des chaînes de nom; il n'influence pas la durée de vie des locus créés localement.

Les règles de visibilité détaillées pour les modules sont données au 12.2.

propriétés statiques: une *occurence de définition* dans un *module* définit un nom de **module** ainsi qu'un nom d'**étiquette**. Ce nom a un *module* qui lui est associé (considéré comme un modulion, c'est-à-dire en excluant la *liste de contextes* et *l'occurence de définition*, s'il en existe).

Un *module* est développé par fragments si, et seulement si une *liste de contextes* est spécifiée.

Un *module* est un **corps de module** si, et seulement si **BODY** est spécifié.

conditions statiques: si spécifiée, la *chaîne de nom simple* doit être égale à la chaîne de nom de *occurence de définition*.

Un *modulion distant* dans un *module* doit référencer un *module*.

exemples:

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 Régions

syntaxe:

<région> ::= *(1)*
[*<liste de contextes>*] [*<occurence de définition>* :]
REGION [**BODY**] *<corps de région>* **END**
[*<filet>*] [*<chaîne de nom simple>*] ; *(1.1)*
| *<modulion distant>* *(1.2)*
| *<instanciation de région générique>* *(1.3)*

sémantique: une région est un moyen de réaliser l'exclusion mutuelle, pour accéder aux objets informatifs créés localement, lors de l'exécution simultanée des processus (voir l'article 11). Elle détermine la visibilité des noms créés localement de la même manière qu'un module.

propriétés statiques: une *occurence de définition* dans une *région* définit un nom de **région**. Elle est rattachée à la région (considérée comme un modulion, c'est-à-dire en excluant la *liste de contextes* et l'*occurence de définition*, s'il en existe).

Si, et seulement si une *liste de contextes* est spécifiée, une *région* est développée par fragments.

Une *région* est un **corps de région** si et seulement si **BODY** est spécifié.

conditions statiques: si elle est spécifiée, la *chaîne de nom simple* doit être égale à la chaîne de nom de *l'occurence de définition*.

Une *région* ne doit pas être englobée par un bloc autre que la définition de processus imaginaire le plus externe.

Un *modulion distant* dans une *région* doit référencer une *région*.

exemples: voir 13.1-13.28.

10.8 Programme

syntaxe:

<programme> ::= *(1)*
{ *<module>* | *<module de spec>* | *<région>* | *<région de spec>*
| *<énoncé déclaratif moreta>*
| *<énoncé de définition de synmode moreta>*
| *<énoncé de définition de néomode moreta>*
| *<gabarit>* }$^{+}$ *(1.1)*

sémantique: les programmes consistent en une liste d'unités de programme (conformément à la règle de syntaxe) englobées par une définition de processus imaginaire le plus externe.

Les définitions de noms prédéfinis par le CHILL (voir III.2) et les routines prédéfinies par l'implémentation et les modes entiers sont considérés, pendant leur durée de vie, comme étant définis dans le domaine de la définition d'un processus imaginaire le plus externe. Pour leur visibilité, voir 12.2.

10.9 Allocation de mémoire et durée de vie

Le temps durant lequel un locus ou une procédure existent dans un programme est appelé sa durée de vie.

Un locus est créé par une déclaration ou par l'exécution d'un appel de routine prédéfinie *GETSTACK* ou *ALLOCATE*.

La durée de vie d'un locus déclaré dans le domaine d'un bloc est le temps pendant lequel le contrôle est dans le bloc ou dans une procédure dont l'appel parvient de ce bloc, sauf s'il est déclaré avec l'attribut **STATIC**. La durée de vie d'un locus déclaré dans le domaine d'un modulion est la même que s'il était déclaré dans le domaine du bloc englobant du plus près le modulion. La durée de vie d'un locus déclaré avec l'attribut **STATIC** est la même que s'il était déclaré dans le domaine de la définition de processus imaginaire le plus externe. Ceci implique que pour une déclaration de locus avec l'attribut **STATIC**, l'allocation de mémoire ne se fait qu'une fois, lorsque le processus imaginaire le plus externe démarre. Si une telle déclaration apparaît dans une définition de procédure ou une définition de processus, un seul locus existera pour toutes les invocations ou activations.

La durée de vie d'un locus créé en exécutant l'appel de routine prédéfinie *GETSTACK* s'achève lorsque le bloc immédiatement englobant se termine.

La durée de vie d'un locus créé par un appel de routine prédéfinie *ALLOCATE* est le temps qui s'écoule à partir de l'appel *ALLOCATE* jusqu'au moment où aucun programme CHILL ne peut plus accéder au locus. Tel est toujours le cas si un appel de routine prédéfinie *TERMINATE* est exécuté sur une valeur référence **affectée** référençant le locus.

La durée de vie d'un accès, créé dans une déclaration d'identité de locus est le bloc englobant du plus près la déclaration d'identité de locus.

La durée de vie d'une procédure est le bloc englobant du plus près la définition de procédure.

propriétés statiques: un *locus* est dit **statique** si et seulement si c'est un *locus de mode statique* d'une des sortes suivantes:

- un *nom de locus* déclaré avec l'attribut **STATIC** ou dont la définition n'est pas englobée par un bloc autre que la définition de processus imaginaire le plus externe;
- un *élément de chaîne* ou un segment *de chaîne* où le *locus chaîne* est **statique**, et soit l'*élément de gauche* et l'*élément de droite*, soit l'*élément de début* et le segment *de chaîne* sont **constants**;
- un *élément de matrice* où le *locus matrice* est **statique** et l'*expression* est **constante**;
- une *bande matricielle* où le *locus matrice* est **statique** et soit l'*élément inférieur* et l'*élément supérieur*, soit le *premier élément* et la *taille de bande* sont **constants**;
- un *champ de structure* où le *locus structure* est **statique**;
- une *conversion de locus* où le *locus* qui s'y trouve est **statique**.

10.10 Constructions pour la programmation par fragments

Les modules et les régions sont les unités (fragments) élémentaires dans lesquels un programme CHILL complet qui est mis au point par fragments peut être subdivisé. Le texte de ces fragments est indiqué par des constructions distantes (voir 10.10.1). Le CHILL définit la syntaxe et la sémantique des programmes complets, dans lesquels toutes les occurrences de fragments distants ont été virtuellement remplacées par le texte référencé.

10.10.1 Fragments distants

syntaxe:

<modulion distant> ::= *(1)*
[*<chaîne* de nom simple> :] **REMOTE** *<indicateur de fragment>* ; *(1.1)*

<spec distante> ::= *(2)*
[*<chaîne de nom simple>* :] **SPEC REMOTE** *<indicateur de fragment>* ; *(2.1)*

<contexte distant> ::= *(3)*
CONTEXT REMOTE *<indicateur de fragment>*
[*<corps de contexte>*] **FOR** *(3.1)*

<module de contexte> ::= *(4)*
CONTEXT MODULE REMOTE *<indicateur de fragment>* ; *(4.1)*

<indicateur de fragment> ::= *(5)*
<littéral de chaîne de caractères> *(5.1)*
| *<nom de référence de texte>* *(5.2)*
| *<vide>* *(5.3)*

<unité de programme distante> ::= *(6)*
[*<chaîne de nom simple>* :] **REMOTE** *<indicateur de fragment>* ; *(6.1)*

syntaxe dérivée: la notation:

CONTEXT MODULE REMOTE *<indicateur de fragment>*

est une syntaxe dérivée pour:

CONTEXT REMOTE *<indicateur de fragment>* **FOR**
MODULE SEIZE ALL; **END**;

NOTE – Cette construction est redondante mais peut être utilisée pour vérifier la cohérence.

sémantique: les *modulions distants*, *specs distantes*, *contextes distants*, *modules de contexte* et *unités de programme distantes* sont des moyens utilisés pour représenter le texte source d'un programme sous la forme d'un ensemble de fichiers (interconnectés).

Un *indicateur de fragment* se rapporte d'une manière définie par l'implémentation et comme indiqué ci-après, à une description de fragment de texte source CHILL:

- si l'*indicateur de fragment* est vide, le texte source est extrait d'une position déterminée par la structure du programme dans lequel il se trouve;
- si l'*indicateur de fragment* contient un *littéral chaîne de caractères*, celui-ci est utilisé pour extraire le texte source;
- si l'*indicateur de fragment* contient un *nom de référence de texte*, celui-ci est interprété d'une manière définie par l'implémentation pour extraire le texte source.

Un programme avec: 1) des *modulions distants*, 2) des *specs distantes*, 3) des *unités de programme distantes* est équivalent au programme formé en remplaçant chaque 1) *modulion distant*, 2) *spec distante*, 3) *unité de programme distante* par le fragment de texte CHILL auquel se réfère l'*indicateur de fragment*.

Un programme ayant des *contextes distants* est équivalent au programme constitué en remplaçant chaque *contexte distant* par le fragment de texte CHILL référencé par son *indicateur de fragment* dans lequel le *corps de contexte* a été virtuellement inséré immédiatement après la dernière *occurrence du corps* de contexte dans *la liste de contextes* auxquels renvoie l'*indicateur de fragment*.

Si le fragment désigné par le modulion distant n'est pas disponible comme texte CHILL, l'*indicateur de fragment* qu'il contient est considéré renvoyer à un fragment équivalent de texte CHILL qui est introduit virtuellement.

Bien que la sémantique d'un fragment distant soit définie en termes de remplacement, le CHILL n'implique aucune substitution textuelle.

conditions statiques: l'indicateur de fragment dans 1) un *modulion distant*, 2) une *spec distante*, 3) un *contexte distant*, 4) un *module de contexte*, 5) une *unité de programme* distante doit se référer à une description de fragment de texte source qui est une production terminale 1) d'un *module* ou d'une *région* qui n'est pas un *modulion distant*, 2) d'un *module de spec* ou d'une *région de spec* qui n'est pas une *spec distante*, 3), 4) d'une *liste de contextes* qui n'est pas un *contexte distant* et 5) d'une *unité de programme* qui n'est pas distante.

Lorsque le texte source mentionné par l'*indicateur de fragment* dans un *modulion distant* commence par une *occurence de définition*, le *modulion distant* doit alors commencer par une *chaîne de nom simple* qui est la *chaîne* du nom de cette *occurence de définition*.

Lorsque le texte source mentionné par l'*indicateur de fragment* dans une *spec distante* commence par une *chaîne de nom simple*, la *spec distante* doit alors commencer par la *chaîne de nom simple*.

Lorsque le texte source mentionné par l'*indicateur de fragment* dans une *unité de programme distante* commence par une *chaîne de nom simple*, la première occurence de définition dans l'*unité de programme distante* doit commencer par la même *chaîne de nom simple*.

exemples:

25.9 stack: **REMOTE** *"exemple 27 ou 28"*; *(1.1)*

25.9 "exemple 27 ou 28" *(5.1)*

10.10.2 Modules de spec, régions de spec et contextes

syntaxe:

<module de spec> ::= *(1)*
<module de spec simple> *(1.1)*
| *<spec de module>* *(1.2)*
| *<spec distante>* *(1.3)*

<module de spec simple> ::= *(2)*
[*<liste de contextes>*] [*<chaîne de nom simple>*:] **SPEC MODULE**
<corps de module de spec> **END** [*<chaîne de nom simple>*] ; *(2.1)*

<spec de module> ::= *(3)*
[*<liste de contextes>*] *<chaîne de nom simple>*: **MODULE SPEC**
<corps de module de spec> **END** [*<chaîne de nom simple>*] ; *(3.1)*

<région de spec> ::= *(4)*
<région de spec simple> *(4.1)*
| *<spec de région>* *(4.2)*
| *<spec distante>* *(4.3)*

<région de spec simple> ::= (5)
[*<liste de contextes>*] [*<chaîne de nom simple>* :] **SPEC REGION**
<corps de région de spec> **END** [*<chaîne de nom simple>*] ; *(5.1)*

<spec de région> ::= *(6)*
[*<liste de contextes>*] *<chaîne de nom simple>* : **REGION SPEC**
<corps de région de spec> **END** [*<chaîne de nom simple>*] ; *(6.1)*

<liste de contextes> ::= *(7)*
<contexte> { *<contexte>* }* *(7.1)*
| *<contexte distant>* *(7.2)*

<contexte> ::= *(8)*
CONTEXT*<corps de contexte>* **FOR** *(8.1)*

sémantique: les *modules de spec simple*, les *régions de spec simple* et les *contextes* sont utilisés pour spécifier les propriétés statiques des noms. Ils sont redondants mais ils peuvent servir à une programmation par fragments.

Les *chaînes de nom simple* dans les *modules de spec* et les *régions de spec* ne sont pas des noms; elles ne sont pas **liées** et n'ont pas de règles de visibilité.

1) un *module de spec*, 2) une *région de spec* dans un domaine **réel** indiquent les propriétés d'un ou de plusieurs 1) *modules*, 2) *régions* qui sont compilés par fragments et qui sont considérés être englobés dans ce domaine. Le texte de ces 1) *modules*, 2) *régions* est indiqué par les occurrences des *modulions distants*. Une *liste de contextes* indique les domaines englobants (on notera qu'un modulion qui est compilé par fragments est toujours précédé d'une *liste de contextes*).

Pour chaque *chaîne de nom OP ! NS* **visible** dans le domaine d'une 1) *spec de module*, 2) *spec de région* et **reliée** ici à une **quasi**-occurence de définition **s** et qui est octroyée dans un domaine **réel** tel *NP ! NS*, un énoncé (virtuel) d'octroi avec la même *chaîne* **ancienne** *OP ! NS* et la **nouvelle** *chaîne NP ! NS* est considérée comme étant introduite dans le domaine du 1) **corps de module**, 2) **corps de région** correspondant.

conditions statiques: dans un *module de spec* ou une *région de spec*, la *chaîne de nom simple* optionnelle suivant **END** ne peut être présente que si *la chaîne de nom simple* optionnelle avant **SPEC** est présente. Quand les deux sont présentes, elles doivent avoir des chaînes de nom simple identiques.

Un *contexte* qui n'a pas de groupe immédiatement englobant ne peut contenir d'énoncés de visibilité.

Un domaine **réel** qui contient 1) un *module de spec*, 2) une *région de spec* doit aussi contenir un *modulion distant* et vice versa.

Si un **domaine réel r** contient 1) un *module* qui est un **corps de module**, 2) une *région* qui est un **corps de région**, il doit contenir aussi 1) une *spec de module*, 2) une *spec de région*, de telle sorte que les *chaînes de nom simple* qui les précèdent doivent avoir des chaînes de nom identiques. La 1) *spec de module*, 2) *spec de région* est dite avoir 1) un **corps de module**, 2) un **corps de région**, **correspondant**.

Une *spec distante* dans 1) un *module de spec*, 2) une *région de spec* doit faire référence à 1) un *module de spec*, 2) une *région de spec*.

Un *module de spec* ou une *région de spec* ne doit pas être entouré d'un bloc autre que la définition du processus imaginaire le plus externe

exemple:

```
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 Quasi-énoncés

syntaxe:

```
<quasi-énoncé informatif> ::=                               (1)
        <quasi-énoncé déclaratif>                           (1.1)
    |   <quasi-énoncé définissant>                          (1.2)

<quasi-énoncé déclaratif> ::=                               (2)
        DCL <quasi-déclaration> { , <quasi-déclaration> }* ;    (2.1)

<quasi-déclaration> ::=                                     (3)
        <quasi-déclaration de locus>                        (3.1)
    |   <quasi-déclaration d'identité de locus>             (3.2)

<quasi-déclaration de locus> ::=                            (4)
        <liste d'occurences de définitions> <mode>          (4.1)

<quasi-déclaration d'identité de locus> ::=                 (5)
        <liste d'occurences de définitions> <mode>
        LOC [ NONREF ] [ DYNAMIC ]                          (5.1)

<quasi-énoncé définissant> ::=                              (6)
        <énoncé de définition de synmode>                   (6.1)
    |   <énoncé de définition de néomode>                   (6.2)
    |   <énoncé de définition de synonyme>                  (6.3)
    |   <quasi-énoncé de définition de synonyme>            (6.4)
    |   <quasi-énoncé de définition de procédure>           (6.5)
    |   <quasi-énoncé de définition de processus>           (6.6)
    |   <quasi-énoncé de définition de signal>              (6.7)
    |   <énoncé de définition de signal>                    (6.8)
    |   <vide> ;                                           (6.9)

<quasi-énoncé de définition de synonyme> ::=                (7)
        SYN <quasi-définition de synonyme> { , <quasi-définition de synonyme> }* ;   (7.1)

<quasi-définition de synonyme> ::=                          (8)
        <liste d'occurences de définitions> { <mode> = [ <valeur constante> ] |
        [ <mode> ] = <expression littérale> }               (8.1)

<quasi-énoncé de définition de procédure> ::=               (9)
        <occurence de définition>: PROC ( [ <quasi-liste de paramètres formels> ] )
        [ <spec de résultat> ] [ EXCEPTIONS ( <liste d'exceptions> ) ]
        <liste d'attributs de procédure> [ END [ <chaîne de nom simple> ]] ;   (9.1)

<quasi-liste de paramètres formels> ::=                     (10)
        <quasi-paramètre formel> { , <quasi-paramètre formel> }*   (10.1)

<quasi-paramètre formel> ::=                                (11)
        <chaîne de nom simple> { , <chaîne de nom simple> }* <spec de paramètre>   (11.1)

<quasi-énoncé de définition de processus> ::=               (12)
        <occurence de définition> : PROCESS ( [ <quasi-liste de paramètres formels> ] )
        [ END [ <chaîne de nom simple> ]] ;                 (12.1)
```

<quasi-énoncé de définition de signal> ::= *(13)*
SIGNAL *<quasi-définition de signal>* { , *<quasi-définition de signal>* }* ; *(13.1)*

<quasi-définition de signal> ::= *(14)*
<occurence de définition> [= (*<mode>* { , *<mode>* }*)] [**TO**] *(14.1)*

sémantique: des quasi-énoncés sont utilisés dans des *modules de spec*, des *régions de spec* et des *contextes* pour spécifier les propriétés statiques des noms. Les *modules de spec*, *régions de spec* et *contextes* peuvent contenir des quasi-énoncés et des énoncés réels. Les quasi-énoncés peuvent être redondants, mais sont utilisés pour la programmation par fragments.

Une implémentation qui ne peut garantir l'égalité des valeurs entre **quasi**-noms de **synonyme constants** et les noms **réels** correspondants peut ne pas permettre l'indication de la *valeur constante*.

On notera que dans le CHILL, il n'existe pas de **quasi**-*occurences de définitions* pour les noms d'**étiquette**.

propriétés statiques: les quasi-énoncés sont des formes restreintes des *énoncés* correspondants et ils ont les mêmes propriétés statiques.

Le nom défini par une *occurence de définition* dans une *quasi-déclaration d'identité de locus* est **référençable** si **NONREF** n'est pas spécifié.

conditions statiques: les quasi-énoncés sont des formes restreintes des énoncés correspondants et ils sont soumis aux mêmes conditions statiques.

Un *quasi-énoncé de définition de synonyme* ou un *quasi-énoncé de définition de signal* peut seulement être immédiatement englobé dans un *module de spec simple*, une *région de spec simple* ou un *contexte*. Un *énoncé de définition de synonyme* ou un *énoncé de définition de signal* dans un *quasi-énoncé de définition* peut seulement être immédiatement englobé dans une *spec de module* ou une *spec de région*.

10.10.4 Correspondance entre quasi-occurences de définitions et occurences de définitions

Deux *occurences de définitions* sont dites **correspondre** si elles ont une catégorie sémantique identique et:

- si elles sont des noms de **synonyme**, elles doivent avoir la même **régionalité** et la même valeur, le mode **racine** de leurs classes doit être **semblable** et elles doivent avoir toutes deux une M-classe par valeur, par dérivation, par référence, **nulle** ou **toute**, et si celle qui est quasi est **littérale**, l'autre doit l'être aussi;
- si elles sont des noms de **néomode** ou de **synmode**, leurs modes doivent être **semblables**;
- si elles sont des noms de **locus** ou des noms d'**identité de locus**, elles doivent avoir la même **régionalité**, elles doivent être, ou ne pas être, toutes deux **référençables**, et leurs modes doivent être **semblables**;
- si elles sont des noms de **procédure**, elles doivent avoir les mêmes **régionalité** et **généralité**, elles doivent être ou ne pas être toutes deux **critiques**, elles doivent satisfaire aux mêmes conditions de ressemblance que les modes de procédure, et les *chaînes de nom simple* correspondantes (par position) dans la *liste de paramètres formels* et la *quasi-liste de paramètres formels* doivent être les mêmes;
- si ce sont des noms de **processus**, les paramètres de leurs définitions de processus doivent satisfaire aux mêmes conditions de correspondance et de ressemblance que les paramètres des noms de **procédure**;
- si ce sont des noms de **signal**, elles doivent toutes deux spécifier ou ne pas spécifier **TO**, leurs listes de modes doivent avoir le même nombre de modes et les modes correspondants doivent être **semblables**;

Si deux modes de structure sont **liés par la nouveauté** dans un domaine R, ils doivent avoir le même ensemble de noms de champs **visibles** dans R.

Les règles suivantes doivent être respectées:

- si une *chaîne de nom* dans un domaine qui n'est pas celui d'un *module de spec*, d'une *région de spec* ou d'un *contexte* est liée à une **quasi**-*occurence de définition*, elle doit être aussi **liée** à une *occurence de définition* qui n'est pas une **quasi**-*occurence de définition* et de plus:
  - soit une *chaîne de nom* **liée** à une **quasi**-*occurence de définition* QD et **liée** aussi à une *occurence de définition* **réelle** RD dans le domaine R; dans ces conditions:
    1) QD et RD doivent **correspondre** comme défini plus haut;
    2) RD et QD doivent être toutes deux englobées dans un groupe englobé de R ou toutes deux ne pas être englobées dans le groupe de R ou bien, si R est le domaine d'un *module* ou d'une *région* qui est un **corps de module** ou **de région**, QD doit être englobée dans le groupe de la *spec de module* ou de *région* **correspondante** et RD doit être englobée dans le groupe de R;

- si une *chaîne de nom* dans un domaine **réel** R est **liée** à une **quasi-***occurence de définition* qui est englobée dans le groupe de R (c'est-à-dire entourée par une spec de modulion), elle doit aussi être **liée** à une *occurence de définition* **réelle** qui est entourée par le groupe d'un *module* ou d'une *région* qui sont indiqués par un *modulion distant* immédiatement englobé dans R (informellement: si l'interface octroie, il doit en être de même pour l'implémentation). Si la **quasi-***occurence de définition* est englobée dans le groupe d'une *spec de module* ou d'une *spec de région*, la définition **réelle** doit être englobée dans le groupe du modulion **correspondant**;
- pour chaque *chaîne de nom* dans le domaine Q d'un *module de spec* ou d'une *région de spec* immédiatement englobée dans un domaine **réel** R qui est **lié** à une *occurence de définition* non entourée par Q, il doit y avoir une *chaîne de nom* identique dans le domaine d'un *module* ou d'une *région* qui est indiqué par un *modulion distant* immédiatement englobé dans R qui est **lié** à la même *occurence de définition* (informellement, si l'interface saisit, il doit en être de même pour l'implémentation);

- si deux *chaînes de nom* sont **liées** à la même 1) *occurence de définition* **réelle**, 2) **quasi-***occurence de définition* dans un domaine, les deux *chaînes de nom* doivent être **liées** à la même 1) **quasi-***occurence de définition*, 2) *occurence de définition* **réelle** ou bien les deux ne doivent plus être **liées**;
- une **nouveauté réelle** peut ne pas être **liée par la nouveauté** à deux **quasi-nouveautés** dans chaque domaine;

  soit une **quasi-nouveauté** QN et une **nouveauté réelle** RN **liées par la nouveauté** l'une à l'autre dans un domaine R; dans ces conditions, RN et QN doivent être toutes deux englobées dans un groupe englobé de R ou toutes deux ne pas être englobées dans le groupe de R, ou si R est le domaine d'un *module* ou d'une *région* qui est un **corps de module** ou **de région**, alors RN doit être englobée dans le groupe de R et QN doit être englobé dans le groupe de la *spec de module* ou de *la spec. de région* **correspondante**.

10.11 Généricité

Beaucoup d'algorithmes résolvent des problèmes portant sur des éléments d'information structurés d'une manière analogue mais dont les modes composantes sont différents. La généricité offre le moyen d'implémenter ces algorithmes comme des procédures à programme qui sont instanciées par substitution de données réelles aux définitions formelles.

syntaxe:

<gabarit> ::= *(1)*
- *<gabarit de module générique>* *(1.1)*
- | *<gabarit de région générique>* *(1.2)*
- | *<gabarit de procédure générique>* *(1.3)*
- | *<gabarit de processus générique>* *(1.4)*
- | *<gabarit de mode module générique>* *(1.5)*
- | *<gabarit de mode région générique>* *(1.6)*
- | *<gabarit de mode tâche générique>* *(1.7)*
- | *<gabarit de mode interface générique>* *(1.8)*
- | *<unité de programme distante>* *(1.9)*

<gabarit de module générique> ::= *(2)*
[*liste de contextes*>] [*<occurence de définition>* :]
<partie générique> **MODULE** [**BODY**] *<corps de module>* **END**
[*<filet>*] [*<chaîne de nom simple>*] ; *(2.1)*

<gabarit de région générique> ::= *(3)*
[*<liste de contextes>*] [*<occurence de définition>* :]
<partie générique> **REGION** [**BODY**] *<corps de région>* **END**
[*<filet>*] [*<chaîne de nom simple>*] ; *(3.1)*

<gabarit de procédure générique> ::= *(4)*
<occurence de définition> : *<partie générique> <définition de procédure>*
[*<filet>*] [*<chaîne de nom simple>*] ; *(4.1)*

<gabarit de processus générique> ::= *(5)*
<occurence de définition>: *<partie générique> <définition de processus>*
[*<filet>*] [*<chaîne de nom simple>*] ; *(5.1)*

<gabarit de mode module générique> ::= *(6)*
<partie générique> <spécification de mode module> *(6.1)*

<gabarit de mode région générique> ::= *(7)*
<partie générique> <spécification de mode région> *(7.1)*

<gabarit de mode tâche générique> ::= *(8)*
<partie générique> *<spécification de mode tâche>* *(8.1)*

<gabarit de mode interface générique> ::= *(9)*
<partie générique> *<mode interface>* *(9.1)*

<partie générique> ::= *(10)*
GENERIC { *<énoncé de saisie>* }* *<liste de paramètres génériques formels>* *(10.1)*

<liste de paramètres génériques formels> ::= *(11)*
{ *<paramètre générique formel>* }* *(11.1)*

<paramètre générique formel> ::= *(12)*
SYN *<liste de synonymes génériques formels>* ; *(12.1)*
| **MODE** *<liste de modes génériques formels>* ; *(12.2)*
| **PROC** *<spec de procédure générique formelle>* ; *(12.3)*

<liste de synonymes génériques formels> ::= *(13)*
<synonyme générique formel> { , *<synonyme générique formel>* }* *(13.1)*

<liste de modes génériques formels> ::= *(14)*
<mode générique formel> { , *<mode générique formel>* }* *(14.1)*

<synonyme générique formel> ::= *(15)*
<liste d'occurences de définitions> =
{*<mode>* | **ANY_DISCRETE** | **ANY_INT** | **ANY_REAL** } *(15.1)*

<mode générique formel> ::= *(16)*
<liste d'occurences de de définitions> = *<indication de mode générique formel>* *(16.1)*

<indication de mode générique formel> ::= *(17)*
ANY *(17.1)*
| **ANY_ASSIGN** *(17.2)*
| **ANY_DISCRETE** *(17.3)*
| **ANY_INT** *(17.4)*
| **ANY_REAL** *(17.5)*
| *<nom de mode moreta>* *(17.6)*

<spec de procédure générique formelle> ::= *(18)*
<chaîne de nom simple> ([*<liste de paramètres formels>*]) [*<spec de résultat>*]
[**EXCEPTIONS** (*<liste d'exceptions>*)] *(18.1)*

<instanciation de module générique> ::= *(19)*
<chaîne de nom simple> : **MODULE** = **NEW** *<nom de module générique>*
{ *<énoncé de saisie>* }*

<liste de paramètres génériques effectifs> **END** [*<chaîne de nom simple>*] ; *(19.1)*

<instanciation de région générique> ::= *(20)*
<chaîne de nom simple> : **REGION** = **NEW** *<nom de région générique>*
{ *<énoncé de saisie>* }*
<liste de paramètres génériques effectifs> **END** [*<chaîne de nom simple>*] ; *(20.1)*

<instanciation de procédure générique> ::= *(21)*
<chaîne de nom simple> : **PROC** = **NEW** *<nom de procédure générique>*
{ *<énoncé de saisie>* }*
<liste de paramètres génériques effectifs> **END** [*<chaîne de nom simple>*] ; *(21.1)*

<instanciation de processus générique> ::= *(22)*
<chaîne de nom simple> : **PROCESS** = **NEW** *<nom de processus générique>*
{ *<énoncé de saisie>* }*
<liste de paramètres génériques effectifs> **END** [*<chaîne de nom simple>*] ; *(22.1)*

<instanciation de mode moreta générique> ::= *(23)*
NEW *<nom de mode moreta générique>*
{ *<énoncé de saisie>* }*
<liste de paramètres génériques effectifs> **END** [*<chaîne de nom simple>*] ; *(23.1)*

<liste de paramètres génériques effectifs> ::= *(24)*
<paramètre générique effectif> { *<paramètre générique effectif>*}* *(24.1)*

<paramètre générique effectif> ::= *(25)*
<énoncé de définition> *(25.1)*
| *<énoncé de définition de synmode>* *(25.2)*
| *<énoncé de définition de néomode>* *(25.3)*
| *<procédure générique effective>* *(25.4)*

<procédure générique effective> ::= *(26)*
PROC *<liste d'occurences de définitions> = <nom de procédure>* ; *(26.1)*

sémantique: par *unité* on entend un module, une région, une procédure, un processus ou un mode moreta.

Une unité générique est une unité qui contient une partie générique.

Une unité générique est un gabarit à partir duquel on peut obtenir des unités non génériques au moyen d'un processus appelé instanciation générique.

Une unité générique peut contenir des paramètres génériques formels. Au cours de l'instanciation générique, l'unité générique est copiée et les paramètres génériques formels sont remplacés par les paramètres génériques réels dans l'ensemble de l'unité. Après ce remplacement, la partie générique est supprimée et seule subsiste l'unité non générique.

propriétés statiques: les synonymes génériques formels se caractérisent par deux propriétés:

a) les propriétés d'un paramètre générique formel à l'intérieur de l'unité générique;

b) les propriétés dont doit disposer un paramètre générique réel correspondant pour être accepté:

mode:	prop. form.:	propriétés du mode donné qui ne doivent pas avoir la propriété de **non-valeur**
	prop. réelle:	la valeur du paramètre générique réel doit être une valeur du mode.
ANY_DISCRETE:	prop. form.:	opérations possibles: : =, relationnel, PRED, SUCC, NUM, SIZE
	prop. réelle:	la valeur du paramètre générique réel doit être une valeur du mode discret.
ANY_INT:	prop. form.:	ANY_DISCRETE et +, –, *, /, mod, abs, rem.
	prop. réelle:	la valeur du paramètre générique réel doit être une valeur d'un mode entier.
ANY_REAL:	prop. form.:	opérations possibles: ANY_ASSIGN et relationnel, +, –, *, /.
	prop. réelle:	la valeur du paramètre générique réel doit être une valeur d'un mode réel.

Les modes génériques formels se caractérisent par deux propriétés:

a) les propriétés d'un paramètre générique formel à l'intérieur de l'unité générique

b) les propriétés dont doit disposer un paramètre générique réel correspondant pour être accepté:

ANY:	prop. form.:	SIZE; ne peut pas être utilisé en tant que mode d'un locus ou d'un paramètre; (peut être utilisé comme un mode référencé).
	prop. réelle:	tout mode acceptable
ANY_ASSIGN:	prop. form.:	opérations possibles: :=, comparaison, SIZE.
	prop. réelle:	le mode doit avoir les propriétés formelles.
ANY_DISCRETE:	prop. form.:	opérations possibles: :=, relationnel, PRED, SUCC, NUM, SIZE.
	prop. réelle:	le mode doit avoir les propriétés formelles.
ANY_INT:	prop. form.:	ANY_DISCRETE et +, –, *, /, mod, abs, rem.
	prop. réelle:	le mode doit avoir les propriétés formelles.
ANY_REAL:	prop. form.:	opérations possibles: ANY_ASSIGN et relationnel, +, –, *, /.
	prop. réelle:	le mode doit avoir les propriétés formelles.
moreta mode name:	prop. form.:	celles du mode.
	prop. réelle:	le même mode ou tout successeur.

Les procédures génériques formelles se caractérisent par deux propriétés:

a) les propriétés d'un paramètre générique formel à l'intérieur de l'unité générique

b) les propriétés dont doit disposer un paramètre générique réel correspondant pour être accepté:

prop. form.: conformément à la spec de procédure générique formelle en question.

prop. réelle: la spec de procédure générique formelle en question doit être **compatible** avec la classe du paramètre générique réel.

conditions statiques: les restrictions suivantes s'appliquent aux calculs faisant intervenir des gabarits de mode moreta générique: si la base est un gabarit, toute entité qui en découle doit aussi être un gabarit, si la base n'est pas un gabarit, toute entité qui en découle peut être un gabarit.

Il faut, dans une instanciation générique, exactement un paramètre générique réel pour chaque paramètre générique formel de l'unité générique en cours d'instanciation.

Les restrictions relatives à l'imbrication de groupes sont données dans le tableau ci-après qui s'applique aux groupes ordinaires, aux groupes génériques et aux instanciation génériques.

groupe interne / groupe externe	MODULE	RÉGION	PROCÉDURE	PROCESSUS	Mode Module	Mode Région	Mode Tâche	Mode Interface
Début-fin	Oui	Non	Oui	Non	Oui	Non	Non	Oui
PROCÉDURE	Oui	Non	Oui	Non	Oui	Non	Non	Oui
PROCESSUS	Oui	Non	Oui	Non	Oui	Non	Non	Oui
MODULE	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
RÉGION	Oui	Non	Oui	Non	Oui	Non	Non	Oui
Mode module	Non	Non	Oui	Oui	Non	Non	Non	Non
Mode région	Non	Non	Oui	Non	Non	Non	Non	Non
Mode tâche	Non	Non	Oui	Non	Non	Non	Non	Non
Mode interface	Non	Non	Non	Non	Non	Non	Non	Non
Programme	Oui	Oui	Oui	Non	Oui	Oui	Oui	Oui

Le tableau est fondé sur la correspondance suivante entre les gabarits et les entités du langage CHILL. Pour un gabarit de la colonne de gauche, les restrictions de l'entité correspondante de la colonne de droite s'appliquent.

gabarit de module générique	énoncé de définition de procédure
gabarit de région générique	région
gabarit de procédure générique	énoncé de définition de procédure
gabarit de processus générique	énoncé de définition de processus
gabarit de mode module générique	énoncé de définition de procédure
gabarit de mode région générique	région
gabarit de mode tâche générique	énoncé de définition de processus
gabarit de mode interface générique	énoncé de définition de procédure

11 Exécution simultanée

11.1 Les processus, les tâches, les fils d'exécution et leurs définitions

Un fil d'exécution est un processus ou une tâche. Un processus est l'exécution séquentielle d'une série d'énoncés. Il peut être exécuté en parallèle avec d'autres fils d'exécution. Le comportement d'un processus est décrit par une définition de processus (voir 10.5), qui décrit les objets locaux au processus et la série d'énoncés d'action à exécuter séquentiellement.

Un processus est créé par l'évaluation d'une expression démarrer (voir 5.2.15). Il devient actif (c'est-à-dire en exécution) et il est considéré être exécuté en parallèle avec d'autres fils d'exécution. Le processus créé est une activation de la définition indiquée par le nom de **processus** de la définition du processus. Un nombre arbitraire de processus qui ont la même définition peuvent être créés et peuvent être exécutés en parallèle. Chaque processus est identifié univoquement

par une valeur, donnée comme résultat de l'expression démarrer, ou l'évaluation de l'opérateur **THIS**. La création d'un processus cause la création de ses locus déclarés localement, sauf ceux qui sont déclarés avec l'attribut **STATIC** (voir 10.9), et de ses valeurs et de ses procédures définies localement. Les locus déclarés localement ainsi que les valeurs et procédures sont dits avoir la même activation que le processus créé auquel ils appartiennent. Le processus imaginaire le plus externe (voir 10.8) qui est tout le programme CHILL en exécution, est considéré comme étant créé par une expression démarrer exécutée par le système sous le contrôle duquel le programme est exécuté. A la création d'un processus, ses paramètres formels, si présents, dénotent les valeurs et locus donnés par les paramètres effectifs correspondants dans l'expression démarrer.

Un processus est terminé par l'exécution d'une action arrêter, en atteignant la fin du corps de processus ou en terminant un filet spécifié à la fin de la définition de processus (passant les bornes). Si le processus imaginaire le plus externe exécute une action arrêter ou passe les bornes, la terminaison ne se fera que quand et seulement quand tous les autres fils d'exécution du programme seront terminés.

Une tâche est l'exécution séquentielle d'une série d'énoncés. Elle peut être exécutée en parallèle avec d'autres fils d'exécution. Le comportement d'une tâche est décrit par une définition de mode tâche.

Une tâche est créée dans le cadre de la création et de l'initialisation d'un locus de mode tâche (voir 4.1), et on dit qu'elle appartient à ce locus. Une tâche se termine lorsque son locus de mode tâche est détruit (voir 10.2). Un fil d'exécution est, au niveau du programme CHILL, toujours dans l'un des deux états: il est soit actif (c'est-à-dire en exécution) ou en attente (voir 11.3). La transition d'actif à en attente est appelée la mise en attente du fil d'exécution, la transition d'en attente à actif est appelée sa réactivation.

11.2 Exclusion mutuelle et régions

11.2.1 Généralités

Les régions (voir 10.7) et les locus de région (voir 3.15) sont un moyen de fournir aux fils d'exécution un accès mutuellement exclusif aux locus déclarés à l'intérieur des régions ou des locus de région par des procédures d'octroi. Les conditions de contexte statiques (voir 11.2.2) sont telles que des accès par un fil d'exécution, autre que le processus imaginaire le plus externe, aux locus déclarés dans une région ou un locus de région ne peuvent se faire qu'en appelant des procédures qui sont définies à l'intérieur de la région et octroyées par la région ou le locus de région.

NOTE – La seule situation dans laquelle les locus déclarés à l'intérieur d'une région ou d'un locus de région peuvent être directement accédés par un fil d'exécution T est celle où la région ou le locus de région est entré et où ses initialisations domaniales (éventuelles) sont exécutées par T. Un nom de **procédure** est dit dénoter une procédure **critique** (et c'est un nom de **procédure critique**) s'il est défini à l'intérieur d'une région et octroyé par la région.

Un nom de **procédure de composante** est dit dénoter une procédure de composante **critique** (et c'est un nom de **procédure de composante critique**) s'il est défini à l'intérieur d'un mode région et octroyé par le mode région. Une région est dite libre si et seulement si le contrôle ne réside dans aucune de ses procédures **critiques** ni dans la région elle-même pour effectuer les initialisations domaniales.

Un locus de région est dit libre si et seulement si le contrôle ne réside dans aucune de ses procédures de composante **critique** ni dans la région elle-même pour effectuer les initialisations domaniales. La région sera verrouillée (pour empêcher l'exécution simultanée) si:

- elle est entamée (on notera que les régions n'étant pas entourées d'un bloc, il est impossible de faire plusieurs essais simultanés pour entamer la région);
- une procédure **critique** de la région est appelée;
- un processus, en attente sur la région, est réactivé.

Le locus de région sera verrouillé (pour empêcher l'exécution simultanée) si:

- il est entamé;
- une procédure de composante critique du locus de région est appelé;
- un fil d'exécution, mis en attente sur le locus de région, est réactivé.

La région sera à nouveau libre si:

- elle est quittée après ses initialisations domaniales;
- une procédure **critique** revient;
- une procédure **critique** exécute une action qui cause l'attente du processus exécutant (voir 11.3). Dans le cas d'appels de procédures **critiques** imbriquées dynamiquement, seule la région verrouillée en dernier sera libérée;
- le processus exécutant la procédure **critique** est terminé. Dans le cas d'appels de procédures **critiques** imbriquées dynamiquement, toutes les régions verrouillées par le processus seront libérées.

Le locus de région sera à nouveau libre si:

- il est quitté après ses initialisations domaniales;
- une composante critique revient;
- une composante critique exécute une action qui cause la mise en attente du fil d'exécution en cours (voir 11.3). Dans le cas d'appels de procédures critiques imbriquées dynamiquement, seule la région verrouillée en dernier sera libérée;
- un fil exécutant une procédure de composante critique se termine. Dans le cas d'appels de procédure de composantes critiques imbriquées dynamiquement, toutes les régions verrouillées par le fil d'exécution seront libérées. Si, pendant que la région est verrouillée, un fil d'exécution tente d'appeler une de ses procédures **critiques** ou qu'un fil d'exécution mis en attente dans la région est réactivé, ce fil d'exécution est mis en suspens jusqu'à ce que la région soit libérée (à noter que le fil d'exécution reste actif au sens du langage CHILL).

Si, pendant que le locus de région est verrouillé, un fil d'exécution tente d'appeler une de ses procédures de composantes **critiques** ou qu'un fil d'exécution mis en attente dans le locus de région est réactivé, ce fil d'exécution est mis en suspens jusqu'à ce que le locus de région soit libéré (à noter que le fil d'exécution reste actif au sens du langage CHILL). Quand une région est libérée et que plus d'un fil d'exécution a été suspendu en essayant d'appeler une de ses procédures **critiques** ou d'être réactivé dans une de ses procédures **critiques**, un seul fil d'exécution sera sélectionné pour verrouiller la région suivant un algorithme de sélection défini par l'implémentation.

Quand un locus de région est libéré et que plus d'un fil d'exécution a été suspendu en essayant d'appeler une de ses procédures de composantes **critiques** ou d'être réactivé dans une de ses procédures de composantes **critiques**, un seul fil d'exécution sera sélectionné pour verrouiller le locus de région suivant un algorithme de sélection défini par l'implémentation.

11.2.2 Régionalité

Pour permettre une vérification statique du fait qu'un locus déclaré dans une région ne peut être accédé qu'en appelant une procédure **critique** ou en entamant la région pour effectuer les initialisations domaniales, les conditions de contexte statiques suivantes doivent être respectées:

- les conditions de **régionalité** mentionnées dans les sections adéquates (action d'affectation, appel de procédure, action envoyer, action résulter, etc.);
- les procédures **intrarégionales** ne sont pas **générales** (voir 10.4);
- les procédures **critiques** ne sont ni **générales**, ni **récursives** (voir 10.4).

Afin de permettre de s'assurer statiquement qu'un locus de composante déclaré dans un locus de région n'est accessible qu'à des procédures de **composantes critiques** ou par l'entrée dans le locus de région pour effectuer des initialisations domaniales, les conditions de contexte statiques suivantes sont appliquées:

- les prescriptions de régionalité mentionnées dans les sections appropriées (action affecter, appel de procédure, action envoyer, action résultat, etc.);
- les procédures de composantes intrarégionales ne sont pas générales (voir 10.4);
- les procédures de composantes critiques sont ni générales, ni récursives (voir 10.4);
- les procédures de composantes critiques ne sont pas, par ailleurs, (voir 3.15).

Un *locus* et un *appel de procédure* ont une **régionalité** qui est **intrarégionale** ou **extrarégionale**. Une *valeur* a une **régionalité** qui est **intrarégionale**, **extrarégionale** ou **nulle**. Ces propriétés sont définies de la manière suivante:

1) **Locus**

Un *locus* est **intrarégional** si et seulement si une des conditions suivantes est remplie:

- c'est un *nom d'accès* qui est:
  - soit un *nom de locus* déclaré textuellement à l'intérieur d'une *région* ou d'une *région de spec* et qui n'est pas défini dans un *paramètre formel* d'une procédure **critique**,
  - soit un *nom de locus* déclaré textuellement à l'intérieur d'un *mode de région* et qui n'est pas défini dans un *paramètre formel* d'une procédure de composante **critique**,
  - soit un *nom d'identité de locus* où le *locus*, dans sa déclaration, est **intrarégional** ou qui est défini dans un *paramètre formel* d'une procédure **intrarégionale**,

  - soit un *nom d'identité de locus* où le *locus*, dans sa déclaration, est **intrarégional** ou qui est défini dans un *paramètre formel* d'une procédure de composante **intrarégionale**,
  - soit un *nom d'énumération de locus*, dont le *locus matrice* ou le *locus chaîne* dans *l'action faire* associée est **intrarégional**,
  - soit un *nom de locus faire-avec*, dont le *locus structure* dans l'action faire associée est **intrarégional**.
- C'est une *référence liée déréférencée*, contenant une *valeur primitive référence liée* qui est **intrarégionale**.
- C'est une *référence libre déréférencée*, contenant une *valeur primitive référence libre* qui est **intrarégionale**.
- C'est un *descripteur déréférencé*, contenant une *valeur primitive descripteur* qui est **intrarégionale**.
- C'est un *élément de matrice* ou une *bande matricielle*, contenant un *locus matrice* qui est **intrarégional**.
- C'est un *élément de chaîne* ou un *segment de chaîne*, contenant un *locus chaîne* qui est **intrarégional**.
- C'est un *champ de structure*, contenant un *locus structure* qui est **intrarégional**.
- C'est un *appel de procédure rendant locus*, tel que dans *l'appel de procédure rendant locus* on spécifie un *nom de procédure* qui est **intrarégional**.
- C'est un *appel de routine prédéfinie rendant locus*, que la définition CHILL ou l'implémentation spécifie comme étant **intrarégional**.
- C'est une *conversion de locus*, contenant un *locus de mode statique* qui est **intrarégional**.

Un *locus* qui n'est pas **intrarégional** est **extrarégional**.

2) **Valeur**

Une *valeur* a une **régionalité** qui dépend de sa classe. Si elle a la M-classe par dérivation, la classe **toute** ou la classe **nulle**, alors elle a une **régionalité nulle**. Sinon elle a la M-classe par valeur ou la M-classe par référence et elle a une **régionalité** qui dépend du mode M comme suit:

si la *valeur* a la M-classe et si M n'a pas la **propriété de référencer**, alors la **régionalité** est **nulle**; sinon, la *valeur* est un *opérande–7* (et a la **propriété de référencer**) ou une *expression conditionnelle*:

si c'est une *valeur primitive*:

- et si c'est un *contenu de locus* qui est un *locus*, c'est celle du *locus*;
- et si c'est un *contenu de locus de composante* qui est un *locus de composante*, c'est celle du *locus* de composante;
- et si c'est un *nom de valeur*, alors:
  - si c'est un *nom de synonyme*, c'est celui de la *valeur constante* dans sa définition;
  - si c'est un *nom de valeur faire-avec*, c'est celui de la *valeur primitive structure* de l'action faire associée;
  - si c'est un *nom de valeur reçue*, elle est **extrarégionale**.
- Si c'est un *multiplet* et si l'une de ses occurrences de *valeur* a une **régionalité** non **nulle**, c'est celle de cette *valeur* (peu importe le choix qui est fait, voir 5.2.5, conditions statiques); sinon, il est **nul**.
- Si c'est une *valeur élément de matrice*, ou une *valeur bande matricielle*, c'est celle de la *valeur primitive matrice* qu'elle contient.
- Si c'est une *valeur champ de structure*, c'est celle de la *valeur primitive structure* qu'elle contient.
- Si c'est une *conversion d'expression*, c'est celle de l'*expression* qu'elle contient.
- Si c'est un *appel de procédure rendant valeur*, c'est celle de l'*appel de procédure* qu'elle contient.
- Si c'est un *appel de procédure de composante*, c'est celle de l'*appel de procédure de composante* qu'elle contient.
- Si c'est un *appel de routine prédéfinie rendant valeur* que la définition CHILL ou l'implémentation spécifie comme étant **intrarégional** ou **extrarégional**.

Si c'est un *locus référencé*, c'est celui du *locus* qu'elle contient.

Si c'est une *expression conditionnelle*, alors si l'une de ses occurrences de *sous-expression* a une **régionalité** non **nulle**, c'est celle de cette *sous-expression* (peu importe le choix qui est fait, voir 5.3.2, conditions statiques); sinon elle est **nulle**.

3) **Nom de procédure**

Un *nom de procédure* est **intrarégional** si et seulement s'il est défini à l'intérieur d'une *région* ou d'une *région de spec* et qu'il n'est pas **critique** (c'est-à-dire qu'il n'est pas octroyé par la région). Sinon, il est **extrarégional**.

Un *nom de procédure de composante* est **intrarégional** si et seulement s'il est défini à l'intérieur d'un *mode région* et qu'il n'est pas **critique** (c'est-à-dire qu'il n'est pas octroyé par la région). Sinon, il est **extrarégional**.

4) **Appel de procédure**

Un *appel de procédure* est **intrarégional** s'il contient un *nom de procédure* qui est **intrarégional**; sinon, il est **extrarégional**.

Un *appel de procédure de composante* est **intrarégional** s'il contient un *nom de procédure de composante* qui est **intrarégional**; sinon, il est **extrarégional**.

Une *valeur* est **régionalement sûre** pour un non-terminal (utilisée seulement pour *locus*, *appel de procédure* et *nom de procédure*) si et seulement si:

- le non-terminal est **extrarégional** et la *valeur* n'est pas **intrarégionale**;
- le non-terminal est **intrarégional** et la *valeur* n'est pas **extrarégionale**;
- le non-terminal a la **régionalité nulle**.

11.3 Mise en attente d'un fil d'exécution

Un fil d'exécution actif peut être mis en attente en exécutant l'une des actions suivantes:

- action mettre en attente (voir 6.16);
- action mettre en attente avec cas (voir 6.17);
- action recevoir signal avec cas (voir 6.19.2);
- action recevoir tampon avec cas (voir 6.19.3);
- action envoyer tampon (voir 6.18.3);
- action appeler une procédure de composante d'un locus de région (voir 3.15.3);
- action appeler une procédure de composante d'un locus de tâche au cas où il n'y a pas assez de mémoire pour exécuter l'étape c) 2) du 6.7 (voir 3.15.4).

Quand un fil d'exécution est mis en attente pendant que le contrôle réside dans une procédure **critique** ou une procédure de **composante critique**, la région associée sera libérée. Le contexte dynamique du fil d'exécution sera retenu jusqu'à ce que celui-ci soit réactivé. Le fil d'exécution essaie alors de verrouiller à nouveau la région ou le locus de région, ce qui peut provoquer son interruption.

11.4 Réactivation d'un fil d'exécution

Un fil d'exécution en attente peut être réactivé lorsqu'il est soumis à une supervision temporelle et qu'une interruption se produit (voir l'article 9). Il peut également être réactivé si un autre fil d'exécution exécute une des actions ci-après:

- action continuer (voir 6.15);
- action envoyer signal (voir 6.18.2);
- action envoyer tampon (voir 6.18.3);
- action recevoir tampon (voir 6.19.3);
- libération d'un locus de région (voir 3.15.3);
- au début de l'exécution d'une procédure de composante, appelée de l'extérieur, d'un locus de tâche (voir 3.15.4).

Quand un fil d'exécution, qui a verrouillé une région, réactive un autre fil d'exécution, il reste actif, c'est-à-dire qu'il ne libérera pas la région à cet endroit.

11.5 Enoncés de définition de signal

syntaxe:

<énoncé de définition de signal> ::= *(1)*
SIGNAL *<définition de signal>* { , *<définition de signal>* }* ; *(1.1)*

<définition de signal> ::= *(2)*
<occurence de définition> [= (*<mode>* { , *<mode>* }*)] [**TO** *<nom de processus>*] *(2.1)*

sémantique: une définition de signal définit une fonction de composition et décomposition pour des valeurs à transmettre entre processus. Si un signal est envoyé, la liste des valeurs spécifiée est transmise. Si aucun processus n'est en attente du signal dans une action recevoir avec cas, les valeurs sont gardées jusqu'à ce qu'un processus les reçoive.

propriétés statiques: une *occurence de définition* dans une *définition de signal* définit un nom de **signal**.

Un nom de **signal** a les propriétés suivantes:

- il a une liste facultative de modes qui sont les modes mentionnés dans la *définition de signal*;
- il a un nom de **processus** facultatif qui est le *nom de processus* spécifié après **TO**.

conditions statiques: aucun *mode* dans un *énoncé de définition de signal* ne peut avoir la **propriété de non-valeur**.

exemple:

15.27 **SIGNAL** *initiate* = (*INSTANCE*),

terminate; *(1.1)*

11.6 Fin des locus de région et de tâche

Un locus L de région "REGION" et un locus L de tâche "TASK" contiennent des files d'attente formées de fils d'exécution attendant d'exécuter une procédure de L.

Après sa création, L est *ouvert*, c'est-à-dire que les appels qui ne peuvent pas être exécutés immédiatement sont placés dans la file d'attente de L. Si L est un objet REGION, le fil d'exécution appelant est également bloqué.

Un RTL (locus de REGION ou de TASK) peut être fermé. S'il est *fermé*, aucun appel n'est mis en attente, mais une exception se produit à l'action appeler correspondante.

Un locus L de RTL est exécuté jusqu'à ce que toutes ses files d'attente soient vides, puis il est mis à l'état *vide*. Si tous les RTL qui dépendent de L (voir 12.2.6) sont terminés, le locus L lui-même est *terminé*.

Un RTL terminé peut être détruit. Si le RTL est dans un état "ouvert", "fermé" ou "vide", il ne peut pas être détruit.

12 Propriétés sémantiques générales

12.1 Règles de vérification des modes

12.1.1 Propriétés des modes et des classes

12.1.1.1 Propriété de protection

Informel

Un mode a la **propriété de protection** si c'est un mode **protégé** ou s'il contient une composante, ou une sous-composante, etc. qui est un mode **protégé**.

Définition

Un mode a la **propriété de protection** si et seulement si c'est:

- un mode matrice ayant un mode **élément** qui a la **propriété de protection**;
- un mode structure dont au moins un des modes **champ** a la **propriété de protection**, où le champ n'est pas un champ **étiquette** ayant un mode **protégé** implicitement d'un mode **structure paramétré**;
- un mode **protégé**.

12.1.1.2 Modes paramétrables

Informel

Un mode est **paramétrable** s'il peut être paramétré.

Définition

Un mode est **paramétrable** si et seulement si c'est:

- un mode chaîne;
- un mode matrice;
- un mode structure **variable paramétrable**.

12.1.1.3 Propriété de référencer

Informel

Un mode a la **propriété de référencer** si c'est un mode référence ou s'il contient une composante ou une sous-composante, etc. qui est un mode référence.

Définition

Un mode a la **propriété de référencer** si et seulement si c'est:

- un mode référence;
- un mode matrice avec un mode **élément** qui a la **propriété de référencer**;
- un mode structure dont au moins un des modes **champ** a la **propriété de référencer**.

12.1.1.4 Propriété d'étiquetage et de paramétrage

Informel

Un mode a la **propriété d'étiquetage** si c'est un mode structure **paramétré avec étiquettes** ou s'il contient une composante ou une sous-composante, etc, qui est un mode structure **paramétré avec étiquettes**.

Définition

Un mode a la **propriété d'étiquetage et de paramétrage** si et seulement si c'est:

- un mode matrice ayant un mode **élément** qui a la **propriété d'étiquetage** et de **paramétrage**;
- un mode structure dont au moins un des modes **champ** a la propriété d'étiquetage et de **paramétrage**;
- un mode structure **paramétré avec étiquettes**.

12.1.1.5 Propriété de non-valeur

Informel

Un mode a la **propriété de non-valeur** s'il n'existe, pour ce mode, aucune expression ni dénotation de valeur primitive.

Définition

Un mode a la **propriété de non-valeur** si et seulement si c'est:

- un mode événement, un mode tampon, un mode accès, un mode association ou un mode texte;
- un mode matrice ayant un mode **élément** qui a la **propriété de non-valeur**;
- un mode structure dont au moins un des modes **champ** a la **propriété de non-valeur**;
- un mode moreta **non_assignable**;
- un mode moreta **abstrait**;
- un mode moreta dont une composante au moins a la **propriété de non-valeur.**

12.1.1.6 Mode racine

Tout mode M a un mode **racine** défini de la façon suivante:

- si M n'est pas un mode intervalle discret ou un mode intervalle à virgule flottante;
- le mode **parent** de M, si M est un mode intervalle discret ou un mode intervalle à virgule flottante.

Toute M-classe par valeur ou M-classe par dérivation a un mode **racine** qui est le mode **racine** de M.

12.1.1.7 Classe résultante

Etant donné deux classes **compatibles** (voir 12.1.2.16), qui sont soit la classe **toute**, soit une M-classe par valeur, soit une M-classe par dérivation, où M et N sont des modes discret, à virgule flottante, ensembliste ou chaîne, la **classe résultante** est définie comme suit:

- la **classe résultante** de la M-classe par valeur et de la N-classe par valeur est la R-classe par valeur;
- la **classe résultante** de la M-classe par valeur et de la N-classe par dérivation ou la classe **toute** est la P-classe par valeur;
- la **classe résultante** de la M-classe par dérivation et de la N-classe par dérivation est la R-classe par dérivation;

- la **classe résultante** de la M-classe par dérivation et de la classe **toute** est la P-classe par dérivation;
- la **classe résultante** la classe **toute** et de la classe **toute** est la classe **toute**;

R étant le mode **résultant** de M et de N, et P le mode **racine** de M.

Etant donné deux modes M et N **similaires**, le mode **résultant** R est défini ainsi:

- si le mode **racine** de l'un est un mode chaîne **fixe** et l'autre un mode chaîne **variable**, c'est le mode **racine** de celui (entre M et N) dont le mode **racine** est un mode chaîne **variable**;
- sinon c'est P.

Etant donné une liste C_i de classes **compatibles** deux à deux (i = 1, …, n), la **classe résultante** de la liste de classes est définie récursivement comme, si n> 1, la **classe résultante** de la **classe résultante** de la liste de classes C_i (i = 1, …, n – 1) et de la classe C_n, sinon la **classe résultante** de C_1 et de C_1.

12.1.2 Relations entre modes et classes

12.1.2.1 Généralités

Dans les paragraphes qui suivent, les relations de compatibilité sont définies entre les modes, entre les classes, et entre les modes et les classes. Ces relations sont employées partout dans la présente Recommandation | Norme internationale pour définir les conditions statiques.

Les relations de compatibilité elles-mêmes sont définies en termes d'autres relations qui, dans le présent article, sont employées principalement dans ce but.

12.1.2.2 Relations d'équivalence sur les modes

Informel

Les relations d'équivalence suivantes jouent un rôle dans la formulation des relations de compatibilité:

- deux modes sont **similaires** s'ils sont de la même sorte, c'est-à-dire s'ils ont les mêmes propriétés héréditaires;
- deux modes sont **v-équivalents** (équivalents en valeur) s'ils sont **similaires** et ont aussi la même **nouveauté**;
- deux modes sont **équivalents** s'ils sont **v-équivalents** et si on prend aussi en considération les différences possibles dans la représentation des valeurs en mémoire ou dans la taille minimale de mémoire;
- deux modes sont **l-équivalents** (équivalents en locus) s'ils sont **équivalents** et ont la même spécification de **protection**;
- deux modes sont **semblables** s'ils sont impossibles à distinguer, c'est-à-dire si toutes les opérations qui peuvent être appliquées aux objets de l'un de ces modes peuvent être appliquées à celles de l'autre, à condition de ne pas tenir compte de la **nouveauté**;
- deux modes sont dits **liés par la nouveauté** s'ils sont **semblables** et ont une spécification de **nouveauté** égale.

Définition

Dans les paragraphes qui suivent, on donne les relations d'équivalence entre modes sous la forme d'un ensemble de relations (partielles). On obtient l'algorithme d'équivalence complet en prenant la fermeture symétrique, réflexive et transitive de cet ensemble de relations. Les modes mentionnés dans les relations peuvent être introduits virtuellement ou dynamiques. Dans ce dernier cas, la vérification complète d'équivalence peut seulement être faite à l'exécution. Une détection d'anomalie dans la partie dynamique de la vérification donnera lieu à l'exception *RANGEFAIL* ou *TAGFAIL* (voir les paragraphes appropriés).

Vérifier l'équivalence de deux modes récursifs exige la vérification de l'équivalence des modes associés dans les chemins correspondants de l'ensemble des modes récursifs par lequel ils sont définis. Les modes sont équivalents si aucune contradiction n'est trouvée. (En conséquence, un chemin de l'algorithme de vérification s'arrête avec succès si deux modes sont comparés, qui l'avaient été auparavant.)

12.1.2.3 La relation similaire

Deux modes sont **similaires** si et seulement si:

- ce sont des modes entier;
- ce sont des modes à virgule flottante;
- ce sont des modes booléen;

- ce sont des modes caractère;
- ce sont des modes ensemble du type suivant:
  1) ils définissent le même **nombre de valeurs**;
  2) pour chaque nom d'**élément d'ensemble** défini par un mode, il existe un nom d'**élément d'ensemble** défini par l'autre mode qui a la même *chaîne* de nom et la même valeur de représentation;
  3) ils sont tous deux des modes ensemble **avec numéros** ou des modes ensemble **sans numéros**;
- ce sont des modes intervalle discret qui ont des modes **parents similaires**;
- ce sont des modes intervalle à virgule flottante;
- l'un est un mode intervalle discret ou un mode intervalle à virgule flottante dont le mode **parent** est **similaire** à l'autre;
- ce sont des modes ensembliste dont les modes **primitifs** sont **équivalents**;
- ce sont des modes référence liée tels que leurs modes **référencés** sont **équivalents**;
- ce sont des modes référence libre;
- ce sont des modes descripteur dont les modes **référencés originels** sont **équivalents**;
- ce sont des modes procédure du type suivant:
  1) ils ont le même nombre de **specs de paramètre** et les **specs de paramètre** correspondantes (par position) ont des modes **l-équivalents**, les mêmes attributs de paramètre, si présents;
  2) tous deux ont ou n'ont pas une **spec de résultat**. Si présentes, les **specs de résultat** doivent avoir des modes **l-équivalents** et les mêmes attributs, si présents;
  3) ils ont la même liste de noms d'**exception**;
  4) ils ont la même **récursivité**;
- ce sont des modes instance;
- ce sont des modes événement qui ont soit la même **longueur d'événement**, soit n'en ont pas;
- ce sont des modes tampon du type suivant:
  1) tous deux n'ont pas de **longueur de tampon** ou ont la même;
  2) ils ont des modes **élément de tampon** qui sont **l-équivalents**;
- ce sont des modes association;
- ce sont des modes accès du type suivant:
  1) tous deux ont des modes **indice équivalents** ou n'en ont pas;
  2) au moins un n'a pas de mode **enregistrement**, ou tous deux ont des modes **enregistrement** qui sont **l-équivalents** et qui sont tous deux soit des modes **enregistrement statiques** soit des modes **enregistrement dynamiques**;
- ce sont des modes texte du type suivant:
  1) ils ont la même **longueur de texte**;
  2) ils ont des modes **enregistrement de texte l-équivalents**;
  3) ils ont des modes **accès l-équivalents**;
- ce sont des modes durée;
- ce sont des modes temps absolu;
- ce sont des modes chaîne tels que leurs modes **élément** sont **équivalents**;
- ce sont des modes matrice du type suivant:
  1) leurs modes **indice** sont **v-équivalents**;
  2) leurs modes **élément** sont **équivalents**;
  3) leurs **implantations d'élément** sont **équivalentes**;
  4) ils ont le même **nombre d'éléments**. Cette vérification est dynamique si l'un ou les deux modes sont dynamiques. Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL*;

- ce sont des modes structure qui ne sont pas des modes structure **paramétrés** du type suivant:
  1) en syntaxe stricte, ils ont le même nombre de *champs* et les *champs* correspondants (par position) sont **équivalents**;
  2) si ce sont tous les deux des modes structure **variable paramétrables**, leurs listes de classes doivent être **compatibles**;
- ce sont des modes structure **paramétrés** du type suivant:
  1) leurs modes structure **variable originels** sont **similaires**;
  2) les valeurs correspondantes (par position) sont les mêmes. Cette vérification est dynamique si l'un ou les deux modes sont dynamiques. Une détection d'anomalie donnera lieu à l'exception *TAGFAIL*.
- ce sont des modes moreta ayant des noms de mode synonymes.

12.1.2.4 La relation v-équivalent

Deux modes sont **v-équivalents** si et seulement s'ils sont **similaires** et ont la même **nouveauté**.

12.1.2.5 La relation équivalent

Deux modes sont **équivalents** si et seulement s'ils sont **v-équivalents** et:

- si l'un est un mode intervalle discret, l'autre mode doit aussi être un mode intervalle discret et les deux **bornes supérieures** doivent être égales ainsi que les deux **bornes inférieures**;
- si l'un est un mode intervalle à virgule flottante, l'autre mode doit aussi être un mode intervalle à virgule flottante; ils doivent avoir les mêmes **bornes supérieures**, les mêmes **bornes inférieures** et la même **précision**;
- si l'un est un mode chaîne **fixe**, l'autre doit être aussi un mode chaîne **fixe** et ils doivent avoir la même **longueur de chaîne**. Cette vérification est dynamique au cas où l'un des modes, ou les deux, sont dynamiques. Un échec de vérification a pour résultat l'exception *RANGEFAIL*;
- si l'un est un mode chaîne **variable**, l'autre doit l'être également et ils doivent avoir la même **longueur de chaîne**. Cette vérification est dynamique au cas où l'un des modes, ou les deux, sont dynamiques. Un échec de vérification a pour résultat l'exception *RANGEFAIL*.

12.1.2.6 La relation l-équivalent

Deux modes sont **l-équivalents** si et seulement s'ils sont **équivalents** et, si l'un a le mode **protection**, l'autre doit l'avoir aussi, et:

- si les deux sont des modes référence liée, leurs modes **référencés** doivent être **l-équivalents**;
- si les deux sont des modes descripteur, leurs modes **référencés originels** doivent être **l-équivalents**;
- si les deux sont des modes matrice, leurs modes **élément** doivent être **l-équivalents**;
- si ce sont des modes structure qui ne sont pas des modes structure **paramétrés**, les *champs* correspondants (par position) en syntaxe stricte doivent être **l-équivalents**; si ce sont des modes structure **paramétrés**, leurs modes structure **variables originels** doivent être **l-équivalents**.

12.1.2.7 Les relations équivalent et l-équivalent pour les champs

Deux *champs* (tous les deux pris dans le contexte de deux modes structure donnés), sont 1. **équivalents**, 2. **l-équivalents** si et seulement s'ils sont tous les deux des *champs fixes* qui sont 1. **équivalents**, 2. **l-équivalents** ou s'ils sont tous les deux des *champs alternatifs* qui sont 1. **équivalents**, 2. **l-équivalents**.

Les relations **équivalent** et **l-équivalent** sont définies récursivement pour, respectivement, des *champs fixes*, des *champs récurrents*, des *champs alternatifs* et des *alternatives variant* correspondants et cela, de la manière suivante:

- *Champs fixes* et *champs récurrents*
  1) Les deux *champs fixes et récurrents* doivent avoir des **implantations de champ équivalentes**.
  2) Les modes des deux **champs** doivent être 1. **équivalents**, 2. **l-équivalents**.
- *Champs alternatifs*
  1) Les deux *champs alternatifs* ont des *listes d'étiquettes* ou les deux n'en ont pas. Dans le premier cas, les *listes d'étiquettes* doivent avoir le même nombre de noms de **champs étiquettes** et les noms de **champs étiquettes** correspondants (par position) doivent dénoter des *champs fixes* correspondants.

2) Les deux doivent avoir le même nombre de *alternatives variant* et les *alternatives variant* correspondants (par position) doivent être 1. **équivalents**, 2. **l-équivalents**.

3) Les deux doivent ne pas avoir de spécification de **ELSE** ou les deux doivent l'avoir. Dans le deuxième cas, le même nombre de *champs récurrents* doit suivre et les *champs récurrents* correspondants (par position) doivent être 1. **équivalents**, 2. **l-équivalents**.

- *Alternatives variant*

1) Les deux *alternatives variant* doivent avoir le même nombre de *listes d'étiquettes de cas* et les *listes d'étiquettes de cas* correspondantes (par position) doivent être toutes les deux *indifférentes*, ou toutes les deux définir le même ensemble de valeurs.

2) Les deux *alternatives variant* doivent avoir le même nombre de *champs récurrents* et les *champs récurrents* correspondants (par position) doivent être 1. **équivalents**, 2. **l-équivalents**.

12.1.2.8 La relation équivalent pour les implantations

Dans la suite, on admettra que chaque *pos* est de la forme:

POS (<*nombre*>, <*bit initial*>, <*longueur*>)

et que chaque *pas* est de la forme:

STEP (<*pos*>, <*taille de pas*>)

Le paragraphe 3.13.5 donne les règles appropriées pour donner à *pos* ou à *pas* la forme désirée.

- *Implantation de champ*

Deux **implantations de champ** sont **équivalentes** si elles sont toutes les deux **NOPACK**, ou toutes les deux **PACK**, ou toutes les deux *pos*. Dans le dernier cas, le premier *pos* doit être **équivalent** au second (voir plus loin).

- *Implantation d'élément*

Deux **implantations d'élément** sont **équivalentes** si elles sont toutes les deux **NOPACK**, ou toutes les deux **PACK**, ou toutes les deux *pas*. Dans ce dernier cas, le *pos* dans le premier *pas* doit être **équivalent** au *pos* dans le second *pas* (voir plus loin) et la *taille de pas* doit donner la même valeur pour les deux **implantations d'élément**.

- *Pos*

Un *pos* est **équivalent** à un autre *pos* si et seulement si les deux occurrences de *mot* donnent la même valeur, les deux occurrences de *bit initial* donnent la même valeur, et les deux occurrences de *longueur* donnent la même valeur.

12.1.2.9 La relation semblable

Deux modes sont **semblables** si et seulement si tous deux sont ou ne sont pas des modes **protégés** et s'ils ont tous deux la **nouveauté nulle** ou si tous deux ont la même **nouveauté** et que:

- ce sont des modes entier;
- ce sont des modes booléen;
- ce sont des modes caractère;
- ce sont des modes ensemble **similaires**;
- ce sont des modes intervalle discret ayant des **bornes supérieures** égales et des **bornes inférieures** égales;
- ce sont des modes intervalle à virgule flottante ayant les mêmes **bornes supérieures**, les mêmes **bornes inférieures** et la même **précision**;
- ce sont des modes ensembliste dont les modes **primitifs** sont **semblables**;
- ce sont des modes référence liée dont les modes **référence** sont **semblables**;
- ce sont des modes référence libre;
- ce sont des modes descripteur dont les modes **référencés originels** sont **semblables**;

- ce sont des modes procédure du type suivant:
  1) ils ont le même nombre de **specs de paramètre** et les **specs de paramètre** correspondantes (par position) ont des modes **semblables**, les mêmes attributs de paramètre, si présents;
  2) ils ont ou n'ont pas de **spec de résultat**. Si présentes, les **specs de résultat** doivent avoir des modes **semblables** et les mêmes attributs, si présents;
  3) ils ont la même liste de noms **d'exception**;
  4) ils ont la même **récursivité**;
- ce sont des modes instance;
- ce sont des modes événement qui ont tous deux la même **longueur d'événement** ou n'en n'ont pas;
- ce sont des modes tampon du type suivant:
  1) tous deux ont la même **longueur tampon** ou n'en ont pas;
  2) ils ont des modes **élément tampon** qui sont **semblables**;
- ce sont des modes association;
- ce sont des modes accès du type suivant:
  1) tous deux ont des modes **indice semblables** ou n'en ont pas;
  2) un au moins n'a pas de mode **enregistrement** ou tous deux ont des modes **enregistrement** qui sont **semblables** et qui sont tous deux des modes **enregistrement statiques** ou des modes **enregistrement dynamiques**;
- ce sont des modes texte du type suivant:
  1) ils ont la même **longueur de texte**;
  2) leurs modes **enregistrement de texte** sont **semblables**;
  3) leurs modes **accès** sont **semblables**;
- ce sont des modes durée;
- ce sont des modes temps absolu;
- ce sont des modes chaîne du type suivant:
  1) leurs modes **élément** sont **semblables**;
  2) ils ont la même **longueur de chaîne**;
  3) ils sont tous deux des modes chaîne **fixe** ou **variable**;
- ce sont des modes matrice du type suivant:
  1) leurs modes **indice** sont **semblables**;
  2) leurs modes **élément** sont **semblables**;
  3) leurs **implantations des éléments** sont **équivalentes**;
  4) ils ont le même **nombre d'éléments**;
- ce sont des modes structure qui ne sont pas des modes structure **paramétrés** du type suivant:
  1) en syntaxe stricte, ils ont le même nombre de *champs* et les *champs* correspondants (par position) sont **semblables**;
  2) s'ils sont tous deux des modes structure **variables paramétrables**, leurs listes de classes doivent être **compatibles**;
- ce sont des modes structure **paramétrés** du type suivant:
  1) leurs modes structure **variables originels** sont **semblables**;
  2) leurs valeurs correspondantes (par position) sont les mêmes.

12.1.2.10 Les relations semblables pour les champs

Deux *champs* (tous deux dans le contexte de deux modes structure donnés) sont **semblables** si et seulement s'ils sont tous deux des *champs fixes* qui sont **semblables** ou tous deux des *champs alternatifs* qui sont **semblables**.

La relation **semblable** est définie récursivement pour, respectivement, des *champs fixes*, des *champs récurrents*, des *champs alternatifs* et des *alternatives variant* correspondants et cela, respectivement de la manière suivante:

- *champs fixes* et *champs récurrents*
  1) Les deux *champs fixes* ou *variables* doivent avoir une **implantation de champ équivalente**;
  2) Les deux modes de **champs** doivent être **semblables**;
  3) Les deux *champs fixes ou variables* doivent avoir la même *chaîne de nom*;
- *champs alternatifs*
  1) les *champs alternatifs* doivent avoir tous deux des *listes d'étiquettes* ou ne pas en avoir. Dans le premier cas, les *listes d'étiquettes* doivent avoir le même nombre de noms de **champ étiquette** et les noms de **champ étiquette** correspondants (par position) doivent dénoter des *champs fixes* correspondants;
  2) tous deux doivent avoir le même nombre de *alternatives variant* et les *alternatives variant* correspondants (par position) doivent être **semblables**;
  3) tous deux peuvent ne pas avoir de spécification **ELSE** ou tous deux doivent avoir la spécification **ELSE**. Dans ce dernier cas, le même nombre de *champs récurrents* doit suivre et les *champs récurrents* correspondants (par position) doivent être **semblables**;
- *alternatives variant*
  1) les deux *alternatives variant* doivent avoir le même nombre de *listes d'étiquettes de cas* et les *listes d'étiquettes de cas* correspondantes (par position) doivent soit être toutes deux *indifférentes* soit définir toutes deux le même ensemble de valeurs;
  2) deux *alternatives variant* doivent avoir le même nombre de *champs récurrents* et des *champs récurrents* correspondants (par position) doivent être **semblables**.

12.1.2.11 La relation liée par la nouveauté

Informel

Dans un programme, chaque **quasi**-néomode doit représenter au plus un néomode **réel**. Cela s'établit ainsi: quand une *chaîne de nom* est **liée** à la fois à une définition **réelle** et à une **quasi**-*occurence de définition*, tous les néomodes impliqués sont appariés. La relation **liée par la nouveauté** est alors établie entre **nouveautés**.

Définition

La relation appariée par la **nouveauté** s'applique entre deux modes et un domaine. Pour chaque *chaîne de nom* **liée** dans un domaine R à la fois à une définition **réelle** et à une **quasi**-*occurence de définition*:

- si ce sont des noms de **synonyme**, les modes **racine** de leurs classes sont **appariés par la nouveauté** dans R;
- si ce sont des noms de **locus** ou d'**identité de locus**, les modes de locus sont **appariés par la nouveauté** dans R;
- si ce sont des noms de **procédure**, les modes des **specs de paramètre** et de la **spec de résultat**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des noms de **processus**, les modes des **specs de paramètre** sont **appariés par la nouveauté** dans R;
- si ce sont des noms de **signal**, les modes dans la liste de modes sont **appariés par la nouveauté** dans R.

Si deux modes sont **appariés par la nouveauté** dans un domaine R, alors:

- si ce sont des modes ensembliste, leurs modes **primitifs** sont **appariés par la nouveauté** dans R;
- si ce sont des modes référence liée, leurs modes **référencés** sont **appariés par la nouveauté** dans R;
- si ce sont des modes descripteur, leurs modes **référencés originels** sont **appariés par la nouveauté** dans R;
- si ce sont des modes procédure, les modes de leurs **specs de paramètre** et de la **spec de résultat**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des modes tampon, leurs modes **élément tampon** sont **appariés par la nouveauté** dans R;
- si ce sont des modes accès, leurs modes **indice**, si présents, et modes **enregistrement**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des modes texte, leurs modes **indice**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des modes matrice, leurs modes **indice** et modes **élément** sont **appariés par la nouveauté** dans R;

- si ce sont des modes structure **paramétrés**, leurs modes de structure **variable originel** sont **appariés par la nouveauté** dans R;
- si ce sont des modes structure **variable paramétrable**, leurs modes **champ** et les modes des classes de leurs listes de classes sont **appariés par la nouveauté** dans R;
- sinon, si ce sont des modes structure, leurs modes **champ** sont **appariés par la nouveauté** dans R.

Si deux modes sont **appariés par la nouveauté** dans un domaine R et que leurs **nouveautés** ne sont pas égales, la nouveauté **réelle** et la **quasi-nouveauté** des modes sont **liées par la nouveauté** entre elles dans R.

Deux **nouveautés** sont considérées identiques s'il s'agit:

- de la même **nouveauté réelle**;
- d'une **nouveauté réelle** et d'une **quasi-nouveauté** qui sont **liées par la nouveauté**.

12.1.2.12 La relation compatible en lecture

Informel

La relation **compatible en lecture** est applicable à des modes **équivalents**. On dit qu'un mode M est **compatible en lecture** avec un mode N si lui ou ses (sous-)composantes éventuelles ont des spécifications de **protection** égales ou plus restrictives et, si ce sont des modes référence, renvoient à des locus **l-équivalents**. Cette relation est donc asymétrique.

Exemple:

READ REF READ *CHAR* est **compatible en lecture** avec **REF READ** *CHAR*

Définition

Un mode M est dit être **compatible en lecture** avec un mode N (relation asymétrique) si et seulement si M et N sont **équivalents** et, si N est un mode **protégé**, alors M doit aussi être un mode **protégé**, et de plus:

- si M et N sont des modes référence liée, le mode **référencé** de M doit être **l-équivalent** avec le mode **référencé** de N;
- si M et N sont des modes descripteur, le mode **référencé originel** de M doit être **l-équivalent** avec le mode **référencé originel** de N;
- si M et N sont des modes matrice, le mode **élément** de M doit être **compatible en lecture** avec le mode **élément** de N;
- si M et N sont des modes structure qui ne sont pas des modes structure **paramétrés**, tout mode **champ** de M doit être **compatible en lecture** avec le mode **champ** correspondant de N. Si M et N sont des modes structure **paramétrés**, le mode structure **variable originel** de M doit être **compatible en lecture** avec le mode structure **variable originel** de N.

12.1.2.13 Les relations équivalent dynamique et compatible en lecture dynamique

Informel

Les relations 1. **équivalent dynamique** et 2. **compatible en lecture dynamique** ne s'appliquent qu'aux modes qui peuvent être dynamiques, c'est-à-dire les modes chaîne, matrice et structure **variable**. Un mode M **paramétrable** est dit 1. **équivalent dynamique**, 2. **compatible en lecture dynamique** avec un mode N (éventuellement dynamique) s'il existe une version dynamiquement paramétrée de M qui est 1. **équivalent**, 2. **compatible en lecture** avec N.

Définition

Un mode M est 1. **équivalent dynamique** d'un mode N, 2. **compatible en lecture dynamique** avec un mode N (relation asymétrique) si et seulement si l'une des conditions ci-après se vérifie:

- M et N sont des modes chaîne tels que M*(p)* est 1. **équivalent**, 2. **compatible en lecture** avec N, où *p* est la longueur (éventuellement dynamique) de N. La valeur *p* ne doit pas être supérieure à la **longueur de chaîne** de M. Cette vérification est dynamique si N est un mode dynamique. Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL*.
- M et N sont des modes matrice tels que M*(p)* est 1. **équivalent**, 2. **compatible en lecture** avec N, où *p* est tel que *NUM(p) – LOWER* (M) + 1 est le **nombre d'éléments** (éventuellement dynamiques) de N. La valeur *p* ne doit pas être supérieure à la **borne supérieure** de M. Cette vérification est dynamique si N est un mode dynamique. Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL*.
- M est un mode structure **variable paramétrable** et N est un mode structure **paramétré** tel que M(p_1, ..., p_n) est 1. **équivalent**, 2. **compatible en lecture** avec N, où p_1, ..., p_n désigne la liste des valeurs de N.

12.1.2.14 La relation limitable à

Informel

La relation **limitable à** est applicable à des modes **équivalents** qui ont la **propriété de référencer**. On dit qu'un mode M est **limitable à** un mode N si lui ou ses (sous-)composantes éventuelles référencent des locus qui ont des spécifications de **protection** égales ou plus restrictives que ceux référencés par N. Cette relation est donc asymétrique.

Exemple:

REF READ *INT* est **limitable à REF** *INT*

STRUCT (*P **REF READ** BOOL)* est **limitable à STRUCT** *(Q **REF** BOOL)*

Définition

Un mode M est **limitable à** un mode N (relation asymétrique) si et seulement si M et N sont **équivalents** et que, de plus:

- si M et N sont des modes référence liée, le mode **référencé** de M doit être **compatible en lecture** avec le mode **référencé** de N;
- si M et N sont des modes descripteur, le mode **référencé originel** de M doit être **compatible en lecture** avec le mode **référencé originel** de N;
- si M et N sont des modes matrice, le mode **élément** de M doit être **limitable au** mode **élément** de N;
- si M et N sont des modes structure, chaque mode **champ** de M doit être **limitable au** mode **champ** correspondant de N.

12.1.2.15 Compatibilité entre un mode et une classe

- tout mode M est **compatible** avec la classe **toute**;
- un mode M est **compatible** avec la classe **nulle** si et seulement si M est un mode référencé, un mode procédure ou un mode instance;
- un mode M est **compatible** avec la N-classe par référence si et seulement si c'est un mode référence et l'une des conditions suivantes est satisfaite:
  1) N est un mode non moreta statique et M est un mode référence liée dont le mode **référencé** est **compatible en lecture** avec N;
  2) N est un mode moreta statique et M est un mode référence liée REF MM et MM et N sont sur le même chemin;
  3) N est un mode statique et M est un mode référence libre;
  4) M est un mode descripteur dont le mode **référencé originel** est **compatible en lecture dynamique** avec N;
- un mode M est **compatible** avec une N-classe par dérivation si et seulement si M et N sont **similaires**;
- un mode M est **compatible** avec une N-classe par valeur si et seulement si l'une des conditions suivantes est satisfaite:
  1) si M n'a pas la **propriété de référencer**, M et N doivent être **v-équivalents**;
  2) si M a la **propriété de référencer**, M doit être **limitable** à N.

12.1.2.16 Compatibilité entre classes

- Toute classe est **compatible** avec elle-même.
- La classe **toute** est **compatible** avec toute autre classe.
- La classe **nulle** est **compatible** avec toute M-classe par référence.
- La classe **nulle** est **compatible** avec la M-classe par dérivation ou la M-classe par valeur si et seulement si M est un mode référence, un mode procédure ou un mode instance.
- La M-classe par référence est **compatible** avec la N-classe par référence si et seulement si M et N sont **équivalents**. Si M et/ou N est (sont) un mode dynamique, la partie dynamique de la vérification est ignorée, c'est-à-dire aucune exception ne peut être causée.
- La M-classe par référence est **compatible** avec la N-classe par valeur si et seulement si N est un mode référence et l'une des conditions suivantes est satisfaite:
  1) M est un mode statique et N est un mode référence lié dont le mode **référencé** est **équivalent** à M.
  2) M est un mode statique et N est un mode référence libre.
  3) N est un mode descripteur dont le mode **référencé originel** est **équivalent dynamique** de M.

- La M-classe par dérivation est **compatible** avec la N-classe par dérivation ou la N-classe par valeur si et seulement si M et N sont **similaires**.
- La M-classe par valeur est **compatible** avec la N-classe par valeur, si et seulement si M et N sont **v-équivalents**.

Deux listes de classes sont **compatibles** si et seulement si les deux listes ont le même nombre de classes et les classes correspondantes (par position) sont **compatibles**.

12.1.2.17 Conformité des noms de mode

Deux noms de mode A et B sont conformes l'un à l'autre si et seulement si:

- tous deux dénotent des modes de type "REF MM", où MM est un mode moreta et A et B sont sur le même chemin;
- ou si A et B sont synonymes.

12.1.3 Définitions pour les modes moreta

Si M est un mode moreta:

M_S = la partie spécification de M (également l'ensemble de composantes de cette partie);

M_B = la partie corps de M (également l'ensemble de composantes de cette partie);

M_p = l'ensemble de composantes publiques de M_S définies directement dans M_S;

M_{p+} = l'ensemble de toutes les composantes publiques de M_S (y compris les composantes héritées);

M_I = l'ensemble de composantes internes de M_S;

M_{I+} = l'ensemble de toutes les composantes internes de M_S (y compris les composantes héritées) ;

M_R = l'ensemble de composantes privées de M_B;

M_{R+} = l'ensemble de toutes les composantes privées de M_S (y compris les composantes héritées);

M_{CD} = l'ensemble de constructeurs et de destructeurs de M_S;

M_{inv} = l'invariant de M_S;

M_O = l'ensemble des composantes (logiques) contenues dans un locus de mode M.

Si P est une procédure de composante d'un mode moreta:

PS = la partie signature de P;

PD = la définition (complète) de P;

Ppre = la précondition de P;

Ppost = la postcondition de P;

PE = l'ensemble d'exceptions spécifié dans PS.

Si X est une procédure ou un mode moreta:

attr(X, A) = X contient l'attribut A: par exemple attr(P, INLINE);

prop(X, P) = X a la propriété P: par exemple prop(P, assignable);

GRANDTed = exporté explicitement;

granted = signifie GRANTed $\vee$ exporté explicitement.

12.1.3.1 Noms qualifiés de composantes de mode moreta et de locus moreta

Si M est la chaîne de nom simple d'un mode moreta, que L est la chaîne de nom simple d'un locus moreta et que C est le nom simple d'une composante de M ou d'une composante publique de L, on peut utiliser le même nom M.C ou L.C comme nom exclusif de la composante C afin de la distinguer des composantes ayant la même chaîne de nom simple. Au besoin, le nom qualifié est pris pour hypothèse.

12.1.3.2 Relations de successeur et de prédécesseur pour les noms de mode moreta

Un nom DM de mode moreta est un successeur direct (dsucc) d'un nom BM de mode moreta si et seulement si les noms D et B: (B syn BM) $\wedge$ (D syn DM) $\wedge$ existent (B est mentionné dans la clause d'héritage de D).

Un nom DM de mode moreta est un successeur (succ) d'un nom BM de mode moreta si et seulement DM syn BM ou ($\exists$MM: (DM succ MM) $\wedge$ (MM dsucc BM)).

La relation "prédécesseur" est l'inverse de "successeur".

Deux noms A et B de mode moreta se trouvent sur le même chemin si et seulement si (A succ B) ∨ (B succ A).

Ces relations sont valables au plan isomorphe pour les modes des types "REF MM", où MM est un mode moreta.

12.1.3.3 Correspondance entre signatures de procédure et définitions de procédure

Il y a correspondance entre une signature S de procédure protégée et une définition D de procédure protégée si et seulement si il y a correspondance entre:

S.<liste de paramètres> et D.<liste de paramètres formels> ∧

S.<spec de résultat> et D.<spec de résultat>sont diffèrent tout au plus dans l'occurrence de RESULT ∧

S.<liste d'exceptions> = D.<liste d'exceptions> ∧

S.<liste d'attributs de procédure protégée> = D.<liste d'attributs de procédure protégée>

Il y a correspondance entre une liste P de paramètres et une liste F de paramètres formels avec une syntaxe F′ stricte si et seulement si:

|P| = |F′| ∧

et si tous les éléments correspondants de P et F′ ont le même mode et les mêmes attributs de paramètre.

12.2 Visibilité et rattachement de nom

La définition de la visibilité et du rattachement de nom repose sur la terminologie suivante:

- *chaîne de nom*: désigne un symbole terminal auquel est attachée une *chaîne* de nom **canonique** (voir 2.7) et des propriétés de visibilité;
- nom: désigne une *chaîne de nom simple* associée à l'*occurence de définition* qui l'a créée (voir 10.1);
- *nom*: désigne une occurrence d'utilisation d'un nom (avec éventuellement une chaîne de nom préfixe).

12.2.1 Degrés de visibilité

Les rattachements de nom sont fondés sur la visibilité des *chaînes de nom* dans les domaines d'un programme. Dans un domaine, chaque *chaîne de nom* possède l'un des quatre degrés de visibilité suivants:

Tableau 1/Z.200 – Degrés de visibilité

Visibilité	Propriétés (informelles)
directe	La *chaîne de nom* est **visible** par création, octroi, saisie ou héritage de la spec au corps
indirecte	La *chaîne de nom* est prédéfinie ou héritée via une imbrication de bloc
publique	La *chaîne de nom* est le nom d'une composante publique d'un *mode moreta* utilisée dans un nom de composante moreta ou elle est le nom d'une composante d'un mode moreta M utilisée dans un nom de composante moreta qui survient à l'intérieur de M ou de tout successeur de M
privée	La *chaîne de nom* est le nom d'un énoncé P de définition de procédure partagée contenue dans un corps B de mode moreta et la spécification de mode moreta de B ne contient pas d'énoncé de signature de procédure protégée correspondante
invisible	La *chaîne de nom* ne peut pas être utilisée.

Une *chaîne de nom* est dite **visible** dans un domaine si elle est **directement visible** ou **indirectement visible** dans ce domaine. Autrement, le *nom* est dit être **invisible** dans ce domaine. Les énoncés de structuration du programme et les énoncés de visibilité déterminent d'une façon univoque à quelle classe de visibilité chaque *chaîne de nom* appartient.

Lorsqu'une *chaîne de nom* est **visible** dans un domaine, elle peut être **directement liée** à une autre *chaîne de nom* dans un autre domaine, ou **directement liée** à une *occurence de définition* dans le programme. Les règles de **liaison directe** sont énoncées au 12.2.3. On notera que toute utilisation d'une règle introduit une nouvelle **liaison directe** pour une *chaîne de nom*.

Sur la base de la **liaison directe**, la notion de **liaison** (non nécessairement **directe**) se définit comme suit:

une *chaîne de nom* N_1, **visible** dans le domaine R_1, est dite être **liée** à une *chaîne de nom* N_2 dans le domaine R_2 ou à une *occurence de définition* D, si et seulement si l'une des conditions suivantes se vérifie:

- N_1 dans R_1 est **directement lié** à N_2 en R_2 ou à D. Toutefois, si N_1 est **directement lié** à plusieurs *occurences de définitions* dans R_1, toutes ces *occurences de définitions* sauf une sont superflues et N_1 est **lié** arbitrairement à une de ces définitions dans R_1. Cela ne s'applique pas si N_1 est la *chaîne* de nom d'un énoncé de signature de procédure protégée simple dans une spécification de mode moreta.
- N_1 dans R_1 est **directement lié** à un N dans un R et, dans R, N est **lié** à N_2 en R_2, ou à D.

12.2.2 Conditions de visibilité et identification

Dans chaque domaine d'un programme, les conditions suivantes doivent être satisfaites:

- si une *chaîne de nom* est **visible** dans un domaine et si elle a plus d'une **liaison directe**, elle doit être **liée** à exactement une *occurence de définition* **réelle** et une **quasi**-*occurence de définition*, ou à exactement une *occurence de définition* **réelle** dans un énoncé de signature de procédure protégée **simple** dans un mode M qui n'est pas un mode interface, et exactement une *occurence de définition* **réelle** dans un énoncé de définition de procédure protégée **simple** correspondante, et éventuellement à plusieurs *occurences de définitions* **réelles** dans un énoncé de signature de procédure protégée **simple** dans un ou plusieurs modes interface qui sont des modes de base d'un mode moreta M, ou éventuellement à plusieurs *occurences de définitions* **réelles** dans un énoncé de signature de procédure protégée **simple** dans un ou plusieurs modes interface qui sont des modes de base d'un mode moreta M et où la *chaîne* de nom n'a pas de lien direct dans M.

Une *chaîne de nom* NS, **visible** dans le domaine R, est dite **liée** dans R à plusieurs *occurences de définitions* selon les règles suivantes:

- si NS est **visible** dans R, NS est **liée** aux *occurences de définitions* auxquelles elle est **liée** dans R (en tant que *chaîne de nom* **visible**). Si elle est **liée** à la fois à une **quasi**-*occurence de définition* et à une *occurence de définition* **réelle**, la **quasi**-*occurence de définition* est redondante et ne participe plus à la visibilité et à l'identification (c'est-à-dire qu'elle n'est pas saisie, octroyée ou héritée). Si elle est liée à exactement une *occurence de définition* **réelle** dans un énoncé de signature de procédure protégée **simple** dans un mode M qui n'est pas un mode interface et à exactement une *occurence de définition* **réelle** dans un énoncé de définition de procédure protégée **simple** correspondante, et éventuellement à plusieurs *occurences de définitions* **réelles** dans un énoncé de signature de procédure protégée **simple** dans un ou plusieurs modes interface qui sont des modes de base d'un mode moreta M, elle est liée à l'occurrence dans M.
- sinon, NS n'est pas **liée** dans R.

conditions statiques: la *chaîne de nom* attachée à chaque *nom* immédiatement englobé dans un domaine doit être **liée** dans ce domaine.

identification: un *nom* N ayant une *chaîne de nom* NS dans un domaine R est **lié** aux *occurences de définitions* auxquelles NS est **lié** dans R.

12.2.3 Visibilité dans les domaines

12.2.3.1 Généralités

Une *chaîne de nom* est **directement visible** dans un domaine, selon les règles suivantes:

- cette *chaîne de nom* est saisie dans le domaine (voir 12.2.3.5);
- cette *chaîne de nom* est octroyée dans le domaine (voir 12.2.3.4);
- il existe une *occurence de définition* ayant cette *chaîne de nom* dans le domaine. En pareil cas, la *chaîne de nom* dans le domaine est **directement liée** à l'*occurence de définition*. (A noter que la *chaîne de nom* peut être **directement liée** à plusieurs *occurences de définitions* dans le domaine);
- à l'intérieur d'un constructeur ou d'un destructeur CD d'un mode moreta M la *chaîne de nom* de M n'est pas cachée par l'occurence de définition de la même *chaîne de nom* dans la définition de CD (mais elle peut être cachée par d'autres occurrences de la même représentation);
- en un point d'un constructeur ou d'un destructeur CD d'un mode moreta M où la *chaîne de nom* S ou M n'est pas cachée, S dénote M ou CD, selon le contexte;
- le domaine est 1. un *corps de module*, 2. un *corps de région* et la *chaîne de nom* est **directement visible dans** le domaine 1. *d'un module de spec*, 2. *d'une région de spec* **correspondante**. La *chaîne de nom* est **directement liée** à la *chaîne de nom* dans le domaine correspondant.

Une *chaîne de nom* qui n'est pas **directement visible** dans un domaine y est **indirectement fortement visible**, selon les règles suivantes:

- le domaine est un bloc et la *chaîne de nom* est **visible** dans le domaine immédiatement englobant. La *chaîne de nom* est dite être héritée par le bloc et elle est **directement liée** à la même *chaîne de nom* dans le domaine immédiatement englobant.
- le domaine n'est pas un bloc dans lequel la *chaîne de nom* est héritée et la *chaîne de nom* est une *chaîne de nom* définie par le langage (voir III.2) ou par l'implémentation. La *chaîne de nom* est considérée être **directement liée** à une *occurence de définition* dans le domaine de la définition de processus imaginaire la plus externe pour sa signification prédéfinie.

12.2.3.2 Enoncés de visibilité

syntaxe:

<énoncé de visibilité> ::= *(1)*
<énoncé d'octroi> *(1.1)*
| *<énoncé de saisie>* *(1.2)*

sémantique: les énoncés de visibilité ne sont autorisés que dans les domaines de modulions et les domaines de mode moreta; ils contrôlent la visibilité *des chaînes de nom* qui y sont mentionnées.

propriétés statiques: un *énoncé de visibilité* a un ou deux domaines **originels** (voir 10.2) et un ou deux domaines de **destination**, définis comme suit:

- si l'*énoncé de visibilité* est un *énoncé de saisie*, son domaine de **destination** est le domaine englobant immédiatement l'*énoncé de saisie*, et ses domaines **originels** sont les domaines immédiatement englobant ce domaine;
- si l'*énoncé de visibilité* est un *énoncé d'octroi*, son domaine **originel** est le domaine englobant immédiatement l'*énoncé d'octroi*, et ses domaines de **destination** sont les domaines englobant immédiatement ce domaine;
- si l'*énoncé de visibilité* est un *énoncé d'octroi* dans une spécification de mode moreta, son domaine originel est le domaine englobant immédiatement l'*énoncé d'octroi*, et ses domaines de destination sont les domaines englobant immédiatement ce domaine.

12.2.3.3 Clause renommer préfixe

syntaxe:

<clause renommer préfixe> ::= *(1)*
(*<ancien préfixe>* –> *<nouveau préfixe>*) ! *<postfixe>* *(1.1)*

<ancien préfixe> ::= *(2)*
<préfixe> *(2.1)*
| *<vide>* *(2.2)*

<nouveau préfixe> ::= *(3)*
<préfixe> *(3.1)*
| *<vide>* *(3.2)*

<postfixe> ::= *(4)*
<postfixe de saisie> { , *<postfixe de saisie>* }* *(4.1)*
| *<postfixe d'octroi>* { , *<postfixe d'octroi>* }* *(4.2)*

syntaxe dérivée: une *clause renommer préfixe* dans laquelle le *postfixe* consiste en plus d'un *postfixe de saisie (postfixe d'octroi)* est la syntaxe dérivée de plusieurs *clauses renommer préfixe*, une pour chaque *postfixe de saisie (postfixe d'octroi)*, séparées par des virgules, avec le même *ancien préfixe* et le même *nouveau préfixe*.

Par exemple:

GRANT *(p –> q) ! a , b* ;

est la syntaxe dérivée de:

GRANT *(p –> q) ! a , (p –> q) ! b* ;

sémantique: les clauses renommer préfixe sont utilisées dans des énoncés de visibilité pour exprimer le changement de préfixe dans des chaînes de nom préfixées qui sont octroyées ou saisies. (Etant donné que les clauses renommer préfixe peuvent être utilisées sans changement de préfixe – lorsque l'*ancien* et le *nouveau préfixes* sont vides – elles sont prises pour base sémantique des énoncés de visibilité.)

propriétés statiques: une *clause renommer préfixe* a un ou deux domaines **originels**, qui sont les domaines **originels** de l'*énoncé de visibilité* dans lequel elle est écrite.

Une *clause renommer préfixe* a un ou deux domaines de **destination**, qui sont les domaines de **destination** de l'*énoncé de visibilité* dans lequel elle est écrite.

Un *postfixe* a un ensemble de *chaînes de nom* qui est l'ensemble de *chaînes de nom* attaché à son *postfixe de saisie* ou à l'ensemble de *chaînes de nom* attaché à son *postfixe d'octroi*. Ces *chaînes de nom* sont les *chaînes de nom* de postfixe de la *clause renommer préfixe*.

Une *clause renommer préfixe* a un ensemble d'**anciennes** *chaînes de nom* et un ensemble de **nouvelles** *chaînes de nom.* Chaque *chaîne de nom* de postfixe attachée à une *clause renommer préfixe* donne à la fois une **ancienne** *chaîne de nom* et une **nouvelle** *chaîne de nom* attachées à cette *clause*, comme suit: on obtient la **nouvelle** *chaîne de nom* en préfixant la *chaîne de nom* de postfixe avec le *nouveau préfixe*; on obtient l'**ancienne** *chaîne de nom* en préfixant la *chaîne* de nom de postfixe avec l'*ancien préfixe*.

Quand une **nouvelle** *chaîne de nom* et une **ancienne** *chaîne de nom* sont obtenues à partir de la même *chaîne de nom* de postfixe, l'**ancienne** *chaîne de nom* est dite être la source de la **nouvelle** *chaîne de nom*.

règles de visibilité: les **nouvelles** *chaînes de nom* attachées à une *clause renommer préfixe* sont **fortement visibles** dans leurs domaines de **destination** et sont **directement liées** dans ces domaines à leurs sources dans les domaines **originels**. Si la *clause renommer préfixe* fait partie d'un *énoncé de saisie (d'octroi)*, ces *chaînes de nom* sont saisies (octroyées) dans leurs domaines de **destination**.

Une *chaîne de nom* NS est dite être **saisissable** par le modulion M immédiatement englobé dans le domaine R si et seulement si elle est **visible** dans R et si elle n'est ni **liée** dans R à une *chaîne de nom* quelconque dans le domaine de M ni **directement liée** à *l'occurence de définition* d'une *chaîne de nom* prédéfinie.

Une *chaîne de nom* NS est dite **octroyable** par le modulion M immédiatement englobé dans le domaine R si et seulement si elle est **visible** dans le domaine de M et si elle n'est ni **liée** dans M à une *chaîne de nom* quelconque dans R ni **directement liée** dans M à *l'occurence de définition* d'une *chaîne de nom* prédéfinie.

conditions statiques: si une *clause renommer préfixe* est dans un *énoncé de saisie* immédiatement englobé dans le domaine du modulion M, alors, chacune de ses **anciennes** *chaînes de nom* doit être:

- **liée** à plusieurs *occurences de définitions* dans le domaine immédiatement englobant le domaine de M;
- **saisissable** par M.

Si une *clause renommer préfixe* est dans un *énoncé d'octroi* immédiatement englobé dans le domaine du modulion M, alors chacune de ses **anciennes** *chaînes de nom* doit être:

- **liée** à plusieurs *occurences de définitions* dans le domaine de M; et
- **octroyable** par M.

Une *clause renommer préfixe* qui intervient dans un *énoncé d'octroi (de saisie)* doit avoir un *postfixe* qui est un *postfixe d'octroi (de saisie).*

exemple:

25.35 (stack ! int –> stack) ! **ALL** *(1.1)*

12.2.3.4 Enoncé d'octroi

syntaxe:

<énoncé d'octroi> ::= *(1)*
GRANT *<clause renommer préfixe>* { , *<clause renommer préfixe>* }* ; *(1.1)*
| **GRANT** *<fenêtre d'octroi>* [*<clause préfixe>*] [*<clause d'ami>*]; *(1.2)*

<fenêtre d'octroi> ::= *(2)*
<postfixe d'octroi> { , *<postfixe d'octroi>* }* *(2.1)*

<postfixe d'octroi> ::= *(3)*
<chaîne de nom> [(*<liste de paramètres>* [[**RETURNS**] (*<spec de résultat>*)]] *(3.1)*
| *<chaîne de nom de néomode> <clause d'interdiction>* *(3.2)*
| [*<préfixe>* !] **ALL** *(3.3)*

<clause préfixe> ::= *(4)*
PREFIXED [*<préfixe>*] *(4.1)*

<clause d'interdiction> ::= *(5)*
FORBID { *<liste de noms d'interdiction>* | **ALL** } *(5.1)*

<liste de noms d'interdiction> ::= *(6)*
(*<nom de champ>* { , *<nom de champ>* }*) *(6.1)*

<clause ami> ::= *(7)*
TO *<liste de noms d'amis>* *(7.1)*

<liste de noms d'amis> ::= *(8)*
<nom d'ami> { , *<nom d'ami>*}* *(8.1)*

<nom d'ami> ::= *(9)*
<nom de mode modulion ou moreta> [! *<nom de procédure ou de processus ami>*] *(9.1)*

<nom de mode modulion ou moreta> ::= *(10)*
<nom de modulion> *(10.1)*
| *<nom de mode moreta>* *(10.2)*

<nom de procédure ou de processus ami> ::= *(11)*
<nom de procédure> [(*<liste de paramètres>* [[**RETURNS**] (*<spec de résultat>*)]] *(11.1)*
| *<nom de processus>* *(11.2)*

sémantique: les énoncés d'octroi sont un moyen d'étendre aux domaines immédiatement englobants la visibilité des chaînes de nom d'un domaine de modulion. **FORBID** ne peut être spécifié que pour des noms de **néomode** qui sont des modes structure. Cela signifie que tous les locus et toutes les valeurs de ce mode ont des champs qui ne peuvent être choisis qu'à l'intérieur du modulion d'octroi, et non à l'extérieur.

Les règles de visibilité suivantes sont applicables:

- si l'*énoncé d'octroi* contient une ou des *clauses renommer préfixe*, l'*énoncé d'octroi* a l'effet de sa ou ses *clauses renommer préfixe* (voir 12.2.3.3);
- si l'*énoncé d'octroi* contient des *fenêtres d'octroi*, c'est la notation abrégée pour un ensemble d'*énoncés d'octroi* avec *clause renommer préfixe* formée comme suit:
  - à chaque *postfixe d'octroi* dans la *fenêtre d'octroi* correspond un *énoncé d'octroi;*
  - l'*ancien préfixe* dans leur *clause renommer préfixe* est vide;
  - le *nouveau préfixe* dans leur *clause renommer préfixe* est le *préfixe* attaché à la *clause préfixe* dans l'*énoncé d'octroi*, ou il est vide s'il n'existe pas de *clause préfixe* dans l'*énoncé d'octroi* originel;
  - le *postfixe* dans la *clause renommer préfixe* est le *postfixe* correspondant dans la *fenêtre d'octroi*.
- La notation **FORBID ALL** est une notation abrégée interdisant tous les *noms de champ* du nom de **néomode** (voir 12.2.5).
- Si une *clause renommer préfixe* dans un *énoncé d'octroi* a un *postfixe d'octroi* qui contient un *préfixe* et **ALL**, alors elle est de la forme:

  (OP–>NP) ! P ! **ALL**

  où *OP* et *NP* sont, respectivement, l'*ancien préfixe* et le *nouveau préfixe* éventuellement vides et *P* le *préfixe* dans le *postfixe d'octroi*. La *clause renommer préfixe* est alors une notation abrégée pour une clause de la forme:

  (OP ! P–> NP ! P) ! **ALL**

- Si une clause ami contenant la visibilité d'objets GRANTed ne porte que sur les groupes qui sont mentionnés dans la *liste de noms d'amis*.

propriétés statiques: une *clause préfixe* a un *préfixe* qui lui est attaché, défini comme suit:

- si la *clause préfixe* contient un *préfixe*, alors, ce *préfixe* lui est attaché;
- sinon, le *préfixe* qui lui est attaché est un *préfixe simple* dont la *chaîne de nom* est déterminée comme suit:
  - si le domaine immédiatement englobant le *préfixe* est un *module* ou une *région*, alors, la *chaîne de nom* est la même que celle du nom de **module** ou du nom de **région** de ce modulion;
  - si le domaine immédiatement englobant le *préfixe* est une *région de spec* ou un *module de spec*, alors, la *chaîne* de nom est la *chaîne* de nom qui précède **SPEC**.

Un *postfixe d'octroi* a un ensemble de *chaînes de nom* qui lui est attaché, défini comme suit:

- si c'est une *chaîne de nom*, ou s'il contient une *chaîne de nom de néomode*, alors, l'ensemble ne contient que cette *chaîne de nom.*
- sinon, soit *OP* l'*ancien préfixe* (éventuellement vide) de la *clause renommer préfixe* dans laquelle est placé le *postfixe d'octroi*, l'ensemble contient toutes les chaînes *de nom* de la forme *OP ! N* (c'est-à-dire obtenues en préfixant *N* par *OP*) pour toute *chaîne de nom N* telle que *OP ! N* est **visible** dans le domaine du modulion dans lequel le *postfixe d'octroi* est placé et **octroyable** par ce modulion.

conditions statiques: la *chaîne de nom* de *néomode* avec *clause d'interdiction* doit être **visible** dans le domaine *R* du modulion dans lequel se trouve l'*énoncé d'octroi*. La *chaîne de nom* de *néomode* doit être **liée** dans *R* à *l'occurence de définition* d'un néomode, qui doit être un mode structure, et chaque nom de champ dans la *liste de noms d'interdiction* doit être un *nom de champ* de ce mode. L'*occurence de définition* de néomode doit être immédiatement englobée par *R*. Tous les *noms de champ* dans une *liste de noms d'interdiction* doivent avoir des chaînes de nom différentes.

Si l'*énoncé d'octroi* se trouve dans le domaine d'une *région* ou d'une *région de spec*, il ne doit pas octroyer une *chaîne de nom* qui est **liée** dans le domaine de *l'occurence de définition* d':

- un nom **de locus**; ou
- un nom d'**identité de locus**, où le *locus* dans sa déclaration est **intrarégional**; ou
- un nom de **synonyme** dont la *valeur* est **intrarégionale**.

La *clause renommer préfixe* dans un *énoncé d'octroi* doit avoir un *postfixe d'octroi*.

Si un *énoncé d'octroi* contient une *clause de préfixe* qui ne contient pas de *préfixe*, alors son modulion immédiatement englobant ne doit pas être un *contexte:*

- si son modulion immédiatement englobant est un *module* ou une *région*, il doit être nommé (c'est-à-dire qu'il doit être précédé par une *occurence de définition* suivie par un deux points);
- si son modulion immédiatement englobant est un *module de spec* ou une *région de spec*, il doit être précédé par une *chaîne de nom simple*.

Un nom N contenu dans une *liste de noms* d'amis d'un *énoncé d'octroi*, qui est placé immédiatement dans le domaine du groupe G, doit être défini immédiatement dans le domaine du groupe entourant directement G.

Si l'énoncé d'octroi survient immédiatement dans une spécification moreta, il n'y a pas lieu de préfixer.

exemples:

25.7 **GRANT** *(–> stack ! char) !* **ALL**; *(1.1)*

6.44 *gregorian_date, julian_day_number* *(2.1)*

12.2.3.5 Enoncé de saisie

syntaxe:

<énoncé de saisie> ::= *(1)*
SEIZE *<clause renommer préfixe>* { , *<clause renommer préfixe>* }* ; *(1.1)*
| **SEIZE** *<fenêtre de saisie>* [*<clause de préfixe>*] ; *(1.2)*

<fenêtre de saisie> ::= *(2)*
<postfixe de saisie> { , *<postfixe de saisie>* }* *(2.1)*

<postfixe de saisie> ::= *(3)*
<chaîne de nom>[(*<liste de paramètres formels>*)
[[**RETURNS**] (*<spec de résultat>*)]] *(3.1)*
| [*<préfixe>* !] **ALL** *(3.2)*

sémantique: les énoncés de saisie sont un moyen d'étendre la visibilité des chaînes de nom dans des domaines de groupes vers les domaines de modulions immédiatement englobés.

Les règles de visibilité suivantes s'appliquent:

- si l'*énoncé de saisie* contient une ou des *clauses renommer préfixe*, il a l'effet de sa ou ses *clauses renommer préfixe* (voir 12.2.3.3);

- si l'*énoncé de saisie* contient une *fenêtre de saisie*, c'est la notation abrégée pour un ensemble d'*énoncés de saisie* ayant des *clauses renommer préfixe* formées comme suit:
  - pour chaque *postfixe de saisie* dans la *fenêtre de saisie*, il existe un *énoncé de saisie* correspondant;
  - l'*ancien préfixe* de leur *clause renommer préfixe* est le préfixe attaché à la *clause de préfixe* dans l'*énoncé de saisie*, ou il est vide s'il n'existe pas de *clause préfixe* dans l'*énoncé de saisie* originel;
  - le *nouveau préfixe* de leur *clause renommer préfixe* est vide;
  - le *postfixe* de leur *clause renommer préfixe* est le *postfixe* correspondant de la *fenêtre de saisie*;
- si une *clause renommer préfixe* dans un *énoncé de saisie* a un *postfixe de saisie* qui contient un *préfixe* et **ALL**, alors il a la forme

  (OP–>NP) ! P ! **ALL**

  où *OP* et *NP* sont respectivement l'*ancien préfixe* et le *nouveau préfixe* éventuellement vides et *P* le *préfixe* dans le *postfixe de saisie*. La *clause renommer préfixe* est alors la notation abrégée pour une clause de la forme

  (OP ! P–>NP ! P) ! **ALL**

propriétés statiques: un *postfixe de saisie* a un ensemble de *chaînes de nom*, défini comme suit:

- si le *postfixe de saisie* est une *chaîne de nom*, l'ensemble ne contient que la *chaîne de nom;*
- sinon, si le *postfixe de saisie* est **ALL**, soit *OP* l'*ancien préfixe* (éventuellement vide) de la *clause renommer préfixe* dont fait partie le *postfixe de saisie*, alors, l'ensemble contient toutes les *chaînes de nom* de la forme *OP ! S*, pour toutes les *chaînes de nom S*, telles que:
  - OP!S est **visible** dans le domaine englobant immédiatement le modulion dans lequel se trouve l'*énoncé de saisie*;
  - **qu'il peut être saisi** par ce modulion;
  - qu'il est **lié** à une **quasi**-*occurence de définition* si ce modulion est précédé d'un contexte.

conditions statiques: la *clause renommer préfixe* dans un *énoncé de saisie* doit avoir un *postfixe de saisie.*

Si un *énoncé de saisie* contient une *clause de préfixe* qui ne contient pas de *préfixe*, alors, son modulion immédiatement englobant ne doit pas être un *contexte*:

- si son modulion immédiatement englobant est un *module* ou une *région*, alors, il doit être nommé (c'est-à-dire qu'il doit être précédé par une *occurence de définition* suivie d'un deux points);
- si son modulion immédiatement englobant est un *module de spec* ou une *région de spec*, alors, il doit être précédé par une *chaîne de nom simple*.

exemple:

25.35 **SEIZE** (*stack ! int –> stack*) *!* **ALL**; *(1.1)*

12.2.4 Visibilité des noms d'éléments d'ensemble

Un nom d'*élément d'ensemble* ne peut survenir que dans le contexte d'un *littéral ensemble*.

Si un *nom de mode ensemble* est spécifié dans le *littéral ensemble*, la *chaîne de nom* d'un *nom d'élément d'ensemble* peut être **liée** à une *occurence de définition de nom d'élément d'ensemble* dans le mode de la classe du *littéral ensemble*.

Sinon, un *nom de mode ensemble* n'est pas spécifié et la *chaîne de nom* ne peut être **liée** à une *occurence de définition de nom d'élément d'ensemble* que si elle n'est pas **visible** dans le domaine dans lequel le *littéral ensemble* est placé.

12.2.5 Visibilité de noms de champ

Des *noms de champ* ne peuvent apparaître que dans les contextes suivants:

- un *champ de structure* ou un *champ de valeur structure*;
- les *multiplets de structure avec noms de champ*;
- les *clauses d'interdiction* d'un *énoncé d'octroi*.

On notera qu'un *nom de champ* ne peut pas survenir dans un *postfixe d'octroi* ou dans un *postfixe de saisie*.

Dans ces cas, la *chaîne de nom* du *nom* de *champ* peut être **liée** à une *occurence de définition de nom de champ* dans le mode M ou dans le mode **définissant** de M, obtenue comme suit:

- M est le mode du *locus structure* ou de la *valeur primitive structure* (**forte**);
- M est le mode du *multiplet de structure*;
- M est le mode de l'*occurence de définition* à laquelle la *chaîne de nom de néomode* est **liée** dans le domaine dans lequel se trouve la *clause d'interdiction.*

Cependant, si la **nouveauté** de M est une *occurence de définition* qui définit un nom de **néomode** qui a été octroyé par un *énoncé d'octroi* dans un modulion comme un *postfixe d'octroi* avec *clause d'interdiction*, alors les noms de champ mentionnés dans la liste de noms d'interdiction sont seulement **visibles**:

- dans le groupe du modulion octroyant;
- si la **nouveauté** de M est **liée par la nouveauté** à une **quasi-nouveauté** N, alors dans le groupe du domaine dans lequel N est immédiatement englobé;
- si le modulion est une *spec de module* ou de *région*, alors dans le domaine du modulion **correspondant**.

Hors de ces domaines, les *noms de champ* mentionnés dans la *liste de noms d'interdiction* sont **invisibles** et ne peuvent pas être utilisés.

12.2.6 Dépendance des locus

Une instance LI d'un locus L directement déclaré dépend de l'exécution du groupe l'entourant directement qui a créé LI.

exemple:

```
SYNMODE TM = TASK SPEC ....
SYNMODE MM = MODULE SPEC
        DCL T1 TM;
END MM;
DCL M1 MM;
```

L'instance en vigueur M1-I de M1 contient une instance M1-I.T1 de MM.T1. M1-I.T1 a été crée au cours de l'exécution de "DCL M1 MM;". Pour cette raison, M1-I.T1 dépend de M1-I.

Dépendance des locus tas: *GETSTACK* et *ALLOCATE* créent un nouveau locus L et donnent une valeur de référence R pour L. Il y a deux cas:

a) on sait que le mode de R est RM. Dans ce cas, L dépend du créateur de l'instance appropriée de RM;

b) le mode de R est inconnu (IF ALLOCATE(...) = ALLOCATE(...) .....). Dans ce cas, L dépend du créateur de l'instance appropriée de LM, ce dernier étant le mode de L.

Un locus Lc qui est une sous-composante d'un locus L dépend de L.

12.3 Sélection de cas

syntaxe:

<spécification d'étiquettes de cas> ::= *(1)*
<liste d'étiquettes de cas> { , *<liste d'étiquettes de cas>* }* *(1.1)*

<liste d'étiquettes de cas> ::= *(2)*
(*<étiquette de cas>* { , *<étiquette de cas>* }*) *(2.1)*
| *<indifférent>* *(2.2)*

<étiquette de cas> ::= *(3)*
<expression littérale discrète> *(3.1)*
| *<intervalle littéral>* *(3.2)*
| *<nom de mode discret>* *(3.3)*
| **ELSE** *(3.4)*

<indifférent> ::= *(4)*
(*) *(4.1)*

sémantique: la sélection de cas est un moyen de sélectionner une alternative d'une liste d'alternatives. La sélection se base sur la spécification d'une liste de valeurs de sélecteurs. La sélection de cas s'applique à:

- des champs alternatifs (voir 3.13.4), auquel cas une liste de champs récurrents est sélectionnée;
- des multiplets matriciels avec indices (voir 5.2.5), auquel cas une valeur élément de matrice est sélectionnée;

- des expressions conditionnelles (voir 5.3.2), auquel cas une expression est sélectionnée;
- des actions à cas (voir 6.4), auquel cas une liste d'énoncés d'action est sélectionnée.

Dans les première, troisième et quatrième situations, chaque alternative est étiquetée avec une spécification d'étiquettes de cas; pour le multiplet matriciel avec indice, chaque valeur est étiquetée avec une liste d'étiquettes de cas. Pour faciliter l'explication, la liste d'étiquettes de cas pour le multiplet matriciel avec indice sera considérée dans ce paragraphe comme une spécification d'étiquettes de cas réduite à une seule liste d'étiquettes de cas.

La sélection de cas sélectionne l'alternative qui est étiquetée par la spécification d'étiquettes de cas qui correspond à la liste de valeurs des sélecteurs. (Le nombre de valeurs de sélecteurs sera toujours le même que le nombre de listes d'étiquettes de cas dans la spécification d'étiquettes de cas.) Une liste de valeurs est dite correspondre à une spécification d'étiquettes de cas si et seulement si chaque valeur correspond à la liste d'étiquettes de cas correspondante (par position) dans la spécification d'étiquettes de cas.

Une valeur est dite correspondre à une liste d'étiquettes de cas si et seulement si:

- la liste d'étiquettes de cas consiste en des étiquettes de cas et la valeur est une des valeurs indiquées explicitement par l'une des étiquettes de cas, ou indiquées implicitement dans le cas de **ELSE**;
- la liste d'étiquettes de cas consiste en *indifférent*.

Les valeurs indiquées explicitement par une étiquette de cas sont les valeurs de toute *expression littérale discrète*, ou définies par *l'intervalle littéral* ou le *nom de mode discret*. Les valeurs indiquées implicitement par **ELSE** sont toutes les valeurs possibles des sélecteurs de cas qui ne sont indiquées explicitement par aucune liste d'étiquettes de cas associée (c'est-à-dire appartenant à la même valeur de sélecteur) dans toute spécification d'étiquettes de cas.

propriétés statiques:

- *aux champs alternatifs* avec *spécification d'étiquettes de cas*, un *multiplet matriciel avec indice*, une *expression conditionnelle*, ou une *action à cas* on attache une liste de spécifications d'étiquettes de cas, formée en prenant, respectivement, la *spécification d'étiquettes de cas* précédant chaque *alternative variant*, *valeur*, ou *cas alternatif*;
- à une *étiquette de cas* on attache une classe qui est, s'il s'agit d'une *expression littérale discrète*, la classe de *l'expression littérale discrète*; s'il s'agit d'un *intervalle littéral*, la **classe résultante** des classes de chaque *expression littérale discrète* dans *l'intervalle littéral*; s'il s'agit d'un *nom de mode discret*, la **classe résultante** de la M-classe par valeur où M est le *nom de mode discret*; si c'est **ELSE**, la classe **toute**;
- à une *liste d'étiquettes de cas* on attache une classe qui est, s'il s'agit d'*indifférent*, la classe **toute**, sinon la **classe résultante** des classes de chaque *étiquette de cas*;
- à une *spécification d'étiquettes de cas*, on attache une liste de classes qui sont les classes de chaque liste d'étiquettes de cas;
- à une liste de spécifications d'étiquettes de cas, on attache une **liste résultante des classes**. Cette **liste résultante des classes** est formée en constituant, pour chaque position de la liste, la **classe résultante** de toutes les classes qui ont cette position.

Une liste de spécifications d'étiquettes de cas est **complète** si et seulement si pour toutes les listes de valeurs possibles des sélecteurs, une spécification d'étiquettes de cas existe, qui correspond à la liste de valeurs des sélecteurs. L'ensemble de toutes les valeurs possibles d'un sélecteur est déterminé par le contexte, de la manière suivante:

- pour un mode structure **variable avec étiquette**, c'est l'ensemble des valeurs défini par le mode du champ **étiquette** correspondant;
- pour un mode structure **variable sans étiquette**, c'est l'ensemble des valeurs défini par le mode **racine** de la **classe résultante** correspondante (qui n'est jamais la classe **toute**, voir 3.13.4);
- pour un multiplet matriciel, c'est l'ensemble des valeurs défini par le mode **indice** du mode du multiplet matriciel;
- pour une action de cas avec liste d'intervalles, c'est l'ensemble des valeurs défini par le mode discret correspondant dans la liste d'intervalles;
- pour une action de cas sans liste d'intervalles, ou une expression conditionnelle, c'est l'ensemble des valeurs défini par M, où la classe du sélecteur correspondant est la M-classe par valeur ou la M-classe par dérivation.

conditions statiques: pour chaque *spécification d'étiquettes de cas*, le nombre d'occurrences de *liste d'étiquettes de cas* doit être égal.

Pour tout couple *d'occurences de spécification d'étiquettes de cas*, leurs listes de classes doivent être **compatibles**.

La liste d'occurrences de *spécification d'étiquettes de cas* doit être **cohérente**, c'est-à-dire que chaque liste de valeurs des sélecteurs possible ne correspond qu'à une spécification d'étiquettes de cas.

Si le mode **racine** de la classe d'une *liste de d'étiquettes de cas* est un mode entier, il faut qu'il y ait un mode entier **prédéfini** contenant toutes les valeurs données par chaque *étiquette de cas*.

exemples:

11.9	(*occupied*)	*(2.1)*
11.58	(*rook*), (*)	*(1.1)*
8.26	(**ELSE**)	*(2.1)*

12.4 Définition et résumé des catégories sémantiques

Le présent paragraphe donne un résumé de toutes les catégories sémantiques qui sont indiquées dans la description syntaxique au moyen d'une partie soulignée. Si ces catégories ne sont pas définies dans le paragraphe approprié, la définition est donnée ici, sinon le paragraphe approprié est référencé.

12.4.1 Noms

Noms de mode

nom de mode:	voir 3.2.1.
nom de mode accès:	un *nom* définissant un mode accès.
nom de mode association:	un *nom* définissant un mode association.
nom de mode booléen:	un *nom* définissant un mode booléen.
nom de mode caractère:	un *nom* définissant un mode caractère.
nom de mode chaîne paramétré:	un *nom* définissant un mode chaîne **paramétré**.
nom de mode chaîne:	un *nom* définissant un mode.
nom de mode descripteur:	un *nom définissant* un mode descripteur.
nom de mode discret:	un *nom* définissant un mode discret.
nom de mode temps absolu:	un *nom* définissant un mode temps absolu.
nom de mode ensemble:	un *nom* définissant un mode ensemble.
nom de mode ensembliste:	un *nom* définissant par un mode ensembliste.
nom de mode entier:	un *nom* définissant un mode entier.
nom de mode événement:	un *nom* définissant un mode événement.
nom de mode instance:	un *nom* définissant un mode instance.
nom de mode interface:	un *nom* définissant un mode interface.
nom de mode intervalle:	un *nom* définissant un mode intervalle.
nom de mode intervalle à virgule flottante:	un *nom* définissant un mode intervalle à virgule flottante.
nom de mode intervalle discret:	un *nom* définissant un mode intervalle discret.
nom de mode module:	un *nom* définissant un mode module.
nom de mode modulion ou de mode moreta:	un *nom* définissant un mode modulion ou un mode moreta.
nom de mode moreta:	un *nom* définissant un mode moreta.
nom de mode moreta générique:	un *nom* définissant un mode moreta générique.
nom de mode procédure:	un *nom* définissant un mode procédure.
nom de mode matrice:	un *nom* définissant un mode matrice.
nom de mode matrice paramétré:	un *nom* définissant par un mode matrice **paramétré**.
nom de mode région:	un *nom* définissant un mode région.
nom de mode référence libre:	un *nom* définissant un mode référence libre.

nom de mode référence liée:	un *nom* définissant un mode référence liée.
nom de mode structure:	un *nom* définissant un mode structure.
nom de mode structure paramétré:	un *nom* définissant un mode structure **paramétré**.
nom de mode structure variable:	un *nom* définissant un mode structure **variable**.
nom de mode tâche:	un *nom* définissant un mode tâche.
nom de mode tampon:	un *nom* définissant un mode tampon.
nom de mode texte:	un *nom* définissant un mode texte.
nom de mode virgule flottante:	un *nom* définissant un mode virgule flottante.

Noms d'accès

nom d'identité de locus:	voir 4.1.3.
nom de locus:	voir 4.1.2.
nom de locus faire-avec:	voir 6.5.4.
nom d'énumération de locus:	voir 6.5.2.

Noms de valeur

nom de littéral booléen:	voir 5.2.4.4.
nom de littéral vide:	voir 5.2.4.7.
nom de synonyme:	voir 5.1.
nom de valeur faire-avec:	voir 6.5.4.
nom de valeur reçue:	voir 6.19.2, 6.19.3.
nom d'énumération de valeur:	voir 6.5.2.

Noms divers

nom d'étiquette:	voir 6.1 et 10.6.
nom de champ étiquette:	voir 3.13.4.
nom de module générique:	voir 10.11.
nom de modulion:	voir 12.2.3.4.
nom de procédure générale:	un *nom de procédure* dont la généralité est un nom de procédure **générale**.
nom de procédure ou de processus ami:	voir 12.2.3.4.
nom de procédure:	voir 10.4.
nom de procédure générique:	voir 10.11.
nom de processus générique:	voir 10.11.
nom de processus:	voir 10.5.
nom de région générique:	voir 10.11.
nom de signal:	voir 11.5.
nom de synonyme indéfini:	voir 5.1.
nom d'élément d'ensemble:	voir 3.4.5.
nom de routine prédéfinie:	un nom, défini par le CHILL ou par l'implémentation, dénotant une routine prédéfinie.
nom non réservé:	un *nom* qui n'est aucun des noms réservés mentionnés à l'Appendice III.
chaîne de nom de *néomode*:	une *chaîne* de nom **liée** à la définition d'un nom de **néomode**.

12.4.2 Locus

locus accès:	un *locus* qui a un mode accès.
locus association:	un *locus* qui a un mode association.
locus chaîne de caractères:	un *locus* qui a un mode chaîne de **caractères**.
locus chaîne:	un *locus* qui a un mode chaîne.
locus de mode statique:	un *locus* qui a un mode statique.
locus discret:	un *locus* qui a un mode discret.
locus entier:	un *locus* qui a un mode entier.
locus événement:	un *locus* qui a un mode événement.
locus instance:	un *locus* qui a un mode instance.
locus moreta:	un *locus* qui a un mode moreta.
locus matrice:	un *locus* qui a un mode matrice.
locus structure:	un *locus* qui a un mode structure.
locus tampon:	un *locus* qui a un mode tampon.
locus texte:	un *locus* qui a un mode texte.
locus virgule flottante:	un *locus* qui a un mode virgule flottante.

12.4.3 Expressions et valeurs

expression booléenne:	une *expression* dont la classe est **compatible** avec un mode booléen.
expression chaîne de caractères:	une *expression* dont la classe est **compatible** avec un mode chaîne de **caractères**.
expression chaîne:	une *expression* dont la classe est **compatible** avec un mode chaîne.
expression discrète:	une *expression* dont la classe est **compatible** avec un mode discret.
expression ensembliste:	une *expression* dont la classe est **compatible** avec un mode ensembliste.
expression entière:	une *expression* dont la classe est **compatible** avec un mode entier.
expression littérale à virgule flottante:	une *expression à virgule flottante* qui est un **littéral**.
expression littérale discrète:	une *expression discrète* qui est **littérale**.
expression de littéral entier:	une *expression entière* qui est **littérale**.
expression littérale:	une *expression* qui est un **littéral**.
expression matrice:	une *expression* dont la classe est **compatible** avec un mode matrice.
expression virgule flottante:	une *expression* dont la classe est **compatible** avec un mode virgule flottante.
valeur constante:	une *valeur* qui est **constante**.
valeur primitive chaîne:	une *valeur primitive* dont la classe est **compatible** avec un mode chaîne.
valeur primitive de locus moreta à référence liée:	voir 6.7.
valeur primitive durée:	une *valeur primitive* dont la classe est **compatible** avec un mode durée.
valeur primitive instance:	une *valeur primitive* dont la classe est **compatible** avec un mode instance.

valeur primitive procédure: une *valeur primitive* dont la classe est **compatible** avec un mode procédure.

valeur primitive matrice: une *valeur primitive* dont la classe est **compatible** avec un mode matrice.

valeur primitive rangée: une *valeur primitive* dont la classe est **compatible** avec un mode rangée.

valeur primitive référence libre: une *valeur primitive* dont la classe est **compatible** avec un mode référence libre.

valeur primitive référence liée: une *valeur primitive* dont la classe est **compatible** avec un mode référence liée.

valeur primitive référence: une *valeur primitive* dont la classe est **compatible** avec un mode référence liée, un mode référence libre ou un mode descripteur.

valeur primitive structure: une *valeur primitive* dont la classe est **compatible** avec un mode structure.

valeur primitive temps absolu: une *valeur primitive* dont la classe est **compatible** avec un mode temps absolu.

12.4.4 Catégories sémantiques diverses

appel de procédure de composante moreta: voir 2.7.

appel de procédure rendant locus: voir 6.7.

appel de procédure rendant valeur: voir 6.7.

appel de routine prédéfinie rendant locus: voir 6.7.

appel de routine prédéfinie rendant valeur: voir 6.7.

caractère large non réservé: un caractère large qui n'est ni un guillemet (") ni un accent circonflexe (^).

caractère non réservé: un caractère qui n'est ni un guillemet (") ni un accent circonflexe (^).

caractère non spécial: un caractère qui n'est ni un accent circonflexe (^), ni une parenthèse gauche (*(*).

caractère non-pour cent: un caractère qui n'est pas un pourcentage (%).

énoncé de déclaration moreta: voir 3.15.

énoncé de définition de procédure protégée simple: voir 10.4.

énoncé de définition néomode moreta: voir 3.15.

énoncé de définition synmode moreta:

énoncé de définition de procédure protégée simple: voir 10.4.

énoncé de signature de procédure protégée simple: voir 10.4.

liste de paramètres effectifs de constructeur: voir 4.1.2.

mode chaîne: un mode dans lequel le mode composite est un mode chaîne.

mode discret: un mode dans lequel le mode non composite est un mode discret.

mode matrice: un mode dans lequel le mode composite est un mode matrice.

mode structure variable: un mode dans lequel le mode non composite est un mode structure **variable**.

13 Options pour l'implémentation

13.1 Routines opérations prédéfinies par l'implémentation

sémantique: une implémentation peut fournir un ensemble de routines prédéfinies par l'implémentation en plus de l'ensemble des routines prédéfinies par le langage.

Le mécanisme de passage de paramètres est défini par l'implémentation.

noms prédéfinis: le nom d'une routine prédéfinie par l'implémentation est prédéfini comme un nom **de routine prédéfinie**.

propriétés statiques: à un nom de **routine prédéfinie** peut être attaché un ensemble de noms d'exception définis par l'implémentation. Un *appel de routine prédéfinie* est un *appel de routine prédéfinie* **rendant valeur (rendant locus)** si et seulement si l'implémentation spécifie que pour un choix de propriétés statiques des paramètres et pour le contexte statique de l'appel, l'appel de routine prédéfinie rend une valeur (un locus).

L'implémentation spécifie aussi la **régionalité** de la valeur (du locus).

13.2 Modes entier définis par l'implémentation

Une implémentation définit la **limite supérieure** et la **limite inférieure** du mode entier *INT*. Une implémentation peut définir d'autres modes entier que ceux définis par *INT*, par exemple des entiers courts, entiers longs, entiers sans signe. Ces modes entier doivent être dénotés par des noms de **mode** entier définis par l'implémentation. Ces noms sont considérés comme des noms de **néomode**, **similaires** à *INT*. Leurs intervalles de valeurs sont définis par l'implémentation. Ces modes entier peuvent être définis comme modes **racine** de classes appropriées.

13.3 Modes virgule flottante définis par l'implémentation

Une implémentation définit la **borne supérieure** et la **borne inférieure,** la **limite supérieure négative** et la **limite inférieure positive**, et la **précision** du mode virgule flottante *FLOAT*. Une implémentation peut définir d'autres modes virgule flottante que ceux définis par *FLOAT*, par exemple un modes court ou un mode long. Ces modes virgule flottante entier doivent être dénotés par des noms de **mode** virgule flottante définis par l'implémentation. Ces noms sont considérés comme des noms de **néomode**, **similaires** à *FLOAT*. Leurs intervalles de valeurs, leurs limites inférieures et leur **précision** sont définis par l'implémentation. Ces modes virgule flottante peuvent être définis comme modes **racine** de classes appropriées.

13.4 Noms de processus définis par l'implémentation

Une implémentation peut définir un ensemble de noms de **processus** définis par l'implémentation, c'est-à-dire des noms de **processus** dont la définition n'est pas spécifiée en CHILL. La définition est considérée comme étant placée dans le domaine du processus imaginaire le plus externe ou dans un contexte quelconque. Les processus de ce nom peuvent être démarrés et des valeurs instance les dénotant peuvent être manipulées.

13.5 Filets définis par l'implémentation

Une implémentation peut spécifier qu'un filet défini par l'implémentation termine la définition de processus; un tel filet peut s'appliquer à toute exception.

13.6 Noms d'exception définis par l'implémentation

Une implémentation peut définir un ensemble de noms d'exception.

13.7 Autres caractéristiques définies par l'implémentation

- vérification statique des conditions dynamiques (voir 2.1.2)
- *directive d'implémentation* (voir 2.6)
- cas des chaînes de nom simple **spéciales**

- *nom de référence de texte* (voir 2.7 et 10.10.1)
- **généralité** par défaut (voir 10.4)
- ensemble de valeurs de modes durée (voir 3.12.2)
- ensemble de valeurs de modes temps absolu (voir 3.12.3)
- **implantation d'élément** par défaut (voir 3.13.3)
- comparaison de valeurs de structures **variables sans étiquettes** (voir 3.13.4)
- nombre de bits dans un mot (voir 3.13.5)
- occupation minimale de bits (voir 3.13.5)
- autres (sous-)locus **référençables** (voir 4.2.1)
- sémantique d'un nom de *locus faire-avec* et d'un nom de *valeur faire-avec* qui est un champ **récurrent** d'un locus à structure **variable sans étiquettes** (voir 4.2.2 et 5.2.3)
- sémantique de champs **récurrents** à structure **variable sans étiquettes** (voir 4.2.10, 5.2.14 et 6.2)
- sémantique de la *conversion de locus* (voir 4.2.13)
- sémantique de la *conversion d'expression* et autres conditions (voir 5.2.11)
- autres *paramètres réels* dans une *expression démarrer* (voir 5.2.15)
- intervalles de valeurs pour des expressions **littérales** et **constantes** (voir 5.3.1)
- algorithme de séquencement (voir 6.15, 6.18.2, 6.18.3, 6.19.2 , 6.19.3 et 11.2.1)
- libération de la mémoire dans *TERMINATE* (voir 6.20.4)
- dénotation des fichiers (voir 7.1)
- opérations sur les associations (voir 7.1 et 7.2.1)
- associations non exclusives (voir 7.1)
- autres attributs des valeurs d'association (voir 7.2.2)
- sémantique de *paramètres pour associer* (voir 7.4.2)
- exception *ASSOCIATEFAIL* (voir 7.4.2)
- sémantique des *paramètres pour modifier* (voir 7.4.5)
- exceptions *CREATEFAIL*, *DELETEFAIL* et *MODIFYFAIL* (voir 7.4.5)
- exception *CONNECTFAIL* (voir 7.4.6)
- sémantique de la lecture d'enregistrements qui ne sont pas des valeurs autorisées par le mode enregistrement (voir 7.4.9)
- autres actions **temporisables** (voir 9.2)
- exception *TIMERFAIL* (voir 9.3.1, 9.3.2 et 9.3.3)
- précision des valeurs de durée (voir 9.4.1 et 9.4.2)
- indication des valeurs **constantes** dans des *quasi-définitions de synonyme* (voir 10.10.3)
- **régionalité** des routines prédéfinies (voir 11.2.2).

Appendice I

Jeu de caractères pour le langage CHILL

Le jeu de caractères du langage *CHILL* est une extension de l'Alphabet nº 5 du CCITT, version de référence internationale, Recommandation V.3. Aucune représentation graphique n'est définie pour les valeurs dont les représentations sont supérieures à 127.

La représentation entière est le nombre binaire formé des bits b_8 à b_1, où b_1 est le bit de plus faible poids.

	$b_7b_6b_5$	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

Appendice II

Symboles spéciaux

	Nom	Utilisation
;	point-virgule	terminateur d'énoncé etc.
,	virgule	séparateur dans différentes constructions
(	parenthèse gauche	parenthèse ouvrante dans différentes constructions
)	parenthèse droite	parenthèse fermante dans différentes constructions
[	crochet carré gauche	crochet ouvrant d'un multiplet
]	crochet carré droit	crochet fermant d'un multiplet
(:	crochet de multiplet gauche	crochet ouvrant d'un multiplet
:)	crochet de multiplet droit	crochet fermant d'un multiplet
:	deux points	indicateur d'étiquette, d'intervalle
.	point	symbole de sélection de champ
:=	symbole d'affectation	affectation, initialisation
<	inférieur à	opérateur relationnel
<=	inférieur ou égal à	opérateur relationnel
=	égal à	opérateur relationnel, affectation, initialisation, indicateur de définition
/=	différent de	opérateur relationnel
>=	supérieur ou égal à	opérateur relationnel
>	supérieur à	opérateur relationnel
+	plus	opérateur d'addition
–	minus	opérateur de soustraction
*	astérisque	opérateur de multiplication, valeur indéfinie, valeur anonyme, symbole indifférent
/	solidus	opérateur de division
//	double solidus	opérateur de concaténation
–>	flèche	référenciation ou déréférenciation, renommage de préfixe
<>	diamant	début ou fin d'une clause de directive
/*	ouverture de commentaire	crochet de début de commentaire
*/	fin de commentaire	crochet de fin de commentaire
'	apostrophe	symbole de début ou de fin de divers littéraux
#	dièse	conversion de locus et d'expressions
"	citation	symbole de début ou de fin dans les littéraux de chaîne de caractères
!	opérateur de préfixation	préfixation de noms
B'	qualification de littéral	base binaire pour littéral
b'	qualification de littéral	base binaire pour littéral
D'	qualification de littéral	base décimale pour littéral
d'	qualification de littéral	base décimale pour littéral
H'	qualification de littéral	base hexadécimale pour littéral
h'	qualification de littéral	base hexadécimale pour littéral
O'	qualification de littéral	base octale pour littéral
o'	qualification de littéral	base octale pour littéral
W'	qualification de littéral	caractère large ou littéral de chaîne de caractères
w'	qualification de littéral	caractère large ou littéral de chaîne de caractères
– –	fin de ligne	délimiteur de fin de ligne des commentaires en ligne

Appendice III

Chaînes de nom simple spéciales

III.1 Chaînes de nom simple réservées

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 Chaînes de nom simple prédéfinies

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 Noms d'exception

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

Appendice IV

Exemples de programmes

IV.1 Opérations sur les entiers

```
1     integer_operations:
2     MODULE
3
4         add:
5         PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
6             RESULT i+j;
7         END add;
8
9         mult:
10        PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
11            RESULT i*j;
12        END mult;
13
14        GRANT add, mult;
15        SYNMODE operand_mode=INT;
16        GRANT operand_mode;
17        SYN   neutral_for_add=0,
18              neutral_for_mult=1;
19        GRANT neutral_for_add,
20              neutral_for_mult;
21
22    END integer_operations;
```

IV.2 Mêmes opérations sur les fractions

```
1     fraction_operations:
2     MODULE
3         NEWMODE fraction=STRUCT (num,denum INT);
4
5         add:
6         PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
7             RETURN [f1.num*f2.denum+f2.num*f1.denum,f1.denum*f2.denum];
8         END add;
9
10        mult:
11        PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
12            RETURN [f1.num*f2.num,f2.denum*f1.denum];
13        END mult;
14
15        GRANT add, mult;
16        SYNMODE operand_mode=fraction;
17        GRANT operand_mode;
18        SYN   neutral_for_add fraction=[ 0,1 ],
19              neutral_for_mult fraction=[ 1,1 ];
20        GRANT neutral_for_add,
21              neutral_for_mult;
22
23    END fraction_operations;
```

IV.3 Mêmes opérations sur les nombres complexes

```
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 Arithmétique d'ordre général

```
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 Additionner bit à bit et vérifier le résultat

```
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 Jouer avec les dates

```
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 Nombres romains

```
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 Compter les lettres dans une chaîne de caractères de longueur arbitraire

```
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 Nombres premiers

```
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 Implémenter des piles de deux manières différentes, transparentes pour l'utilisateur

```
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 Fragments pour jouer aux échecs

```
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 Construire et manipuler une liste chaînée circulairement

```
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 Une région pour donner des accès compétitifs à une ressource

```
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 Mettre en attente les appels à un 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 Affecter et désaffecter un ensemble de ressources

```
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 Affecter et désaffecter un ensemble de ressources en employant des tampons

```
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 Parcours de chaîne 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 Parcours de chaîne 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 Enlever un élément d'une liste doublement chaînée

```
queue: MODULE
    SYNMODE info=INT;
    queue_removal:
    MODULE
        SEIZE info;
        GRANT remove;
        remove:
        PROC(p PTR) RETURNS (info) EXCEPTIONS (EMPTY);
                /* This procedure removes the item referred to
                   by p from a queue and returns the information
                   contents of that queue element */
                SYNMODE element = STRUCT (
                                  i info POS (0,8:31),
                                  prev PTR POS (1,0:15),
                                  next PTR POS (1,16:31));
                DCL x REF element LOC:= element(p), prev, next PTR;
                prev := x–>.prev;
                next := x–>.next;
                x–>.prev, x–>.next := NULL;
                RESULT x–>.i;
                p := prev;
                x–>.next := next;
                p := next;
                x–>.prev := prev;
        END remove;
    END queue_removal;
END queue;
```

IV.20 Mettre à jour un fichier

```
read_modify_write:
MODULE

    /* this example indicates how the CHILL i/o concepts can be used     */
    /* to write an application where a record of a random accessible     */
    /* file can be updated or added if not yet in use                    */

    NEWMODE
        index_set = INT (1:1000),
        record_type = STRUCT (
                                  free      BOOL,
                                  count     INT,
                                  name      CHARS (20));

    DCL
        curindex            index_set,
        file_association    ASSOCIATION,
        record_file         ACCESS (index_set) record_type,
        record_buffer       record_type;

    ASSOCIATE (file_association,"DSK:RECORDS.DAT");   /* create association   */
    CONNECT (record_file,file_association,READWRITE); /* connect to file      */
    curindex := 123;                                  /* position record      */
    READRECORD (record_file,curindex,record_buffer);  /* read the record      */
        IF record_buffer.free                         /* if record is free    */
        THEN                                      /* the claim and        */
            record_buffer.free := FALSE           /* initialize it        */
            record_buffer.count := 0;
            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 Fusionner deux fichiers triés

```
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 Lire un fichier ayant des enregistrements de longueur 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 L'emploi de modules 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 Exemple d'un contexte

```
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 L'emploi de la préfixation et de modules distants

```
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 L'emploi d'e/s de texte

```
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 Une pile générique

```
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 type de données abstrait

```
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 Exemple d'un module 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 Orientation-objet: modes pour piles séquentielles simples

```
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 Orientation objet: extension de mode: pile séquentielle simple avec opération "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 Orientation objet: modes pour des piles à synchronisation d'accès

```
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
```

```
MainProgram4: MODULE
   SEIZE RegionStackWithTopMode1;
   DCL Stack1 RegionStackWithTopMode1;
   Producer: PROCESS ( ) ;
      DCL Elem1 RegionStackWithTopMode1!ElementMode;
      DO FOR EVER
         /* compute Elem1 */
         Stack1.Push(Elem1);
      OD;
   END Producer;
   Consumer: PROCESS ( ) ;
      DCL Elem1 RegionStackWithTopMode1!ElementMode;
      DO FOR EVER
         Elem1 := Stack1.Pop ( );
         /* process Elem1 */
      OD;
   END Consumer;
   START Producer ( );
   START Consumer ( );
END MainProgram4;

SYNMODE RegionStackWithTopMode2 = REGION SPEC BASED_ON StackWithTopMode2
      /* In case of an erroneous use different behaviour as StackWithTopMode2:
         use the event mechanism */
   GRANT Push, Pop, Top;
   Push: PROC(Elem ElementMode IN) REIMPLEMENT END Push;
   Pop: PROC( ) RETURNS (ElementMode) REIMPLEMENT END Pop;
   Top: PROC( ) RETURNS (ElementMode) REIMPLEMENT END Top;
   DCL NotEmpty, NotFull EVENT;
END RegionStackWithTopMode2 ;

SYNMODE RegionStackWithTopMode2 = REGION BODY BASED_ON StackWithTopMode2
   Push: PROC(Elem ElementMode IN) REIMPLEMENT
      PushLoop: DO
         BEGIN
            StackWithTopMode2!Push(Elem);
            EXIT PushLoop;
         END
         ON (Overflow): DELAY NotFull; END;
      OD PushLoop;
      CONTINUE NotEmpty;
   END Push;
   Pop: PROC( ) RETURNS(ElementMode) REIMPLEMENT
      PopLoop: DO
         BEGIN
            RESULT StackWithTopMode2!Pop( );
            EXIT PopLoop;
         END
         ON (Underflow): DELAY NotEmpty; END;
      OD PopLoop;
      CONTINUE NotFull;
   END Pop;
   Top: PROC( ) RETURNS (ElementMode) REIMPLEMENT
      TopLoop: DO
         BEGIN
            RESULT StackWithTopMode2!Top( );
            EXIT TopLoop;
         END
         ON (EmptyStack): DELAY NotEmpty; END;
      OD TopLoop;
      CONTINUE NotFull;
   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;
```

Appendice V

Caractéristiques qui ne sont plus en vigueur

Les caractéristiques décrites dans le présent appendice ne font pas partie de la présente Recommandation | Norme internationale, mais elles faisaient partie de la Recommandation Z.200 (1984), dans le *Livre rouge*, (tome VI, fascicule VI.12) et dans la Recommandation Z.200 (1988), *Livre bleu*, (tome X, fascicule X.6). On en trouvera ci-après une brève description; leur définition complète se trouve dans les paragraphes correspondants des versions de 1984 de la Recommandation en question, qui sont mentionnées ci-après. Ces caractéristiques peuvent être acceptées par une implémentation. En l'absence d'indication, se référer à la Recommandation Z.200 (1984).

V.1 Directive de libération (paragraphe 2.6)

Une directive de libération a libéré les chaînes de nom simple **réservées** spécifiées dans la *liste de chaînes de nom simple réservées*, de sorte qu'elles ont pu être redéfinies.

V.2 Syntaxe de mode entier (paragraphe 3.4.2)

BIN était la syntaxe dérivée pour *INT*.

V.3 Modes ensemble avec des trous (paragraphe 3.4.5)

Un mode ensemble a défini un ensemble de valeurs nommées ou anonymes. Un mode ensemble était un mode ensemble **avec des trous**, si et seulement si le nombre de ses noms d'**élément d'ensemble** était inférieur au **nombre de valeurs** du mode ensemble.

V.4 Syntaxe des modes procédure (paragraphe 3.7)

Une *spec de résultat* sans la chaîne de nom simple **réservée** optionnelle **RETURNS** était la syntaxe dérivée pour la *spec de résultat* avec **RETURNS**.

V.5 Syntaxe des modes chaîne (paragraphe 3.11.2)

Les notations **CHAR** *(n)* et **BIT** *(n)* dénotaient respectivement des chaînes de caractères et des chaînes binaires.

V.6 Syntaxe des modes matrice (paragraphe 3.11.3)

La chaîne de nom simple **réservée ARRAY** était optionnelle.

V.7 Notation étagée de structures (paragraphe 3.11.5)

Un *mode structure étagé* était la syntaxe dérivée pour un *mode structure imbriqué*. Dans la notation étagée de structures, les champs étaient précédés d'un numéro de niveau. Si une structure contenait des champs qui étaient eux-mêmes des structures ou des matrices de structures, une hiérarchie de structures était formée et un numéro de niveau pouvait être associé à chaque champ. Au lieu d'écrire des modes structure imbriqués, il a été autorisé dans le *mode structure étagé* d'écrire le numéro de niveau devant le nom de champ.

V.8 Noms de référence d'implantation (paragraphe 3.11.6)

Les noms de référence d'implantation pouvaient être utilisés pour spécifier le mappage d'une manière définie par l'implémentation.

V.9 Déclarations de locus avec base (paragraphe 4.1.4)

Une déclaration de locus avec base sans nom de *locus référence liée ou libre* était la syntaxe dérivée pour un énoncé de définition de synmode. Une déclaration de locus avec base avec un nom de *locus référence liéee ou libre* définissait un ou plusieurs noms d'accès. Ces noms constituaient un autre moyen pour accéder à un locus en déréférençant la valeur référence contenue dans le locus référence spécifié. Cette opération de déréférenciation était accomplie chaque fois que, et seulement quand un accès était obtenu via un nom déclaré avec **base**.

V.10 Littéraux chaîne de caractères (paragraphe 5.2.4.6)

Les littéraux chaîne de caractères étaient délimités par des caractères apostrophe. Outre la représentation imprimable, la représentation hexadécimale pouvait être utilisée. Les littéraux chaîne de caractères de longueur un servaient de littéraux de caractère.

V.11 Expressions recevoir [voir 5.3.9/Z.200 (1988)]

Les expressions recevoir étaient utilisées pour recevoir des valeurs depuis des locus tampon. Le processus d'exécution pouvait être reporté et pouvait réactiver un autre processus reporté par l'envoi d'une valeur au locus tampon spécifié.

V.12 Notation Addr (paragraphe 5.3.8)

ADDR (<*locus*>) était la syntaxe dérivée pour –> <*locus*>.

V.13 Syntaxe d'affectation (paragraphe 6.2)

Le symbole = était la syntaxe dérivée pour le symbole *:=* .

V.14 Syntaxe d'action à cas (paragraphe 6.4)

La *liste d'intervalles* d'une *action à cas* pouvait être spécifiée plus généralement par un *mode discret* et pas seulement par un *nom de mode discret*.

V.15 Syntaxe action faire-pour (paragraphe 6.5.2)

L'intervalle dans l'*énumération par intervalle* d'une action faire-pour pouvait être spécifié plus généralement par un mode *discret* et pas seulement par un *nom de mode discret*.

V.16 Compteurs de boucles explicites (paragraphe 6.5.2)

Si un nom d'accès était visible dans le domaine où était située l'*action faire*, qui était égal à un des noms définis par un *compteur de boucles*, alors, le *compteur de boucles* était **explicite**; sinon, il était **implicite**. Dans le premier cas, la valeur du compteur de boucles était stockée dans le locus dénoté juste avant la terminaison anormale. Une distinction était faite entre terminaison **normale** et terminaison **anormale**. Il y avait terminaison normale lorsque l'évaluation d'un au moins des compteurs de boucles indiquait une terminaison. Il y avait terminaison anormale lorsque l'évaluation d'une condition tandis que donnait *FALSE* ou si l'action faire était abandonnée par un transfert de commande en dehors d'elle.

V.17 Syntaxe d'action appeler (paragraphe 6.7)

La chaîne de nom simple **réservée CALL** était facultative. Une *action appeler* avec **CALL** était dérivée d'une *action appeler* sans **CALL**.

V.18 Exception RECURSEFAIL (paragraphe 6.7)

L'exception *RECURSEFAIL* était causée quand une procédure **non récursive** s'appelait elle-même récursivement.

V.19 Syntaxe d'action démarrer (paragraphe 6.13)

L'*action démarrer* avec l'option **SET** était la syntaxe dérivée pour l'action d'affectation simple:

<*locus instance*> *::=* <*expression démarrer*>.

V.20 Noms explicites de valeur reçue (paragraphe 6.19)

Une action recevoir signal avec cas et une action recevoir tampon avec cas pouvaient introduire des noms de **valeur reçue**. Si un nom était visible dans le domaine où l'*action recevoir signal avec cas* était placée, ce qui était égal à l'un des noms introduits après **IN**, le nom de **valeur reçue** était **explicite**; sinon, il était **implicite**. Dans le premier cas, la valeur reçue était enregistrée dans le locus désigné immédiatement avant l'exécution de la liste d'énoncés d'action.

V.21 Blocs (paragraphe 8.1)

L'*action conditionnelle*, l'*action à cas*, l'*action faire* et l'*action attente* n'étaient pas définies comme des blocs.

V.22 Enoncé d'entrée (paragraphe 8.4)

Une procédure pouvait avoir des points d'entrée multiples au moyen d'énoncés d'entrée. Ces énoncés étaient considérés comme des définitions de procédure supplémentaires. La définition dans l'énoncé d'entrée définissait le nom du point d'entrée de la procédure. Le point d'entrée était défini par la position textuelle de l'énoncé d'entrée.

V.23 Noms de registre (paragraphe 8.4)

Une spécification de registre pouvait être donnée dans le paramètre formel de la procédure, et dans la spec de résultat. Dans le cas d'un passage par valeur, cela signifiait que la valeur effective était contenue dans le registre spécifié; dans le cas d'un passage par locus, cela signifiait que le pointeur (caché) vers le locus effectif était contenu dans le registre spécifié. Si la spécification se trouvait dans la spec de résultat, cela signifiait que la valeur retournée ou le pointeur (caché) vers le locus retourné était contenu dans le registre spécifié.

V.24 Attribut récursif [10.4/Z.200 (1988)]

La **récursivité** des procédures était une caractéristique par défaut propre à l'implémentation, à moins que l'attribut **RECURSIVE** n'était spécifié dans une liste d'attributs de procédure.

V.25 Enoncés de quasi-cause et quasi-filets (paragraphe 8.10.3)

Les énoncés de quasi-cause signalaient la présence d'énoncés de cause dans des modules distants ou des régions distantes directement englobés dans le domaine englobant directement le domaine du module de spec ou de la région de spec dans lequel était placé l'énoncé de quasi-cause. Les quasi-filets signalaient la présence d'un filet dans le programme, atteignable depuis le module, la région ou le contexte directement englobé dans le contexte auquel le filet était adjoint.

V.26 Syntaxe des quasi-énoncés [10.10.3/Z.200 (1988)]

Les énoncés de quasi-procédure et de définition de processus étaient terminés par une **END** <*chaîne de nom simple*>.

V.27 Noms faiblement visibles et énoncés de visibilité [12.2.1/Z.200 (1988)]

Une *chaîne de nom* qui n'était pas **fortement visible** dans un domaine était dite **faiblement visible** dans ce domaine si elle était sous-entendue par une *chaîne de nom* qui était **fortement visible** dans ce domaine. Une *chaîne de nom* dans le domaine était **liée** à des occurences de définition de **impliqué**. Si elle ne définissait pas le même élément d'ensemble de modes ensemble **similaires**, une **faible discordance** se produisait; sinon, la *chaîne de nom* était **liée** à ces définitions. La section 12.2.4 définissait les occurences de définition **impliquées** pour les noms.

V.28 Noms faiblement visibles et énoncés de visibilité (paragraphe 10.2.4.3)

Une *chaîne de nom* NS faiblement visible dans le domaine R était dite saisissable par le modulion M immédiatement englobé dans R si NS était liée à R dans une *occurence de définition* non englobée dans le domaine de M. Une *chaîne de nom* NS faiblement visible dans le domaine R du modulion M était dite octroyable par M si NS était liée dans R à une *occurence de définition* englobée dans R.

V.29 Envahissement (paragraphe 10.2.4.4)

Quand un énoncé d'octroi contenait (**DIRECTLY**) **PERVASIVE**, toutes les *chaînes de nom* octroyées par cet énoncé avaient la propriété d'envahissement (direct) dans les domaines englobants du modulion dans lequel était contenu l'*énoncé d'octroi*. Les chaînes de nom:

- étaient fortement visibles dans un domaine S directement englobant de M;
- avaient aussi, lorsqu'elles étaient pourvues de la propriété d'**envahissement direct** dans S, la propriété d'**envahissement direct** dans M;
- lorsqu'elles n'étaient pas **directement fortement visibles** dans un domaine R et qu'elles étaient **fortement visibles** dans un domaine qui englobait directement R et dans lequel elles avaient la propriété d'envahissement**,** étaient **indirectement fortement visibles** dans R et avaient aussi la propriété d'envahissement dans R.

V.30 Saisie par nom de modulion (paragraphe 10.2.4.5)

Si une *clause renommer préfixe* d'un *énoncé de saisie* avait un *postfixe de saisie* contenant une *chaîne de nom* de modulion et **ALL**, la *clause renommer préfixe* était équivalente à un ensemble *d'énoncés de saisie*, pour toute chaîne de nom fortement visible dans le domaine qui englobait immédiatement le modulion dans lequel était placé l'*énoncé de saisie*; elle était saisissable par ce modulion et octroyée par le modulion lié au *nom de modulion* dans le domaine immédiatement englobant le modulion dans lequel l'*énoncé de saisie* était placé.

V.31 Chaînes de nom simple prédéfinies (paragraphe C.2)

AND, NOT, OR, REM, MOD, THIS et *XOR* étaient des chaînes de nom simple prédéfinies.

Appendice VI

Index des règles de production

Les numéros de page en caractères gras renvoient aux définitions d'un élément; ceux en caractères normaux renvoient aux occurences d'utilisation des éléments de l'index.

SERIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, de télégraphie, de télécopie, circuits téléphoniques et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux de données et communication entre systèmes ouverts
Série Y	Infrastructure mondiale de l'information et protocole Internet
Série Z	**Langages et aspects informatiques généraux des systèmes de télécommunication**

Genève, 2001