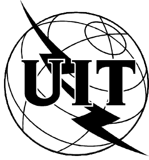


UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

Z.200

(11/1988)

SÉRIE Z: LANGAGES ET ASPECTS INFORMATIQUES GENERAUX DES SYSTEMES DE TELECOMMUNICATION

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

Réédition de la Recommandation du CCITT Z.200,
publiée dans le Livre Bleu, Fascicule X.6 (1988)

NOTES

1 La Recommandation Z.200 du CCITT a été publiée dans le fascicule X.6 du Livre Bleu. Ce fichier est un extrait du Livre Bleu. La présentation peut en être légèrement différente, mais le contenu est identique à celui du Livre Bleu et les conditions en matière de droits d'auteur restent inchangées (voir plus loin).

2 Dans la présente Recommandation, le terme «Administration» désigne indifféremment une administration de télécommunication ou une exploitation reconnue.

© UIT 1988, 2010

Tous droits réservés. Aucune partie de cette publication ne peut être reproduite, par quelque procédé que ce soit, sans l'accord écrit préalable de l'UIT.

Recommandation Z.200

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

(Genève, 1988)

TABLE DES MATIÈRES

Page

1. *Introduction* 1
   - 1.1 Généralités 1
   - 1.2 Vue générale du langage 1
   - 1.3 Modes et classes 2
   - 1.4 Locus et leurs accès 2
   - 1.5 Valeurs et leurs opérations 3
   - 1.6 Actions 3
   - 1.7 Entrée et sortie 3
   - 1.8 Traitement des exceptions 4
   - 1.9 Supervision temporelle 4
   - 1.10 Structure des programmes 4
   - 1.11 Exécution concurrente 5
   - 1.12 Propriétés sémantiques générales 5
   - 1.13 Options pour l'implémentation 5

2. *Préliminaires* 7
   - 2.1 Métalangage 7
     - 2.1.1 Description de la syntaxe acontextuelle 7
     - 2.1.2 Description sémantique 7
     - 2.1.3 Exemples 8
     - 2.1.4 Règles d'identification dans le métalangage 8
   - 2.2 Vocabulaire 8
   - 2.3 Espacements 9
   - 2.4 Commentaires 9
   - 2.5 Commandes de mise en page 9
   - 2.6 Directives au compilateur 10
   - 2.7 Noms et leurs définitions 10

Page

3. *Modes et classes* 12

3.1 Généralités 12

3.1.1 Modes 12

3.1.2 Classes 12

3.1.3 Propriétés des modes, des classes et leurs relations 12

3.2 Définitions de modes 13

3.2.1 Généralités 13

3.2.2 Définitions de synmodes 14

3.2.3 Définitions de neumodes 14

3.3 Classification des modes 15

3.4 Modes discret 16

3.4.1 Généralités 16

3.4.2 Modes entier 16

3.4.3 Modes booléen 17

3.4.4 Modes caractère 17

3.4.5 Modes ensemble 18

3.4.6 Modes intervalle 19

3.5 Modes ensembliste 20

3.6 Modes repère 20

3.6.1 Généralités 20

3.6.2 Modes repère lié 21

3.6.3 Modes repère libre 21

3.6.4 Modes descripteur 21

3.7 Modes procédure 22

3.8 Modes exemplaire 23

3.9 Modes de synchronisation 23

3.9.1 Généralités 23

3.9.2 Modes événement 24

3.9.3 Modes tampon 24

3.10 Modes d'entrée-sortie 25

3.10.1 Généralités 25

3.10.2 Modes association 25

3.10.3 Modes accès 25

3.10.4 Modes texte 26

3.11 Modes temporisation 27

3.11.1 Généralités 27

3.11.2 Modes durée 27

3.11.3 Modes temps absolu 27

Page

3.12 Modes composés 28

3.12.1 Généralités 28

3.12.2 Modes chaîne 28

3.12.3 Modes rangée 29

3.12.4 Modes structure 31

3.12.5 Description d'implantation pour modes rangée et modes structure 34

3.13 Modes dynamiques 37

3.13.1 Généralités 37

3.13.2 Modes chaîne dynamiques 37

3.13.3 Modes rangée dynamiques 37

3.13.4 Modes structure paramétrés dynamiques 37

4. *Les locus et leurs accès* 39

4.1 Déclarations 39

4.1.1 Généralités 39

4.1.2 Déclarations de locus 39

4.1.3 Déclarations de loc-identité 40

4.2 Les locus 41

4.2.1 Généralités 41

4.2.2 Noms d'accès 42

4.2.3 Repères liés dérepérés 42

4.2.4 Repères libres dérepérés 43

4.2.5 Descripteurs dérepérés 43

4.2.6 Eléments de chaîne 44

4.2.7 Tranches de chaîne 45

4.2.8 Eléments de rangée 46

4.2.9 Tranches de rangée 46

4.2.10 Champs de structure 47

4.2.11 Appels de procédure rendant locus 48

4.2.12 Appels d'opération prédéfinie rendant locus 48

4.2.13 Conversions de locus 49

5. *Valeurs et leurs opérations* 50

5.1 Définitions de synonymes 50

5.2 Valeur primitive 50

5.2.1 Généralités 50

5.2.2 Contenu de locus 51

5.2.3 Noms de valeur 51

Page

5.2.4 Littéraux 52

5.2.4.1 Généralités 52

5.2.4.2 Littéraux d'entier 53

5.2.4.3 Littéraux de booléen 53

5.2.4.4 Littéraux de caractère 54

5.2.4.5 Littéraux d'ensemble 54

5.2.4.6 Littéral de vide 54

5.2.4.7 Littéraux de chaîne de caractères 55

5.2.4.8 Littéraux de chaîne de bits 56

5.2.5 Multiplets 56

5.2.6 Valeurs élément de chaîne 60

5.2.7 Valeurs tranche de chaîne 60

5.2.8 Valeurs élément de rangée 61

5.2.9 Valeurs tranche de rangée 62

5.2.10 Valeurs champ de structure 63

5.2.11 Conversions d'expression 63

5.2.12 Appels de procédure rendant valeur 64

5.2.13 Appels d'opération prédéfinie rendant valeur 64

5.2.14 Expressions démarrer 65

5.2.15 Opérateur nullaire 65

5.2.16 Expression parenthésée 65

5.3 Valeurs et expressions 66

5.3.1 Généralités 66

5.3.2 Expressions 67

5.3.3 Opérande-0 68

5.3.4 Opérande-1 69

5.3.5 Opérande-2 69

5.3.6 Opérande-3 71

5.3.7 Opérande-4 72

5.3.8 Opérande-5 73

5.3.9 Opérande-6 74

6. *Actions* 75

6.1 Généralités 75

6.2 Action d'affectation 75

6.3 Action conditionnelle 77

6.4 Action de cas 78

6.5 Action faire 79

6.5.1 Généralités 79

6.5.2 Commande pour 80

6.5.3 Commande tandis 82

6.5.4 Partie avec 83

Page

6.6 Action sortir 83
6.7 Action appeler 84
6.8 Action résulter et action revenir 86
6.9 Action aller 87
6.10 Action affirmer 87
6.11 Action vide 87
6.12 Action causer 88
6.13 Action démarrer 88
6.14 Action arrêter 88
6.15 Action continuer 88
6.16 Action mettre en attente 89
6.17 Action mettre en attente et choisir 90
6.18 Action envoyer 91
6.18.1 Généralités 91
6.18.2 Action envoyer signal 91
6.18.3 Action envoyer tampon 92
6.19 Action recevoir et choisir 92
6.19.1 Généralités 92
6.19.2 Action recevoir signal et choisir 93
6.19.3 Action recevoir tampon et choisir 94
6.20 Appels d'opération prédéfinie CHILL 95
6.20.1 Appels d'opération prédéfinie simple CHILL 95
6.20.2 Appels d'opération prédéfinie rendant locus CHILL 95
6.20.3 Appels d'opération prédéfinie rendant valeur CHILL 96
6.20.4 Opérations prédéfinies de traitement de mémoire dynamique 98

7. *Entrée et sortie* 100
7.1 Modèle de référence E/S 100
7.2 Valeurs d'association 101
7.2.1 Généralités 101
7.2.2 Attributs des valeurs d'association 101
7.3 Valeurs d'accès 102
7.3.1 Généralités 102
7.3.2 Attributs des valeurs d'accès 102
7.4 Opérations prédéfinies pour entrée-sortie 102
7.4.1 Généralités 102
7.4.2 Association avec un objet du monde extérieur 103
7.4.3 Dissociation d'un objet du monde extérieur 103
7.4.4 Accès aux attributs association 104
7.4.5 Modification des attributs association 104
7.4.6 Connexion d'un locus accès 105
7.4.7 Déconnexion d'un locus accès 107
7.4.8 Attributs d'accès de locus accès 107
7.4.9 Opérations de transfert de données 108

Page

7.5 Entrée/sortie de texte 110

7.5.1 Généralités 110

7.5.2 Attributs des valeurs de texte 110

7.5.3 Opérations de transfert de texte 111

7.5.4 Chaîne de commande de format 113

7.5.5 Conversion 114

7.5.6 Edition 116

7.5.7 Commande d'EIS 117

7.5.8 Accès aux attributs d'un locus texte 118

8. *Filets d'exception* 120

8.1 Généralités 120

8.2 Filets 120

8.3 Identification de filet 120

9. *Temporisation* 122

9.1 Généralités 122

9.2 Processus temporisables 122

9.3 Actions de temporisation 122

9.3.1 Action de temporisation relative 122

9.3.2 Action de temporisation absolue 123

9.3.3 Action de temporisation cyclique 123

9.4 Opérations prédéfinies pour le temps 124

9.4.1 Opérations prédéfinies de durée 124

9.4.2 Opération prédéfinie de temps absolu 124

9.4.3 Appel d'opération prédéfinie de temporisation 125

10. *Structure de Programme* 127

10.1 Généralités 127

10.2 Domaines et imbrication 128

10.3 Blocs début-fin 130

10.4 Définitions de procédure 131

10.5 Définitions de processus 133

10.6 Modules 134

10.7 Régions 135

10.8 Programme 135

10.9 Allocation de mémoire et durée de vie 136

10.10 Constructions pour la programmation par fragments 136

10.10.1 Fragments distants 136

10.10.2 Modules de spec, régions de spec et contextes 138

10.10.3 Quasi-énoncés 139

10.10.4 Correspondance entre quasi-définitions et définitions 140

11. *Exécution concurrente* 142
11.1 Les processus et leurs définitions 142
11.2 Exclusion mutuelle et régions 142
11.2.1 Généralités 142
11.2.2 Régionalité 143
11.3 Mise en attente d'un processus 144
11.4 Réactivation d'un processus 145
11.5 Enoncés de définition de signal 145

12. *Propriétés sémantiques générales* 146
12.1 Règles de vérification des modes 146
12.1.1 Propriétés des modes et des classes 146
12.1.1.1 Propriété de protection 146
12.1.1.2 Modes paramétrables 146
12.1.1.3 Propriété de repérer 146
12.1.1.4 Propriété de marquage et de paramétrage 146
12.1.1.5 Propriété de non-valeur 147
12.1.1.6 Mode racine 147
12.1.1.7 Classe résultante 147
12.1.2 Relations entre modes et classes 148
12.1.2.1 Généralités 148
12.1.2.2 Relations d'équivalence sur les modes 148
12.1.2.3 La relation similaire 148
12.1.2.4 La relation v-équivalent 149
12.1.2.5 La relation équivalent 149
12.1.2.6 La relation l-équivalent 150
12.1.2.7 Les relations équivalent et l-équivalent pour les champs 150
12.1.2.8 La relation équivalent pour les implantations 150
12.1.2.9 La relation semblable 151
12.1.2.10 Les relations semblables pour les champs 152
12.1.2.11 La relation liée par la nouveauté 152
12.1.2.12 La relation compatible en lecture 153
12.1.2.13 Les relations équivalent dynamique et compatible en lecture dynamique 154
12.1.2.14 La relation limitable à 154
12.1.2.15 Compatibilité entre un mode et une classe 155
12.1.2.16 Compatibilité entre classes 155
12.2 Visibilité et identification 155
12.2.1 Degrés de visibilité 156
12.2.2 Conditions de visibilité et identification 156
12.2.3 Visibilité dans les domaines 157
12.2.3.1 Généralités 157
12.2.3.2 Enoncés de visibilité 158
12.2.3.3 Clause renommer préfixe 158

Page

12.2.3.4 Enoncé d'octroi 159

12.2.3.5 Enoncé de saisie 161

12.2.4 Représentations textuelles de nom impliquées 162

12.2.5 Visibilité de noms de champ 164

12.3 Sélection de cas 164

12.4 Définition et résumé des catégories sémantiques 166

12.4.1 Noms 166

12.4.2 Locus 167

12.4.3 Expressions et valeurs 167

12.4.4 Catégories sémantiques diverses 168

13. *Options pour l'implémentation* 169

13.1 Opérations prédéfinies par l'implémentation 169

13.2 Modes entier définis par l'implémentation 169

13.3 Noms de processus définis par l'implémentation 169

13.4 Filets définis par l'implémentation 169

13.5 Noms d'exception définis par l'implémentation 169

13.6 Autres caractéristiques définies par l'implémentation 169

Appendice A: Ensembles de caractères pour le langage CHILL 171

Appendice B: Symboles spéciaux et combinaisons de caractères 172

Appencide C: Représentations textuelles de nom simple spéciales 173

C.1: Représentations textuelles de nom simple réservées 173

C.2: Représentations textuelles de nom simple prédéfinies 174

C.3: Noms d'exception 175

Appendice D: Exemples de programmes 176

Appendice E: Caractéristiques retirées 202

Appendice F: Ensemble de règles de production 205

Appendice G: Index des règles de production 230

Appendice H: Index 239

1. INTRODUCTION

La présente Recommandation définit le langage de programmation de haut niveau CHILL du CCITT (CHILL = CCITT High Level Language).

Les sous-sections suivantes du présent chapitre 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 du CCITT «Introduction to CHILL» et «CHILL user's manual».

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

1.1 GÉNÉRALITES

CHILL est un langage bien défini, structuré en blocs et conçu avant tout pour la mise en œuvre de grands systèmes complexes et intégrés.

CHILL est destiné à:

- améliorer la fiabilité et l'efficacité d'exécution au moyen d'un grand nombre de contrôles à la compilation;
- couvrir, grâce à sa souplesse et à sa puissance, la gamme d'applications nécessaire et à exploiter différentes espèces de matériel;
- encourager, en fournissant certaines facilités, le développement progressif et modulaire de grands systèmes;
- permettre des mises en œuvre en temps réel en fournissant des primitives intégrées de concurrence et de temporisation;
- 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.

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. Le non respect des règles dynamiques de CHILL se traduit par des exceptions à l'exécution qui peuvent être interceptées par un filet d'exception approprié (toutefois, la génération de tels contrôles implicites est facultative, à moins qu'un filet défini par l'usager soit explicitement spécifiée).

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

Les objets informatifs 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 informatifs manipulables de CHILL sont les valeurs et les locus où les valeurs peuvent être placées. Les actions définissent les opérations à effectuer sur les objets informatifs 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 informatifs.

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

Dans les sections qui suivent, on récapitule les différents concepts de CHILL. Chaque section est une introduction à un chapitre 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, repérabilité, 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.

CHILL a les catégories de mode suivantes:

modes discrets	modes entier, caractère, booléen, ensemble (symbolique) ainsi que leurs intervalles;
modes ensemblistes	ensembles d'éléments d'un mode discret;
modes repère	repères liés, repères libres et descripteurs utilisés comme repères de locus;
modes composés	modes chaîne, rangée et structure;
modes procédure	procédures considérées comme objets informatifs manipulables;
modes exemplaire	identifications de processus;
modes de synchronisation	modes événement et tampon pour la synchronisation des processus et la communication;
modes d'entrée-sortie	modes d'association d'accès et de texte pour les opérations d'entrée-sortie;
modes de temporisation	modes durée et temps absolu pour la temporisation.

CHILL fournit des notations pour un ensemble de modes standards. 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 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 (abstraits) 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 de loc-identité.

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 repère (voir ci-dessous) du locus nouvellement créé.

Un locus peut être **repérable**. Cela signifie qu'il correspond au locus une valeur repère de ce locus. Cette valeur repère est obtenue comme résultat de l'opération qui consiste à repérer le locus **repérable**. En dérepérant une valeur repère, on obtient le locus repéré. CHILL exige que certains locus soient toujours **repérables**; mais pour d'autres locus, on laisse l'implémentation décider s'ils sont **repérables** ou non. La propriété d'être ou non repérable 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 **repérable**. 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 rangées, 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:

repérabilité: un repère existe-t-il 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 de CHILL), soit une **valeur indéfinie** (au sens de CHILL). L'utilisation d'une valeur indéfinie dans des contextes déterminés produit une situation indéfinie (au sens de CHILL) et le programme est considéré incorrect.

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 repérer 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 d'opération prédéfinie invoque une opération prédéfinie (une opération 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 de 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 causer	pour causer 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 concurrents d'actions, CHILL fournit les actions de cas démarrer, arrêter, mettre en attente, continuer, envoyer, mettre en attente et choisir, recevoir ainsi que les expressions et recevoir et choisir.

1.7 ENTRÉE ET SORTIE

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

Le modèle repère 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 d'opération prédéfinie, de lire et de modifier les attributs des associations définis par le langage, c'est-à-dire **existants, visibles, 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 de 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.) 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 spécifiable (c'est-à-dire si l'exception a un nom) et 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 de 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 CHILL, la condition dynamique associée doit être contrôlée.

1.9 SUPERVISION TEMPORELLE

La supervision temporelle de CHILL permet de réagir au déroulement du temps dans le monde extérieur. Les processus CHILL ne peuvent être interrompus qu'à des instants **temporisables** au cours de l'exécution. Lorsque cela se produit, le contrôle est transféré à un processus 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 opérations 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 et la **région**. 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 opérations 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 et les régions fournissent les moyens de réaliser une structure d'exécutions concurrentes.

Un programme CHILL complet est une liste de modules ou de régions 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.

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 CONCURRENTE

CHILL prévoit l'exécution concurrente d'unités de programme. Le processus est l'unité d'exécution concurrente. L'évaluation d'une expression démarrer cause la création d'un nouveau processus de la définition de processus indiquée. Ce processus est alors considéré comme exécuté concurremment avec le processus 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, termine ce processus.

Un processus 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 processus, la transition de l'état en attente à l'état actif est appelée la réactivation du processus. L'exécution d'actions de mise en attente sur des événements, d'actions de réception sur des tampons ou signaux, ou d'actions envoyer sur des tampons, peut mettre en attente le processus qui les exécute. L'exécution d'actions continuer sur des événements, d'actions envoyer sur des tampons ou signaux, ou d'actions recevoir sur des tampons, peut rendre de nouveau actif un processus en attente.

Les tampons et les événements sont des locus à utilisation restreinte. Les opérations envoyer, recevoir et recevoir et choisir sont définies sur les tampons; les opérations mettre en attente, mettre en attente et choisir 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 et choisir 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 processus.

1.12 PROPRIÉTÉS SÉMANTIQUES GÉNÉRALES

Les conditions sémantiques (liées au contexte) de 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 et aussi d'éventuels noms **impliqués** des noms 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

CHILL permet des modes entier définis par l'implémentation, des opérations 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 neumode non spécifié en CHILL. Il est permis d'étendre aux modes entier définis par l'implémentation les opérations arithmétiques existantes prédéfinies par CHILL, dans le cadre des règles syntaxiques et sémantiques de 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.

NOTE

La Recommandation Z.200 a été élaborée par le Comité consultatif international télégraphique et téléphonique (CCITT) de l'Union internationale des télécommunications (UIT) et a été adoptée selon une procédure accélérée, en tant que norme internationale ISO/CEI 9496 par l'ISO (Organisation internationale de normalisation) et la CEI (Commission électrotechnique internationale) dans le cadre de leur Comité technique mixte ISO/IEC ITC 1.

Le texte de la Recommandation Z.200 du CCITT sert également de norme ISO/CEI 9496.

2 PRÉLIMINAIRES

2.1 MÉTALANGAGE

La description de 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 d'une extension de la forme de Backus-Naur (BNF). Les catégories syntaxiques sont indiquées par un ou plusieurs mots français, écrits en caractères 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 consistant chacune en productions non terminales et/ou terminales. Ces constructions sont séparées par une barre verticale (|) et dénotent 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 la section 2.1.2).

Des éléments syntaxiques peuvent être groupés par l'utilisation d'accolades ({ et }). La répétition d'un groupe entre accolades est indiquée par un astérisque (*) ou un plus (+). Un astérisque indique que le groupe est facultatif 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 facultatif. Un groupe entre accolades 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 de la syntaxe acontextuelle est choisie de façon à faciliter la description sémantique dans ce document et non pour faciliter un algorithme particulier d'analyse (par exemple, quelques ambiguïtés acontextuelles ont été introduites dans l'intérêt de la clarté). En cas d'ambiguïté, se référer à la catégorie sémantique des éléments syntaxiques.

2.1.2 Description sémantique

Pour chaque catégorie syntaxique (symbole non terminal), la description sémantique est donnée dans les 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 la section 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.

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 D. Pour chaque exemple, on indique via quelle règle de syntaxe il est produit et dans quel exemple il a été pris.

Ainsi, 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** de CHILL (voir l'Appendice C). Ces représentations textuelles 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 A). L'alphabet est représenté par la catégorie syntaxique < caractère > à partir de laquelle tout caractère de l'ensemble de caractères CHILL peut être obtenu comme production terminale.

Les éléments lexicaux de CHILL sont:

- les symboles spéciaux,
- les noms,
- les littéraux.

En plus des éléments lexicaux, il existe aussi des combinaisons de caractères spéciaux. La liste des combinaisons de symboles et de caractères spéciaux figure à l'Appendice B.

Les représentations textuelles de noms simples sont formés d'après la syntaxe suivante:

syntaxe:

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

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

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

sémantique:

Le caractère souligné (_) fait partie de la représentation textuelle de nom simple, c'est-à-dire que la représentation textuelle de nom simple *life_time* est différente de la représentation textuelle de nom simple *lifetime*. Lettres majuscules et minuscules sont différentes, par exemple, *Status* et *status* sont deux représentations textuelles de nom simple différentes.

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

Les représentations textuelles de nom simple **spéciales** dans un fragment doivent être soit entièrement en représentation majuscule soit entièrement en représentation minuscule. Les représentations textuelles 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 *représentation textuelle de nom* simple ne peut pas être une des représentations textuelles de noms simple **réservées** (voir l'Appendice C.1).

2.3 ESPACEMENTS

Un espace délimite tout élément lexical ou toute combinaison de caractères spéciaux. Les éléments lexicaux sont également terminés par le premier caractère qui ne peut pas en faire partie. Par exemple, *IFBTHEN* sera considéré comme *représentation textuelle 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)
{ *<caractère>* }*	(4.1)

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 occurence de la séquence spéciale */. Un **commentaire de fin de ligne** se termine par la première occurence de fin de ligne.

exemples:

4.1 /* *from collected algorithms from CACM no.93* */ (2.1)

2.5 COMMANDES DE MISE EN PAGE

Les commandes de mise en page BS (retour arrière), CR (retour de chariot), FF (présentation de forme), HT (tabulation horizontale), LF (interligne) et VT (tabulation verticale) de l'ensemble de caractères (voir l'Appendice A, FE_0 à FE_5) ne sont pas mentionnées dans la description de la syntaxe acontextuelle de 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 de CHILL, si et seulement si il est correct sans ces directives.

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

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 d'identification défini par l'implémentation et qui n'influence pas les règles d'identification de CHILL (voir la section 12.2).

2.7 NOMS ET LEURS DÉFINITIONS

syntaxe:

<nom> ::=	(1)
<représentation textuelle de nom>	(1.1)
<représentation textuelle de nom ::=	(2)
<représentation textuelle de nom simple>	(2.1)
| *<représentation textuelle de nom préfixe>*	(2.2)
<représentation textuelle de nom préfixe> ::=	(3)
<préfixe> ! *<représentation textuelle de nom simple>*	(3.1)
<préfixe> ::=	(4)
<préfixe simple> { ! *<préfixe simple>* }*	(4.1)
<préfixe simple> ::=	(5)
<représentation textuelle de nom simple>	(5.1)
<définition> ::=	(6)
<représentation textuelle de nom simple>	(6.1)
<liste de définitions> ::=	(7)
<définitions> {, *<définitions>* }*	(7.1)
<nom de champ> ::=	(8)
<représentation textuelle de nom simple>	(8.1)
<définition de nom de champ> ::=	(9)
<représentation textuelle de nom simple>	(9.1)
<liste de définitions de noms de champ> ::=	(10)
<définition de nom de champ> {, *<définition de nom de champ>* }*	(10.1)
<nom d'exception> ::=	(11)
<représentation textuelle de nom simple>	(11.1)
| *<représentation textuelle de nom préfixe>*	(11.2)
<nom de repère de texte> ::=	(12)
<représentation textuelle de nom simple>	(12.1)
| *<représentation textuelle de nom préfixe>*	(12.2)

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 d'identification de la section 12.2 donnent les *définitions* (formellement: occurrences de productions terminales de *définition*) auxquelles ce (cette occurrence de) *nom* est **identifié.** Ainsi, le *nom* désigne l'objet défini ou déclaré par les *définitions.* (Il ne peut y avoir plus d'une *définition* pour un *nom* que dans le cas de *noms d'éléments d'ensemble* ou de *noms* avec **quasi**-*définitions.*) On dit des *définitions* qu'elles définissent le *nom.* On dit qu'un nom est une application du nom créé par la définition à laquelle il est **identifié.** Le nom a sa représentation textuelle de nom simple le plus à droite égale à celle du nom.

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

Les *noms d'exception* sont utilisés pour identifier des filets d'exceptions selon les règles énoncées au chapitre 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 dans la section 10.10.1.

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

définition de notation :

Soit une *représentation textuelle 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 représentation textuelle de nom rattachée à la *représentation textuelle 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 *représentation textuelle de nom simple* est attachée une représentation textuelle de nom **canonique** qui est la *représentation textuelle de nom simple* proprement dite. A une *représentation textuelle de nom* est attachée une représentation textuelle de **nom canonique** définie comme suit:

- si la *représentation textuelle de nom* est une *représentation textuelle de nom simple,* c'est la représentation textuelle de nom canonique de cette *représentation textuelle de nom simple*;
- si la *représentation textuelle de nom* est une *représentation textuelle de nom préfixe,* la concaténation reste dans l'ordre juste de toutes les *représentations textuelles de nom simple* de la *représentation textuelle 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 ce document:

- la représentation textuelle de nom d'un *nom, nom d'exception,* ou le *nom de texte de référence,* est utilisée pour désigner la représentation textuelle du nom **canonique** de la *représentation textuelle de nom* du *nom,* du *nom d'exception,* ou du *nom de repérage de texte,* respectivement;
- la représentation textuelle de nom d'une *définition,* d'un *nom de champ* ou d'une *définition de nom de champ* sert à désigner la représentation textuelle de nom **canonique** de la *représentation textuelle de nom simple* dans cette *définition,* ce *nom de champ* ou cette *définition de nom de champ,* respectivement.

Les règles d'identification sont telles que:

- les *noms* appartenant à une *représentation textuelle de nom simple* sont **identifiés** aux *définitions* ayant la même *représentation textuelle*;
- les *noms* appartenant à une *représentation textuelle de nom préfixée* sont **identifiés** aux *définitions* d'une *représentation textuelle de nom* identique à la *représentation textuelle de nom simple,* la plus à droite, de la *représentation textuelle de nom préfixée* du *nom*;
- les *noms de champ* sont **identifiés** à des *définitions de nom de champ* ayant la même *représentation textuelle de nom* que les *noms de champ.*

Un nom hérite toutes les propriétés statiques liées au nom défini par la *définition* à laquelle il est identifié. Un *nom de champ* a les propriétés statiques liées à la *définition de nom de champ* à laquelle il est **identifié.**

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 une information de mode est nécessaire.

3.1.1 Modes

CHILL a des modes statiques (c.-à-d. des modes dont on peut déterminer statiquement toutes les propriétés), et des modes dynamiques (c.-à-d. 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.*

Dans le présent document, les noms de **mode** virtuels sont utilisés pour décrire des modes qui ne sont pas explicitement indiqués dans le texte du programme. Dans ce cas, le nom de **mode** est précédé par le caractère perluète (&).

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 repère.
- La classe **nulle**.
- La classe **toute**.

Les deux dernières sont des classes constantes, c.-à-d. 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 repère, 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 dans la section 12.1):

- Un mode a une **nouveauté** (définie aux sections 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 repérer**.
- Un mode peut avoir la **propriété de marquage et de paramétrage**.
- Un mode peut avoir la **propriété de non-valeur**.

En CHILL, des classes peuvent avoir les propriétés suivantes (définies dans la section 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 dans la section 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 de définitions> = *<mode définissant>*	(1.1)
<mode définissant> ::=	(2)
<mode>	(2.1)

syntaxe dérivée:

Une *définition de mode* où la *liste de définitions* comporte plus d'une *définition* est dérivée de plusieurs définitions de mode, une pour chaque *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 neumode. Une définition de synmode est **synonyme** de son mode définissant. Une définition de neumode 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 la section 12.1).

propriétés statiques:

Dans une *définition de mode* une *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 la section 3.4.2) 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 la section 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 ensemble de définitions récursives doit être un ensemble de définitions de modes récursives; voir la section 5.1.)

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 repère*, 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 INT)*	(1.1)

3.2.2 Définitions de synmodes

syntaxe:

<énoncé de définition de synmode> ::= (1)

SYNMODE *<définition de mode>* {, *<définition de mode>* }*; (1.1)

sémantique:

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

propriétés statiques:

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

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

Si le mode **définissant** est un mode intervalle, le mode **parent** du nom **synonyme** est alors celui de son mode **définissant**. Si le mode **définissant** est un mode chaîne **variable**, le mode **composant** du nom **synonyme** est alors celui de son mode **définissant**.

exemples:

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

3.2.3 Définitions de neumodes

syntaxe:

<énoncé de définition de neumode> ::= (1)

NEWMODE *<définition de mode>* {, *<définition de mode>* }*; (1.1)

sémantique:

Les énoncés de définition de neumode définissent des noms de **modes** qui ne sont pas **synonymes** de leur mode définissant.

propriétés statiques:

Une *définition* apparaissant dans une *définition de mode* apparaissant dans un *énoncé de définition de neumode* définit un nom de **neumode** (qui est aussi un nom de **mode**).

La **nouveauté** du nom de **neumode** est la *définition* qui le définit. Si le **mode définissant** du nom de **neumode** est un mode intervalle, le mode virtuel *&nom* est introduit comme le mode **parent** du nom de **neumode.** Le **mode définissant** de *&nom* est le mode **parent** du mode intervalle et la **nouveauté** de *&nom* est celle du nom de **neumode.**

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

Si la 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 composé>*	(1.1)
| [**READ**] *<mode composé>*	(1.2)
<mode non composé> ::=	(2)
<mode discret>	(2.1)
| *<mode ensembliste>*	(2.2)
| *<mode repère>*	(2.3)
| *<mode procédure>*	(2.4)
| *<mode exemplaire>*	(2.5)
| *<mode de synchronisation>*	(2.6)
| *<mode d'entrée-sortie>*	(2.7)
| *<mode de temporisation>*	(2.8)

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 neumode 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 rangée **paramétré**, un mode chaîne **paramétré** ou un mode structure **paramétré** dans lequel le nom de mode rangée **originel**, le nom de mode chaîne **originel** 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 rangée **protégé** (voir la section 3.12.3);
  - c'est un mode **champ** d'un mode structure **protégé** ou le mode d'un **champ marqueur** d'un mode structure **paramétré** (voir la section 3.12.4).

Un *mode* a les mêmes propriétés que le mode *non composé* ou *composé* qu'il contient. Dans les sections 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 dans la section 3.2. Les modes **protégés** ont les mêmes propriétés que leurs modes correspondants non **protégés**, à l'exception de la **propriété** de protection (voir la section 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 un *énoncé de définition de neumode*. 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 **originel**, de son mode rangée **originel** ou de son mode structure **variable originel**, respectivement;
  - si c'est un mode rangée, 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 sections 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>*	(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 la section 5.3). Une implémentation peut définir d'autres modes entiers de bornes différentes (par exemple, *LONG_INT, SHORT_INT*, ...) qui peuvent aussi être utilisés comme modes **parents** d'intervalles (voir la section 13.2). La représentation interne d'une valeur entière est la valeur entière elle-même.

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.**

exemples:

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 la section 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.

exemples:

5.4 *BOOL* (1.1)

3.4.4 Modes caractère

syntaxe:

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

noms prédéfinis:

Le nom CHAR est prédéfini comme nom 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 A). Cet alphabet définit 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 par *CHAR*.
- Le **nombre de valeurs** définies par un mode caractère est 256.

exemples:

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)
<définition> = *<expression littérale entière>*	(4.1)
<extension d'ensemble sans numéros> ::=	(5)
<élément d'ensemble> {, *<élément d'ensemble>* }*	(5.1)
<élément d'ensemble> ::=	(6)
<définition>	(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 *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.

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 littérale entière* 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 littérales entières e1* et *e2* dans l'*extension d'ensemble NUM (e1)* et *NUM (e2)* 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

syntaxe:

<mode intervalle> ::=	(1)
<nom de mode discret> (<intervalle littéral>)	(1.1)
| **RANGE** *(<intervalle littéral>)*	(1.2)
| *BIN (<expression littérale entière>)*	(1.3)
| *<nom de mode intervalle>*	(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 *INT* $(0 : 2^n - 1)$, par exemple, **BIN** *(2+1)* tient lieu de *INT (0 : 7).*

sémantique:

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

- Si le mode intervalle est de forme:
  <nom de mode discret> (<intervalle littéral>)
  alors, si le *nom de mode discret* n'est pas un mode intervalle, 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 forme:
  RANGE *(<intervalle littéral>),*
  alors le mode **parent** est le mode **racine** de la **classe résultante** des classes de la *borne supérieure* et de la *borne inférieure* de l'*intervalle littéral.*
- Si le mode intervalle est un nom de **mode intervalle** qui est un *nom* de **synmode**, alors son mode **parent** est le mode *parent* du mode **définissant** du *nom* de **synmode**. Sinon, c'est un nom de **neumode** et son mode **parent** est le mode **parent** introduit virtuellement (voir la section 3.2.3).

Un mode intervalle a les propriétés héréditaires suivantes:

- Une **borne inférieure** et une **borne supérieure** 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.
- 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 littérale entière* dans le cas de **BIN** doit rendre une valeur non négative.

exemples:

9.5	*INT (2:max)*	(1.1)
11.12	*line*	(1.4)

3.5 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 la section 5.3).

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)

3.6 MODES REPÈRE

3.6.1 Généralités

syntaxe:

<mode repère> ::= (1)
  <mode repère lié> (1.1)
  | *<mode repère libre>* (1.2)
  | *<mode descripteur>* (1.3)

sémantique:

Un mode repère définit des repères (adresses ou descripteurs) de locus **repérables**. Par définition, les repères liés repèrent des locus d'un mode statique donné; les repères libres peuvent repérer des locus de n'importe quel mode statique; les descripteurs repèrent des locus de mode dynamique.

L'opération de dérepérage est définie sur les valeurs repère (voir les sections 4.2.3, 4.2.4 et 4.2.5), rendant le locus qui est repéré.

Deux valeurs repère sont égales si et seulement si toutes deux, soit repèrent le même locus, soit ne repèrent aucun locus (c.-à-d. sont la valeur *NULL*).

3.6.2 Modes repère lié

syntaxe:

<mode repère lié> ::= (1)
  REF *<mode repéré>* (1.1)
  | *<nom de mode repère lié>* (1.2)

<mode repéré> ::= (2)
  <mode> (2.1)

sémantique:

Les repères liés définissent des valeurs repère de locus du mode repéré spécifié.

propriétés statiques:

Un mode repère lié a la propriété héréditaire suivante:

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

exemples:

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

3.6.3 Modes repère libre

syntaxe:

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

noms prédéfinis:

Le nom PTR est prédéfini comme nom de **mode repère libre.**

sémantique:

Un mode repère libre définit des valeurs repère de locus de tout mode statique.

exemples:

19.8 *PTR* (1.1)

3.6.4 Modes descripteur

syntaxe:

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

sémantique:

Un mode descripteur définit des valeurs repère de locus de mode dynamique (qui sont des locus d'un mode paramétré aux paramètres inconnus statiquement).

Une valeur descripteur peut repérer:

- des locus **chaîne de longueur** inconnue statiquement,
- des locus rangée à **borne supérieure** inconnue statiquement,
- des locus structure paramétrée dont les paramètres sont inconnus statiquement.

propriétés statiques:

Un mode descripteur a la propriété héréditaire suivante:

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

condition statique:

Le *mode structure variable* doit être **paramétrable**.

exemples:

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

3.7 MODES PROCÉDURE

syntaxe:

<mode procédure> ::= (1)
  PROC *(* [*<liste de paramètres>*] *)* [*<spec de résultat>*]
  [**EXCEPTIONS** *(* *<liste d'exceptions>* *)*] [**RECURSIVE**] (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.-à-d. 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.-à-d. 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 la section 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.-à-d. 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.*
- Il a une **récursivité** qui est **récursive** si **RECURSIVE** est spécifié. Dans le cas contraire, une option par défaut, définie par l'implémentation, spécifie soit **récursive** soit **non récursive.**

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.**

3.8 MODES EXEMPLAIRE

syntaxe:

<mode exemplaire> ::=	(1)
<nom de mode exemplaire>	(1.1)

noms prédéfinis:

Le nom INSTANCE est prédéfini comme nom de **mode exemplaire.**

sémantique:

Un mode exemplaire définit des valeurs qui identifient des processus. La création d'un nouveau processus (voir les sections 5.2.14, 6.13 et 11.1) produit une valeur exemplaire unique comme identification pour le processus créé.

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

exemples:

15.39	*INSTANCE*	(1.1)

3.9 MODES DE SYNCHRONISATION

3.9.1 Généralités

syntaxe:

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

sémantique:

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

3.9.2 Modes événement

syntaxe:

<mode événement> ::=	(1)
EVENT [*(<longueur d'événement>)*]	(1.1)
| *<nom de mode événement>*	(1.2)
<longueur d'événement> ::=	(2)
<expression littérale entière>	(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 sections 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.

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.

exemples:

14.10	**EVENT**	(1.1)

3.9.3 Modes tampon

syntaxe:

<mode tampon> ::=	(1)
BUFFER [*(<longueur de tampon>)*] *<mode des éléments de tampon>*	(1.1)
| *<nom de mode tampon>*	(1.2)
<longueur de tampon> ::=	(2)
<expression littérale entière>	(2.1)
<mode des éléments de 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, l'action recevoir et choisir et l'expression recevoir, décrites respectivement aux sections 6.18, 6.19 et 5.3.9.

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.

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 des **éléments de tampon** qui est le *mode des éléments de tampon.*

conditions statiques:

La *longueur de tampon* doit rendre une valeur non négative.

Le *mode des éléments de tampon* ne doit pas avoir la **propriété de non-valeur**.

exemples:

16.30 **BUFFER** (1) *user_messages* (1.1)
16.34 *user_buffers* (1.2)

3.10 MODES D'ENTRÉE-SORTIE

3.10.1 Généralités

syntaxe:

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

sémantique:

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

exemples:

20.17 *ASSOCIATION* (1.1)

3.10.2 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 CHILL cette relation est appelée une association; des associations peuvent être créées par l'opération prédéfinie *ASSOCIATE* et terminées par *DISSOCIATE.*

3.10.3 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 de littéral>* (3.2)

syntaxe dérivée:

La notation de mode d'indice *intervalle de littéral* est tirée du mode discret **RANGE** (*intervalle de 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 d'indice*; ce mode d'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 d'indice* n'est spécifié, les enregistrements ne peuvent être transférés qu'en séquence.

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 **d'indice** facultatif, qui est le *mode d'indice*.

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 marqueurs**.

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

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)

3.10.4 Modes texte

syntaxe:

<mode texte> ::=	(1)
TEXT *(<longueur de texte>)* [*<mode d'indice>*] [**DYNAMIC**]	(1.1)
<longueur de texte> ::=	(2)
<expression de littéral entier>	(2.1)

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 **d'indice** qui a la même signification que pour les modes accès.

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**.
- Il a un mode **accès** qui est **ACCESS** [(*mode d'indice*)] **CHARS** (<*longueur de texte*>) **[DYNAMIC]** (<*mode d'indice*> et **DYNAMIC** font partie du mode seulement s'ils sont spécifiés).

exemples:

26.8 **TEXT (80) DYNAMIC** (1.1)

3.11 MODES TEMPORISATION

3.11.1 Généralités

syntaxe:

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

sémantique:

Le mode temporisation est utilisé pour la surveillance des processus décrits au chapitre 9. Les valeurs de temporisation sont créées par un ensemble d'opérations prédéfinies. Les opérateurs relationnels sont définis sur les valeurs de temporisation.

3.11.2 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.

3.11.3 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.

3.12 MODES COMPOSÉS

3.12.1 Généralités

syntaxe :

<mode composé> ::=	(1)
<mode chaîne>	(1.1)
| *<mode rangée>*	(1.2)
| *<mode structure>*	(1.3)

sémantique :

Un mode composé définit des valeurs composées, c'est-à-dire des valeurs constituées par des sous-composantes auxquelles on peut avoir accès ou qu'on peut obtenir (voir les sections 4.2.6-4.2.10 et 5.2.6-5.2.10).

3.12.2 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 originel> (<longueur de chaîne>)	(2.1)
| *<nom de mode chaîne paramétré>*	(2.2)
<nom de mode chaîne originel> ::=	(3)
<nom de mode chaîne>	(3.1)
<type de chaîne> ::=	(4)
BOOLS	(4.1)
| **CHARS**	(4.2)
<longueur de chaîne> ::=	(5)
<expression littérale entière>	(5.1)

sémantique :

Un mode chaîne **fixe** définit des valeurs chaîne de bits ou chaîne de caractères de longueur indiquée ou impliquée par le mode chaîne. Un mode chaîne **variable** définit des valeurs chaîne de bits 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 de bits 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 bien 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 $0 \leqslant i < l$. Une chaîne s précède t quand:

- il existe un indice j tel que $s(j) < t(j)$ et $s(0 : j - 1) = t(0 : j - 1)$ ou
- *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 la section 5.3).

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.
- C'est un mode chaîne de **bits** ou un mode chaîne de **caractères**, selon que le *type de chaîne* spécifie **BOOLS** ou **CHARS**, ou selon que le *nom de mode chaîne originel* est un mode chaîne de **bits** ou de **caractères.**
- C'est un mode chaîne **variable** si **VARYING** est spécifié ou si le *nom de mode chaîne originel* 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 **originel** qui est le mode désigné par le *nom de mode chaîne originel*.

Un mode chaîne **variable** a la propriété non héréditaire suivante: il a un mode **composé**, 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 originel> (<longueur de chaîne>)
  le mode **composé** est *&nom (longueur de chaîne)*, où *&nom* est un nom de **synmode** introduit virtuellement **synonyme** du mode **composant** du *nom de mode chaîne variable originel*.
- Si le mode chaîne **variable** est un *nom de mode chaîne* qui est un nom de **synmode**, son mode **composé** est celui du mode définissant du nom de **synmode**; sinon, c'est un nom de **neumode** et son mode **composant** est le mode **composant** virtuellement introduit (voir la section 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 originel*. Cette condition ne s'applique qu'aux modes chaîne **paramétrés** qui ne sont pas introduits virtuellement.

exemples:

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

3.12.3 Modes rangée

syntaxe:

<mode rangée> ::=	(1)
ARRAY *(<mode d'indice> {, <mode d'indice> }*)*	
*<mode des éléments> { <implantation d'élément> }**	(1.1)
| *<mode rangée paramétré>*	(1.2)
| *<nom de mode rangée>*	(1.3)
<mode rangée paramétré> ::=	(2)
<nom de mode rangée originel>(<indice supérieur>)	(2.1)
| *<nom de mode rangée paramétré>*	(2.2)
<nom de mode rangée originel> ::=	(3)
<nom de mode rangée>	(3.1)
<indice supérieur> ::=	(4)
<expression littérale discrète>	(4.1)
<mode des éléments> ::=	(5)
<mode>	(5.1)

syntaxe dérivée:

Un *mode rangée* avec plus d'un mode d'indice (dénotant une rangée multidimensionnelle), est une syntaxe dérivée pour un *mode rangée* avec un *mode des éléments* qui est lui-même un *mode rangée.* 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 d'indice.* Dans ce cas, l'*implantation d'élément* la plus à gauche est associée au *mode des éléments* le plus interne, etc.

sémantique:

Un mode rangée définit des valeurs composées, qui sont des listes de valeurs définies par son mode des éléments. L'implantation physique d'un locus ou d'une valeur rangée peut être contrôlée par une spécification d'*implantation d'élément* (voir la section 3.12.5). Deux valeurs rangée sont égales si et seulement si toutes les valeurs élément correspondantes sont égales.

propriétés statiques:

Un mode rangée a les propriétés héréditaires suivantes:

- Il a un mode **d'indice** qui est le *mode d'indice* dans le cas où il ne s'agit pas d'un *mode rangée paramétré,* sinon le mode **d'indice** est le mode intervalle construit comme:
  &nom (borne inférieure : borne supérieure),
  où *&nom* est un nom de **synmode** virtuel **synonyme** du mode **d'indice** du *nom de mode rangée originel, borne inférieure* est la *borne inférieure* du mode **d'indice** du *nom de mode rangée originel,* 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 **d'indice.**
- Il a un mode **des éléments** qui est soit *M* soit **READ** *M,* où *M* est le *mode des éléments* ou le mode **des éléments** du *nom de mode rangée originel,* selon le cas. Le mode **des éléments** est **READ** *M* si et seulement si *M* n'est pas un mode **protégé** et que le *mode rangée* est un mode **protégé.** Le mode **des éléments** est un mode **protégé** implicitement s'il est **READ** *M.*
- Il a une **implantation d'élément** qui, s'il est un *mode rangée paramétré,* est l'**implantation d'élément** de son *nom de mode rangée originel* 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 **d'indice.**
- C'est un mode **implanté** si et seulement si une *implantation d'élément* est spécifiée et s'il s'agit d'un *pas.*

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

Un mode rangée **paramétré** a un mode rangée **originel** qui est le mode désigné par le *nom de mode rangée originel.*

conditions statiques:

La classe de l'*indice supérieur* doit être **compatible** avec le mode **d'indice** du *nom de mode rangée originel* et la valeur qu'il rend doit se trouver dans l'intervalle défini par ce mode **d'indice.**

exemples:

5.29	**ARRAY** *(1:16)* **STRUCT** *(c4, c2, c1 BOOL)*	(1.1)
11.12	**ARRAY** *(line)* **ARRAY** *(column) square*	(1.1)
11.17	*board*	(1.3)

3.12.4 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)
| <*choix de champ*> (2.2)

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

<*choix de champs*> ::= (4)
CASE [<*liste de marqueurs*>] **OF**
<*champ à choisir*> {, <*champ à choisir*> }*
[**ELSE** [<*champ récurrent*> {, <*champ récurrent*> }*]] **ESAC** (4.1)

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

<*liste de marqueurs*> ::= (6)
<*nom de champ marqueur*> {, <*nom de champ marqueur*> }* (6.1)

<*champ récurrent*> ::= (7)
<*liste 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*> {, <*expressionlittérale discrète*> }* (10.1)

syntaxe dérivée :

Une occurrence de *champ fixe*, ou une occurrence de *champ récurrent*, où la *liste de définitions de noms de champ* comporte plus d'une *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 *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 composées 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 (composés) 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.-à-d. de champs qui sont toujours présents et auxquels on peut accéder sans aucun contrôle dynamique.

Les structures **variables** ont des champs récurrents, c.-à-d. des champs qui ne sont pas toujours présents. Pour les structures **variables avec marqueurs**, la présence de ces champs est connue seulement à l'exécution d'après la ou les valeurs de certains champs fixes associés, nommés champs **marqueurs**. Les structures **variables sans marqueurs** n'ont pas de champs **marqueurs**. 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 à choisir (pire des cas).

Dans un *choix de champs*, le *choix variable* 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, le *choix variable* qui suit ELSE (qui sera présent) est choisi.

Une structure **paramétrée** est déterminée par un mode structure **variable** pour lequel le choix de champs à choisir 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 **marqueurs**, 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 à la section 3.13.4.

L'implantation d'un locus ou d'une valeur structure peut être contrôlée au moyen d'une spécification d'implantation de champs (voir la section 3.12.5).

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

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 *choix de champs.*
- C'est un mode structure **variable** si et seulement s'il est un *mode structure* contenant au moins une occurrence de *choix de champs.*
- 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 de noms* 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 marqueur** 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 **implanté** 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 *liste de noms* 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 de définitions de noms de champ* dans les *champs fixes* et de l'ensemble des noms définis par toute *liste de définitions de noms de champ* dans les *choix de champs.* Les noms **de champ** définis par une *liste 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 marqueur** si et seulement s'il apparaît dans une des listes de *marqueurs* d'un *choix de champs.* Les *choix de champs* dans lesquels aucune liste de *marqueurs* n'est spécifiée, sont des choix de champs **sans marqueurs**.

- Un mode structure **variable** est un mode structure **variable sans marqueurs** si toutes ses occurrences de *choix de champs* sont **sans marqueurs**. Sinon, c'est un mode structure **variable avec marqueurs.**
- Un mode structure **variable** est un mode structure **variable paramétrable** s'il est soit un mode structure **variable avec marqueurs**, soit un mode structure **variable sans marqueurs** dans lequel, pour chaque occurrence de *choix de champs*, une *spécification d'étiquettes de cas* est donnée pour toutes les occurrences de *champs à choisir* 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 marqueurs**, la liste des M_i-classes par valeur, où les M_i représentent les modes des noms **de champ marqueur** dans l'ordre où ils sont définis dans les *champs fixes*;
  - si c'est un mode structure **variable sans marqueurs**, la liste est construite à partir **des listes** individuelles **résultantes des classes** de chaque *choix de champs* en les concaténant dans l'ordre où les *choix de champs* apparaissent. La **liste résultante des classes** d'une occurrence de *choix de champs* est la **liste résultante des classes** de la liste d'occurrences de *spécification d'étiquettes de cas* qu'elle contient (voir la section 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 de *champs à choisir* sélectionnées par la liste de valeurs définies par la *liste d'expressions littérales.*

  L'ensemble des noms de **champ marqueur** d'un *mode structure paramétré* est l'ensemble des noms de **champ marqueur** 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 marqueurs** si son mode structure **variable originel** est un mode structure **variable avec marqueurs**, sinon le mode structure **paramétré** est **sans marqueurs.**

  Pour les modes structure **paramétrés** dynamiques, voir la section 3.13.4.

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

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

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

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

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

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)

3.12.5 Description d'implantation pour modes rangée 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 littérale entière> (5.1)

<taille de pas> ::= (6)
  <expression littérale entière> (6.1)

<bit initial> ::= (7)
  <expression littérale entière> (7.1)

<bit final> ::= (8)
  <expression littérale entière> (8.1)

<longueur> ::= (9)
  <expression littérale entière> (9.1)

sémantique :

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

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

L'information **PACK, NOPACK** ne s'applique qu'à un niveau, c.-à-d. elle ne s'applique qu'aux éléments de la rangée ou aux champs de la structure, mais pas aux composants possibles des éléments de la rangée 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 rangée ou d'une structure en spécifiant une information de position pour ses composants dans le mode. Cette information de position est donnée de la façon suivante:

- Pour les modes rangée, l'information de position est donnée pour tous les éléments en même temps, sous la forme d'un *pas* suivant le mode rangée.
- 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 représentation 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 littérale entière.*

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 rangée et

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

Le $j$ème élément de la rangée commence à un décalage de bits de $p + b_j - 1$ à partir du début de chaque locus du mode rangée, où p est le décalage de bits spécifié dans *pos.* Chaque élément occupe la longueur donnée dans *pos.*

Défauts

La notation:

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

est sémantiquement équivalente à:

POS *(<numéro de mot> , <bit initial> , NUM (bit final) − NUM (bit initial) + 1)*

La notation:

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

est sémantiquement équivalente à:

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

où *BTAILLE* est le nombre minimal de bits nécessaire à représenter le composant pour lequel le *pos* est spécifié.

La notation:

POS *(<numéro de mot>)*

est sémantiquement équivalente à:

POS *(<numéro de 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 rangée implanté, l'implantation d'élément du mode détermine la repérabilité de ses sous-locus (y compris les sous-rangées et tranches de rangée) comme suit:

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

Pour tout locus d'un mode structure implanté, la repérabilité 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 **repérable** si l'implantation de champ est **NOPACK**.

conditions statiques:

Si le mode **des éléments** d'un mode rangée donné, ou le mode **de champ** d'un nom de **champ** d'un mode structure donné, est lui-même un mode rangée ou structure, ce doit être un mode **implanté** si le mode rangée ou structure donné est un mode **implanté**.

Chaque *mot*, *bit initial*, *bit final*, *longueur* et *taille* de pas doit, s'il est spécifié, donner une valeur non négative; les valeurs données par *bit initial* et *bit final* doivent être inférieures à *WIDTH*, le nombre de bits d'un mot de l'implémentation; la valeur donnée par le *bit initial* doit être inférieure ou égale à celle du *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 rangée, 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 composants de champ ou de rangée associés.

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

Cohérence et faisabilité

Cohérence:

Aucun composant d'une structure ne peut se voir imposer d'occuper un bit déjà occupé par un autre composant du même objet, sauf dans le cas de deux noms **de champ récurrent** définis dans le même *choix de champ*; cependant, dans ce dernier cas, les noms **de champ récurrent** ne peuvent être tous deux définis dans le même *champ à choisir*, 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 repérabilité d'un sous-locus de tout locus (**repérable** ou non) est déterminée seulement par l'implantation (de champ ou d'élément), ce qui est une propriété du mode du locus. Ceci restreint la représentation de composants qui ont eux-mêmes des composants **repérables**.

exemples:

17.5	**PACK**	(1.1)
19.14	**POS** *(1,0:15)*	(4.2)

3.13 MODES DYNAMIQUES

3.13.1 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 ce document. Ces notations virtuelles sont précédées du caractère perluète (&) afin de les distinguer des notations qui peuvent effectivement apparaître dans un texte de programme en CHILL.

3.13.2 Modes chaîne dynamiques

dénotation virtuelle:

& *<nom de mode chaîne originel> (<expression entière>)*

sémantique:

Un mode chaîne dynamique est un mode chaîne paramétré de longueur inconnue statiquement.

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.

3.13.3 Modes rangée dynamiques

dénotation virtuelle:

& *<nom de mode rangée originel> (<expression discrète>)*

sémantique:

Un mode rangée dynamique est un mode rangée paramétré de **borne supérieure** inconnue statiquement.

propriétés statiques:

Les modes rangée dynamique ont les mêmes propriétés que les modes rangée, sauf en ce qui concerne les propriétés ci-après.

propriétés dynamiques:

- A un mode rangée 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 rangée originel.*

3.13.4 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 inconnus statiquement.

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.*

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 de loc-identité>*	(2.2)

sémantique :

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

exemples :

6.9	DCL *jINT := 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 de définitions> *<mode>* [STATIC] [*<initialisation>*]	(1.1)
<initialisation> ::=	(2)
<initialisation domaniale>	(2.1)
| *<initialisation viagère>*	(2.2)
<initialisation domaniale> ::=	(3)
<symbole d'affectation> *<valeur>* [*<filet>*]	(3.1)
<initialisation viagère> ::=	(4)
INIT *<symbole d'affectation>* *<valeur constante>*	(4.1)

sémantique :

Une déclaration de locus crée autant de locus qu'on spécifie de définitions apparaissant dans la *liste 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 la section 10.2) et la valeur obtenue est affectée au(x) locus. Avant que la *valeur* ne soit évaluée, le(s) locus contient (contiennent) une valeur **indéfinie**.

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

Ne pas spécifier d'*initialisation* est équivalent, sémantiquement, à la spécification d'une *initialisation viagère* avec la valeur **indéfinie** (voir la section 5.3.1).

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

- **propriété de marquage et de paramétrage**: le(s) sous-locus de champ avec **marqueurs** créés sont initialisés avec la valeur de paramètre correspondante.
- **propriété de non-valeur :**
  - L'événement créé et/ou le(s) (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.
  - Le(s) (sous-)locus d'association créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne contiennent pas d'association.

- Le(s) (sous-)locus d'accès créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne sont pas connectés à une association.
- Le(s) (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 à la section 10.9 et au chapitre 8.

propriétés statiques:

Une *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 **repérable**.

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 **indéfinie**.

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 la section 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 la section 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 de loc-identité

syntaxe:

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

sémantique:

Une déclaration de loc-identité crée autant de noms d'accès au locus spécifié qu'il y a de *définitions* spécifiées dans la *liste 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 de loc-identité 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 les sections 10.2 et 10.9).

propriétés statiques:

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

Un nom de **loc-identité** est **repérable** si et seulement si le *locus* spécifié est **repérable**.

conditions statiques:

Si DYNAMIC est spécifié dans la *déclaration de loc-identité*, le *mode* doit être **paramétrable**. Le *mode* spécifié doit être **compatible en lecture dynamique** avec le mode du *locus* si **DYNAMIC** est spécifié et, dans les autres cas, **compatible en lecture** avec le mode du *locus*.

Le locus ne doit pas être un *élément de chaîne* ni une *tranche de chaîne* dans lequel le mode du *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.

exemples:

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)
| *<repère lié dérepéré>*	(1.2)
| *<repère libre dérepéré>*	(1.3)
| *<descripteur dérepéré>*	(1.4)
| *<élément de chaîne>*	(1.5)
| *<tranche de chaîne>*	(1.6)
| *<élément de rangée>*	(1.7)
| *<tranche de rangée>*	(1.8)
| *<champ de structure>*	(1.9)
| *<appel de procédure rendant locus>*	(1.10)
| *<appel d'opération prédéfinie rendant locus>*	(1.11)
| *<conversion de locus>*	(1.12)

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 la section 10.9).
- Il peut être **intrarégional** ou **extrarégional** (voir la section 11.2.2).
- Il peut être **repérable** ou non. La définition du langage exige que certains locus soient **repérables** et que d'autres ne le soient pas, comme indiqué dans les sections appropriées. Une implémentation peut étendre la repérabilité 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 de loc-identité>*	(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.-à-d. un nom déclaré explicitement dans une *déclaration de locus* ou déclaré implicitement dans un *paramètre formel* sans l'attribut **LOC**;
- un nom **de loc-identité**, c.-à-d. un nom déclaré explicitement dans une *déclaration de loc-identité* ou déclaré implicitement dans un *paramètre formel* avec l'attribut **LOC**;
- un nom **d'énumération de locus**, c.-à-d. un *compteur de boucle* dans une *énumération de locus*;
- un nom **de locus faire-avec**, c.-à-d. 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 marqueurs, 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 de loc-identité*, du *nom d'énumération de locus* ou du *nom de locus faire-avec.*

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

conditions dynamiques:

Quand on accède à un locus via un *nom de loc-identité*, il ne peut pas dénoter un locus **indéfini**.

En cas d'accès, via un nom *de loc-identité*, à un locus qui est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir la section 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	*cl*	(1.4)

4.2.3 Repères liés dérepérés

syntaxe:

<repère lié dérepéré> ::=	(1)
<valeur primitive repère lié> → [*<nom de mode>*]	(1.1)

sémantique:

Un repère lié dérepéré donne le locus qui est repéré par la valeur repère lié.

propriétés statiques:

Le mode attaché au *repère lié dérepéré* est le *nom de mode* s'il y en a un, sinon le mode **repéré** du mode de la *valeur primitive repère lié.* Un *repère lié dérepéré* est **repérable.**

conditions statiques:

La *valeur primitive repère lié* doit être **forte.** Si le *nom de mode* optionnel est spécifié, il doit être **compatible en lecture** avec le mode **repéré** du mode de la *valeur primitive repère lié.*

conditions dynamiques:

La durée de vie du locus repéré ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive repère lié* donne la valeur *NULL.*

Si le locus repéré est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir la section 4.2.10).

exemples:

10.54 *p* → (1.1)

4.2.4 Repères libres dérepérés

syntaxe:

<*repère libre dérepéré*> ::= (1)
<*valeur primitive repère libre*> → <*nom de mode*> (1.1)

sémantique:

Un repère libre dérepéré donne le locus qui est repéré par la valeur repère libre.

propriétés statiques:

Le mode attaché à un *repère libre dérepéré* est le *nom de mode.* Un *repère libre dérepéré* est **repérable**.

conditions statiques:

La *valeur primitive repère libre* doit être **forte.**

conditions dynamiques:

La durée de vie du locus dérepéré ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive repère libre* donne la valeur *NULL.*

Le *nom de mode* doit être **compatible en lecture** avec le mode du locus repéré.

Si le locus repéré est un champ **récurrent**, les conditions d'accès au champ récurrent pour le locus doivent être satisfaites (voir la section 4.2.10).

4.2.5 Descripteurs dérepérés

syntaxe:

<*descripteur dérepéré*> ::= (1)
<*valeur primitive descripteur*> → (1.1)

sémantique:

Un descripteur dérepéré donne le locus qui est repéré par la valeur descripteur.

propriétés statiques:

Le mode dynamique attaché à un *descripteur dérepéré* 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 **repéré originel** du mode de la *valeur primitive descripteur* et où les paramètres sont, selon le mode **repéré originel**:

- la **longueur de la chaîne** dynamique, dans le cas d'un mode chaîne;
- la **borne supérieure** dynamique, dans le cas d'un mode rangée;
- la liste des valeurs associées au mode du locus de structure paramétré, dans le cas d'un mode structure **variable.**

Un *descripteur dérepéré* est **repérable.**

conditions statiques:

La *valeur primitive descripteur* doit être **forte.**

conditions dynamiques:

La durée de vie du locus repéré ne doit pas être terminée.

L'exception *EMPTY* est causée si la *valeur primitive descripteur* donne *NULL.*

Si le locus repéré est un champ **récurrent**, les conditions d'accès au champ récurrent du locus doivent être satisfaites (voir la section 4.2.10).

exemples:

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 *BOOL* ou *CHAR,* selon que le mode du *locus chaîne* est un mode chaîne **de bits** ou un mode chaîne **de caractères.**

Si le mode du *locus chaîne* est un mode chaîne **variable,** l'*élément de chaîne* n'est pas **repérable.**

conditions dynamiques:

L'exception RANGEFAIL se produit si la relation suivante n'est pas vérifiée:

$0 \leq NUM$ *(élément de début)* $\leq L - 1$, où L est la longueur effective du *locus chaîne.*

exemples:

18.16 *string → (i)* (1.1)

4.2.7 Tranches de chaîne

syntaxe:

<tranche de chaîne> ::=	(1)
<locus chaîne> (<élément de gauche> : <élément de droite>)	(1.1)
| *<locus chaîne> (<élément de début>* **UP** *<taille de tranche>)*	(1.2)
<élément de gauche> ::=	(2)
<expression entière>	(2.1)
<élément de droite> ::=	(3)
<expression entière>	(3.1)
<taille de tranche> ::=	(4)
<expression entière>	(4.1)

sémantique:

Une tranche 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) de la tranche de chaîne est déterminée à partir des expressions spécifiées.

Une *tranche de chaîne* dans laquelle l'élément de *droite* fournit une valeur inférieure à celle fournie par l'*élément de gauche* ou dans laquelle la *taille de tranche* fournit une valeur non positive désigne une chaîne vide.

propriétés statiques:

Le mode (éventuellement dynamique) attaché à une *tranche 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 **composant** et dans lequel la *taille de chaîne* est soit

NUM (élément de droite) − NUM (élément de gauche) + 1

soit

NUM (taille de tranche).

Toutefois, si on dénote une chaîne vide, la *taille de chaîne* est 0. Le mode attaché à une *tranche de chaîne* est statique si la *taille de chaîne* est **littérale**, c.-à-d. si *l'élément de gauche* et *l'élément de droite* sont **littéraux** ou si la *taille de tranche* est **littérale**; sinon, le mode est dynamique.

Si le mode du *locus chaîne* est un mode chaîne **variable**, la *tranche de chaîne* n'est pas **repérable**.

conditions statiques:

Les relations suivantes sont valables:

0 ≤ NUM (élément de gauche) ≤ L − 1
0 ≤ NUM (élément de droite) ≤ L − 1
0 ≤ NUM (élément de début) ≤ L
NUM (élément de début)+ NUM (taille de tranche) ≤ 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 rangée

syntaxe:

<élément de rangée> ::=	(1)
<locus rangée> (<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 rangée* en syntaxe stricte n'a qu'une seule expression (d'indice).

sémantique:

Un élément de rangée donne un (sous-)locus qui est un élément du locus rangée spécifié indiqué par *expression.*

propriétés statiques:

Le mode attaché à l'*élément de rangée* est le mode **des éléments** du mode du *locus rangée.*

Un *élément de rangée* est **repérable** si **l'implantation d'élément** du mode du *locus rangée* est **NOPACK.**

conditions statiques:

La classe de l'*expression* doit être **compatible** avec le mode **d'indice** du mode du *locus rangée.*

conditions dynamiques:

L'exception *RANGEFAIL* est causée si la relation suivante ne se vérifie pas:

$L \leqslant expression \leqslant U$

où L et U sont respectivement la **borne inférieure** et la **borne supérieure** (éventuellement dynamique) du mode du *locus rangée.*

exemples:

11.36	*b(m.lin_1)(m.col_1)*	(1.1)

4.2.9 Tranches de rangée

syntaxe:

<tranche de rangée> ::=	(1)
<locus rangée> (<élément inférieur> : <élément supérieur>)	(1.1)
| *<locus rangée> (<premier élément>* **UP** *<taille de rangée>)*	(1.2)
<élément inférieur> ::=	(2)
<expression>	(2.1)
<élément supérieur> ::=	(3)
<expression>	(3.1)
<premier élément> ::=	(4)
<expression>	(4.1)

sémantique :

Une tranche de rangée donne un locus rangée (éventuellement dynamique) qui est la partie du locus rangée spécifié indiqué par *l'élément inférieur* et *l'élément supérieur* ou par le *premier élément* et la *taille de tranche.* La **borne inférieure** de la tranche de rangée est égale à la borne inférieure de la rangée spécifiée; la **borne supérieure** (éventuellement dynamique) est déterminée à partir des expressions spécifiées.

propriétés statiques :

Le mode (éventuellement dynamique) attaché à une *tranche de rangée* est un mode rangée **paramétré** formé comme suit:

&nom (indice supérieur)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) du *locus rangée* 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 tranche) – 1

où *L* est la **borne inférieure** du mode du *locus rangée.*

Le mode attaché à une *tranche de rangée* est statique si *l'indice supérieur* est **littéral**, c.-à-d. que *l'élément inférieur* et *l'élément supérieur* sont tous deux **littéraux**, ou si la *taille de tranche* est **littérale**; sinon, le mode est dynamique.

Une *tranche de rangée* est **repérable** si **l'implantation d'élément** du mode du *locus rangée* 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 **d'indice** du *locus rangée.*

Les relations suivantes doivent être vérifiées:

L ≤ élément inférieur ≤ élément supérieur ≤ U

1 ≤ NUM (taille de tranche) ≤ NUM (U) – NUM (L) + 1

NUM (L) ≤ NUM (premier élément) ≤ NUM (premier élément) + NUM (taille de tranche) – 1
≤ NUM (U)

où *L* et *U* sont respectivement la **borne inférieure** et la **borne supérieure** du mode du *locus rangée.* 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.

exemples :

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 marqueurs**, 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 **repérable** 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 du *locus structure.*

conditions dynamiques:

Un *locus* ne doit pas dénoter:

- un locus de mode structure **variable avec marqueurs** et la (les) valeur(s) du (des) champ(s) **marqueur(s)** associé(s) indique(nt) 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 (il faut noter que la condition n'inclut pas une exception). L'exception TAGFAIL se produit si elles ne sont pas satisfaites pour le *locus structure.*

exemples:

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 de la **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 **repérable** 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 d'opération prédéfinie rendant locus

syntaxe:

<appel d'opération prédéfinie rendant locus> ::= (1)
<appel d'opération prédéfinie rendant locus> (1.1)

sémantique:

L'appel d'opération prédéfinie rendant locus fournit le locus renvoyé par l'appel d'opération prédéfinie.

propriétés statiques :

Le mode qui s'attache à un *appel d'opération prédéfinie rendant locus* est le mode de la spec de **résultat** de *l'appel d'opération prédéfinie rendant locus.*

conditions dynamiques :

L'*appel d'opération 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 de 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 **repérable.**

conditions statiques :

Le *locus de mode statique* doit être **repérable.**

La relation suivante doit se vérifier:

SIZE (nom de mode) = SIZE (locus de mode statique)

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 de définitions> [*<mode>*] = *<valeur constante>* (2.1)

syntaxe dérivée:

Une *définition de synonyme*, où la *liste de définitions* comporte plus d'une définition, est dérivée de plusieurs occurrences de *définition de synonyme*, une pour chaque 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 *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 **indéfinie** (voir la section 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*.

Les définitions de synonyme ne doivent pas être récursives ni mutuellement récursives via d'autres définitions de synonyme ou définitions de mode, c.-à-d. qu'aucun ensemble de définitions récursives ne peut contenir de définition de synonyme (voir la section 3.2.1).

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 tranche de chaîne>* (1.6)
| *<valeur élément de rangée>* (1.7)
| *<valeur tranche de rangée>* (1.8)
| *<valeur champ de structure>* (1.9)
| *<conversion d'expression>* (1.10)

| *<appel de procédure rendant valeur>* (1.11)
| *<appel d'opération prédéfinie rendant valeur>* (1.12)
| *<expression démarrer>* (1.13)
| *<opérateur nullaire>* (1.14)
| *<expression parenthésée>* (1.15)

sémantique :

Une valeur primitive est le constituant de base d'une expression. Certaines valeurs primitives ont une classe dynamique, c.-à-d. 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**, un *appel d'opération 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 d'opération 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 **indéfinie.**

exemples :

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.-à-d. un nom défini dans un *énoncé de définition de synonyme*;
- un nom **d'énumération de valeur**, c.-à-d. un nom défini par un *compteur de boucle* dans une *énumération de valeur*;
- un nom **de valeur faire-avec**, c.-à-d. un nom de **champ** introduit comme nom de valeur dans *l'action faire* avec une *partie avec*;
- un nom **de valeur reçue**, c.-à-d. un nom introduit dans une *action recevoir et choisir*;
- un nom **de procédure générale** (voir la section 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 d'entier>	(1.1)
| *<littéral de booléen>*	(1.2)
| *<littéral de caractère>*	(1.3)
| *<littéral d'ensemble>*	(1.4)
| *<littéral de vide>*	(1.5)
| *<littéral de chaîne de caractères>*	(1.6)
| *<littéral de chaîne de bits>*	(1.7)

sémantique :

Un littéral donne une valeur **constante**.

propriétés statiques :

La classe du *littéral* est respectivement la classe du *littéral d'entier, littéral de booléen, . . .*, etc. Un *littéral* est **discret** si c'est un *littéral d'entier*, un *littéral de booléen*, un *littéral de caractère* ou un *littéral d'ensemble.*

La lettre suivie d'une apostrophe qui figure au début d'un *littéral d'entier*, d'un *littéral de booléen* et d'un *littéral de chaîne de bits* (c.-à-d. *B', D', H', O', b', d', h', o'*) est une qualification de littéral.

5.2.4.2 Littéraux d'entier

syntaxe :

<littéral d'entier> ::= (1)
  <littéral décimal d'entier> (1.1)
  | *<littéral binaire d'entier>* (1.2)
  | *<littéral octal d'entier>* (1.3)
  | *<littéral hexadécimal d'entier>* (1.4)

<littéral décimal d'entier> ::= (2)
  [{ *D* | *d* } '] { *<chiffre>* | _ } $^{+}$ (2.1)

<littéral binaire d'entier> ::= (3)
  { *B* | *b* } ' { 0 | 1 | _ } $^{+}$ (3.1)

<littéral octal d'entier> (4)
  { *O* | *o* } ' { *<chiffre octal>* | _ } $^{+}$ (4.1)

<littéral hexadécimal d'entier> ::= (5)
  { *H* | *h* } ' { *<chiffre hexadécimal>* | _ } $^{+}$ (5.1)

<chiffre hexadécimal> ::= (6)
  <chiffre> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* | *c* | *d* | *e* | *f* (6.1)

<chiffre octal> ::= (7)
  0 | *1* | *2* | *3* | *4* | *5* | *6* | *7* (7.1)

sémantique :

Un littéral d'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.-à-d. qu'il ne sert qu'à améliorer la lisibilité et qu'il n'a pas d'influence sur la valeur dénotée.

propriétés statiques :

La classe d'un *littéral d'entier* est la *INT*-classe par dérivation. Un *littéral d'entier* est **constant** et **littéral.**

conditions statiques :

Ni la chaîne qui suit l'apostrophe (') ni le *littéral d'entier* tout entier ne doivent consister seulement en caractères soulignés.

exemples :

6.11 *1_721_119* (1.1)
  D'1_721_119 (1.1)
  B'101011_110100 (1.2)
  O'53_64 (1.3)
  H'AF4 (1.4)

5.2.4.3 Littéraux de booléen

syntaxe :

<littéral de booléen> ::= (1)
  <nom de littéral de booléen> (1.1)

noms prédéfinis :

Les noms FALSE et TRUE sont prédéfinis comme noms de **littéral de booléen.**

sémantique :

Un littéral de booléen donne une valeur booléenne.

propriétés statiques :

La classe du *littéral de booléen* est la *BOOL*-classe par dérivation. Un *littéral de booléen* est **constant** et **littéral.**

exemples :

5.46 *FALSE* (1.1)

5.2.4.4 Littéraux de caractère

syntaxe :

<littéral de caractère> ::= (1)
' *<caractère>* | *<séquence de contrôle>* ' (1.1)

sémantique :

Un littéral de caractère fournit une valeur de caractère. Indépendamment de la représentation imprimable, la représentation *séquence de contrôle* peut être utilisée.

propriétés statiques :

La classe d'un *littéral de caractère* est la *CHAR*-classe par dérivation. Un *littéral de caractère* est **constant** et **littéral.**

conditions statiques :

Une *séquence de contrôle* d'un *littéral de caractère* doit dénoter un seul caractère.

exemples :

7.9 *'M'* (1.1)

5.2.4.5 Littéraux d'ensemble

syntaxe :

<littéral d'ensemble> ::= (1)
<nom d'élément d'ensemble> (1.1)

sémantique :

Un littéral d'ensemble donne une valeur d'ensemble. Un littéral d'ensemble est un nom défini dans un mode ensemble.

propriétés statiques :

La classe d'un *littéral d'ensemble* est la *M*-classe par dérivation, où M est le mode **ensemble** attaché au *nom d'élément d'ensemble.* Un *littéral d'ensemble* est **constant** et **littéral.**

exemples :

6.51 *dec* (1.1)
11.78 *king* (1.1)

5.2.4.6 Littéral de vide

syntaxe :

<littéral de vide> ::= (1)
<nom de littéral de vide> (1.1)

noms prédéfinis:

Le nom NULL est prédéfini comme nom **de littéral de vide**.

sémantique:

Le littéral de vide donne soit la valeur repère vide, c.-à-d. une valeur qui ne repère aucun locus, soit la valeur procédure vide, c.-à-d. une valeur qui n'indique aucune procédure, soit la valeur exemplaire vide, c.-à-d. une valeur qui n'identifie aucun processus.

propriétés statiques:

La classe du *littéral de vide* est la classe **nulle**. Un *littéral de vide* est **constant**.

exemples:

10.43 *NULL* (1.1)

5.2.4.7 Littéraux de chaîne de caractères

syntaxe:

<littéral de chaîne de caractères> ::= (1)
"{ *<caractère non réservé>* | *<citation>* | *<séquence de contrôle>*}*" (1.1)

<citation> ::= (2)
" " (2.1)

<séquence de contrôle> ::= (3)
^ (*<expression littérale entière>* {, *<expression littérale entière>* }*) (3.1)
| ^ *<caractère non spécial>* (3.2)
| ^ ^ (3.3)

sémantique:

Un littéral de 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 de chaîne de caractères, il faut l'écrire deux fois (" ").

A part la représentation imprimable, la *séquence de contrôle* peut être utilisée. Une *séquence de contrôle* dans laquelle le caractère accent circonflexe (^) est suivi d'une parenthèse ouverte désigne la séquence de caractères ayant pour représentations son expression *littérale entière*; sinon, si elle est suivie d'un autre caractère accent circonflexe, elle se désigne elle-même, sinon elle désigne le caractère dont la représentation est obtenue par la négation logique de b7 de la représentation interne de son *caractère non spécial* (voir l'Appendice A).

propriétés statiques:

La **longueur de chaîne** d'un *littéral de 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 de chaîne de caractères* est la **CHARS** *(n)*-classe par dérivation, où *n* est la **longueur de chaîne** du *littéral de chaîne de caractères.* Un *littéral de chaîne de caractères* est **constant**.

conditions statiques:

La valeur donnée par une *expression de littéral entier* dans *une séquence de contrôle* doit faire partie d'une gamme de valeurs définies par les représentations des caractères de l'ensemble de caractères CHILL (voir l'Appendice A).

exemples:

8.20 *"A − B<ZAA9K' "* (1.1)

5.2.4.8 Littéraux de chaîne de bits

syntaxe :

<littéral de chaîne de bits> ::= (1)
<littéral binaire de chaîne de bits> (1.1)
| <littéral octal de chaîne de bits> (1.2)
| <littéral hexadécimal de chaîne de bits> (1.3)

<littéral binaire de chaîne de bits> ::= (2)
{ B | b } ' { 0 | 1 | _ }* ' (2.1)

<littéral octal de chaîne de bits> ::= (3)
{ O | o } ' { <chiffre octal> | _ }* ' (3.1)

<littéral hexadécimal de chaîne de bits> ::= (4)
{ H | h } ' { <chiffre hexadécimal> | _ }* ' (4.1)

sémantique :

Un littéral de chaîne de bits donne une valeur chaîne de bits qui peut être de longueur 0. Les notations binaire, octale ou hexadécimale peuvent être employées. Le caractère souligné (_) n'est pas significatif, c.-à-d. qu'il ne sert qu'à améliorer la lisibilité et n'influence pas la valeur indiquée.

Un littéral de chaîne de bits 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 de bits* est soit le nombre d'occurrences de *0* et de *1* après *B'*, soit trois fois le nombre d'occurrences de *chiffre octal* après *O'*, soit quatre fois le nombre d'occurrences de *chiffre hexadécimal* après *H'*.

La classe d'un *littéral de chaîne de bits* est la **BOOLS** *(n)*-classe par dérivation, où *n* est la **longueur de chaîne** du *littéral de chaîne de bits*. Un *littéral de chaîne de bits* 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 rangée>
| <multiplet de structure> } :) (1.1)

<multiplet ensembliste> ::= (2)
[{ <expression> | <intervalle> } {, { <expression> | <intervalle> } }*] (2.1)

<intervalle> ::= (3)
<expression> : <expression> (3.1)

<multiplet de rangée> ::= (4)
<multiplet de rangée sans indices> (4.1)
| <multiplet de rangée avec indices> (4.2)

<multiplet de rangée sans indices> ::= (5)
<valeur> {, <valeur> }* (5.1)

<multiplet de rangée 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 rangée 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.-à-d. qu'il ne dénote aucune valeur. Le multiplet ensembliste peut dénoter la valeur ensembliste vide.

Si c'est une valeur rangée, il consiste en une liste de valeurs (éventuellement indicées) pour les éléments de la rangée; dans un multiplet de rangée sans indices, les valeurs sont données pour les éléments dans l'ordre croissant de leurs indices; dans un multiplet de rangée 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 rangées 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 la section 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 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 la section 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 de rangée*, alors M est le mode **des éléments** du mode du *multiplet de rangée*;
- 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 d'opération 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 de rangée* 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 de rangée*

1. Le mode du *multiplet* doit être un mode rangée.
2. La classe de chaque *valeur* doit être **compatible** avec le mode **des éléments** du mode du *multiplet.*
3. Dans le cas d'un *multiplet de rangée sans indices,* il faut autant d'occurrences de *valeur* que le **nombre d'éléments** dans le mode rangée du *multiplet.*
4. Dans le cas d'un *multiplet de rangée avec indices,* les conditions de sélection de cas doivent être remplies pour la liste d'occurrences de *liste d'étiquettes de cas* (voir la section 12.3). La **classe résultante** de la liste doit être **compatible** avec le mode **d'indice** du mode du *multiplet.* La liste de spécifications d'étiquettes de cas doit être **complète.**
5. Dans le cas d'un *multiplet de rangée 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 **d'indice** du *multiplet.*
6. Dans un *multiplet de rangée sans indices,* au moins une occurrence de *valeur* doit être une *expression.*
7. Pour un *multiplet* de rangée **constant** où le mode **des éléments** du mode du *multiplet* est un mode discret, chaque *valeur* spécifiée doit donner une valeur définie par ce mode **des éléments,** sauf si c'est une valeur **indéfinie.**

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 la section 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 marqueurs** ou un mode structure **paramétré avec marqueurs,** alors:

  5. Chaque *valeur* spécifiée pour un champ **marqueur** 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 (les) valeur(s) donnée(s) par les occurrences d'*expression littérale discrète* spécifiées pour les champs **marqueurs.**
  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 marqueurs** ou un mode structure **paramétré sans marqueurs,** 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 la *liste de noms de champ* et dans le même ordre que dans le mode du *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.*

- Si le mode du *multiplet* est un mode structure **variable avec marqueurs** ou un mode structure **paramétré avec marqueurs**, alors:

  11. Chaque *valeur* spécifiée pour un champ **marqueur** doit être une *expression littérale discrète.*

  12. Seuls les noms de **champ** correspondant à des champs indiqués comme existants par la (les) valeur(s) donnée(s) par les occurrences d'*expression littérale discrète* spécifiées pour les champs **marqueurs** peuvent être spécifiés et tous doivent l'être dans le même ordre que dans le mode du *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 marqueurs** ou un mode structure **paramétré sans marqueurs**, alors:

  14. Les noms de **champ** mentionnés dans la *liste de noms de champ*, et qui sont définis dans le même *choix de champs* doivent être tous définis dans le même *champ à choisir* ou après **ELSE**. Tous les noms de **champ** d'un choix de champs sélectionnés ou définis après **ELSE**, doivent être mentionnés une seule et unique fois et dans le même ordre que dans le mode du *multiplet.*

  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 marqueurs**, la liste de valeurs données par les occurrences d'*expression littérale discrète* spécifiées pour les champs **marqueurs**, 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 indéfinie.*

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 la section 11.2.2).

conditions dynamiques:

Les conditions d'affectation de chaque valeur pour ce qui est du mode **primitif,** du mode **des éléments**, ou du mode **de champ** associé, dans le cas respectivement d'un *multiplet ensembliste, multiplet de rangée* ou *multiplet de structure* (voir la section 6.2) s'appliquent (voir les conditions a2, b2, c4, c7, c10, c13 et c15).

Si le *multiplet* a un mode rangée 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 **indéfinie**.

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 la section 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 BOOL-classe par valeur ou la CHAR-classe par valeur selon que le mode de la *valeur primitive chaîne* est un mode chaîne de **bits** ou un mode chaîne de **caractères.**

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:

$0 \leqslant NUM$ *(élément de début)* $\leqslant L - 1$

où L est la **longueur effective** de la *valeur primitive chaîne.*

5.2.7 Valeurs tranche de chaîne

syntaxe:

<valeur tranche 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 une *tranche de chaîne* (voir la section 4.2.7).

sémantique:

Une valeur tranche 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 tranche.* La longueur (éventuellement dynamique) de la tranche de chaîne est déterminée à partir des expressions spécifiées.

Une *tranche de chaîne* dans laquelle l'*élément de droite* donne une valeur inférieure à celle que donne l'*élément de gauche* ou dans laquelle la *taille de tranche* donne une valeur non positive désigne une chaîne vide.

propriétés statiques:

La classe (éventuellement dynamique) d'une *valeur tranche 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 **composant**, et où *taille de chaîne* est soit

NUM *(élément de droite)* $-$ NUM *(élément de gauche)* $+ 1$

soit

NUM *(taille de tranche).*

Toutefois, si l'on dénote une chaîne vide, la *taille de tranche* est 0. La classe d'une *valeur tranche de chaîne* est une classe statique si la *tranche de chaîne* est **littérale**: c.-à-d. que *l'élément de gauche* et *l'élément de droite* sont **littéraux** ou que la *taille de tranche* est **littérale**; sinon, la classe est une classe dynamique.

conditions statiques:

Les relations suivantes doivent être vérifiées:

$0 \leqslant NUM\ (\textit{élément de gauche}) \leqslant L - 1$
$0 \leqslant NUM\ (\textit{élément de droite}) \leqslant L - 1$
$0 \leqslant NUM\ (\textit{élément de début}) \leqslant L - 1$
$NUM\ (\textit{élément de début}) + NUM\ (\textit{taille de tranche}) \leqslant L$

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 tranche de chaîne* ne doit pas être une valeur **indéfinie**.

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 rangée

syntaxe:

<*valeur élément de rangée*> ::= (1)
<*valeur primitive rangée*> (<*liste d'expressions*>) (1.1)

Note: Si la *valeur primitive rangée* est un *locus rangée*, la construction syntaxique est ambiguë et sera interprétée comme un *élément de rangée* (voir la section 4.2.8).

syntaxe dérivée:

Voir la section 4.2.8.

sémantique:

Une valeur élément de rangée donne une valeur qui est un élément de la valeur rangée spécifiée indiquée par *expression.*

propriétés statiques:

La classe d'une *valeur élément de rangée* est la M-classe par valeur, où M est le mode **des éléments** du mode de la *valeur primitive rangée.*

conditions statiques:

La classe de l'*expression* doit être **compatible** avec le mode **d'indice** du mode de la *valeur primitive rangée.*

conditions dynamiques:

La valeur donnée par une *valeur élément de rangée* ne doit pas être une valeur **indéfinie**.

L'exception *RANGEFAIL* est causée si la relation suivante ne se vérifie pas:

$L \leqslant \textit{expression} \leqslant U$

où L et U sont respectivement la **borne inférieure** et la **borne supérieure** (éventuellement dynamique) du mode de la *valeur primitive rangée.*

5.2.9 Valeurs tranche de rangée

syntaxe :

<valeur tranche de rangée> ::=	(1)
<valeur primitive rangée> (*<élément inférieur>* : *<élément supérieur>*)	(1.1)
| *<valeur primitive rangée>* (*<premier élément>* **UP** *<taille de tranche>*)	(1.2)

Note: Si la *valeur primitive rangée* est un *locus rangée*, la construction syntaxique est ambiguë et sera interprétée comme une *tranche de rangée* (voir la section 4.2.9).

sémantique :

Une valeur tranche de rangée donne une valeur rangée (éventuellement dynamique) qui est la partie de la valeur rangée spécifiée indiquée par *l'élément inférieur* et *l'élément supérieur*, ou par le *premier élément* et la *taille de tranche*. La **borne inférieure** de la valeur tranche de rangée est égale à la **borne inférieure** de la valeur rangée spécifiée; la **borne supérieure** (éventuellement dynamique) est déterminée à partir des expressions spécifiées.

propriétés statiques :

La classe (éventuellement dynamique) d'une *valeur tranche de rangée* est la M-classe par valeur où M est un mode rangée **paramétré** construit comme:

&nom (indice supérieur)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) de la *valeur primitive rangée* 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 tranche) − 1

où L est la **borne inférieure** du mode de la *valeur primitive rangée*.

La classe d'une *valeur tranche de rangée* est une classe statique si *l'indice supérieur* est **littéral**: c.-à-d. que *l'élément inférieur* et *l'élément supérieur* sont tous deux **littéraux** ou que la *taille de tranche* 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 **d'indice** de la *valeur primitive rangée*.

Les relations suivantes doivent être vérifiées:

L ≤ élément inférieur ≤ élément supérieur ≤ U

1 ≤ NUM (taille de tranche) ≤ NUM (U) − NUM (L) + 1

NUM (L) ≤ NUM (premier élément) ≤ NUM (premier élément) + NUM (taille de tranche) − 1
≤ NUM (U)

où L et U sont, respectivement, la **borne inférieure** et la **borne supérieure** du mode de la *valeur primitive rangée*. Si U et la valeur de toutes les *expressions* sont connus statiquement, les relations peuvent être vérifiées statiquement.

conditions dynamiques :

La valeur donnée par une *valeur tranche de rangée* ne doit pas être une valeur **indéfinie**.

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 la section 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 marqueurs** 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*.

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 **indéfinie**.

Une valeur ne doit pas dénoter:

- un mode structure **variable avec marqueurs** et que la (les) valeur(s) de(s) champ(s) **marqueur(s)** associé(s) indique(nt) 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*.

exemples:

16.51 (**RECEIVE** *user_buffer*) . *allocator* (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 la section 4.2.13).

sémantique:

Une conversion d'expression prend le pas sur les règles de vérification de compatibilité et des modes de CHILL. Elle attache explicitement un mode à l'expression. Si le mode du *nom de mode* est un mode discret et que la classe de la valeur donnée par l'expression est discrète, alors, la valeur donnée par la conversion d'expression est telle que:

NUM (nom de mode (expression)) = NUM (expression)

Sinon, la valeur donnée par la conversion d'expression est définie par l'implémentation et dépend de la représentation interne des valeurs.

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**. Une implémentation peut imposer des conditions statiques supplémentaires.

conditions dynamiques:

Si la classe de la valeur donnée par *l'expression* est discrète et si le mode du *nom de mode* est un mode discret qui ne définit pas une valeur mais une représentation interne égale à *NUM (expression)*, alors l'exception *OVERFLOW* est causée. Une implémentation peut imposer des conditions dynamiques supplémentaires qui, si elles sont violées, causent l'occurrence d'une exception définie par l'implémentation.

5.2.12 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 de la **spec de résultat** de *l'appel de procédure rendant valeur.*

conditions dynamiques:

L'*appel de procédure rendant valeur* ne doit pas donner de valeur **indéfinie** (voir les sections 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)

5.2.13 Appels d'opération prédéfinie rendant valeur

syntaxe:

<appel d'opération prédéfinie rendant valeur> ::=	(1)
<appel d'opération prédéfinie rendant valeur>	(1.1)

sémantique:

Un appel d'opération prédéfinie rendant valeur fournit la valeur renvoyée par l'opération prédéfinie.

propriétés statiques:

La classe attachée à l'*appel d'opération prédéfinie rendant valeur* est celle de *l'appel d'opération prédéfinie rendant valeur.*

conditions dynamiques:

L'*appel d'opération prédéfinie rendant valeur* ne doit pas donner de valeur **indéfinie** (voir les sections 5.3.1 et 6.8).

5.2.14 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 le chapitre 11). L'expression démarrer donne une valeur exemplaire 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 la section 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 la section 6.7).

L'*expression démarrer* cause l'exception *SPACEFAIL* si les besoins de mémoire ne peuvent pas être satisfaits.

exemples:

15.35	START *counter ( )*	(1.1)

5.2.15 Opérateur nullaire

syntaxe:

<*opérateur nullaire*> ::=	(1)
THIS	(1.1)

sémantique:

L'opérateur nullaire donne la valeur exemplaire univoque qui identifie le processus qui l'exécute.

propriétés statiques:

La classe de *l'opérateur nullaire* est la *INSTANCE*-classe par dérivation.

5.2.16 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).**

exemples:

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 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 ce document. Une valeur composée est **indéfinie** si et seulement si tous ses sous-composants (c.-à-d. valeurs sous-chaîne, valeurs élément, valeurs champ) sont **indéfinis.**

exemples:

6.40 *(146_097*c)/4+(1_461*y)/4*
+(153+m+c)/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>* *<solution alors>* *<solution sinon>* **FI**	(2.1)
| **CASE** *<liste de sélecteurs de cas>* **OF** { *<solution cas de valeur>* }+ [**ELSE** *<sous-expression>*] **ESAC**	(2.2)
<solution alors> ::=	(3)
THEN *<sous-expression>*	(3.1)
<solution sinon> ::=	(4)
ELSE *<sous-expression>*	(4.1)
| **ELSIF** *<expression booléenne>* *<solution alors>* *<solution sinon>*	(4.2)
<sous-expression> ::=	(5)
<expression>	(5.1)
<solution 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 la *solution alors*; sinon celle de la *solution sinon.*

La valeur fournie par une *solution 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 la *solution alors*; sinon c'est celle de la *solution 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 la section 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 la section 12.3) et les mêmes caractéristiques d'exhaustivité, de cohérence et de compatibilité que pour le cas action doivent être assurées (voir la section 6.4);
- pas d'*expression conditionnelle* peut comporter deux occurrences *de sous-expression*, respectivement **extrarégionale** et **intrarégionale** (voir la section 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 de bits, auquel cas **OR** et **XOR** désignent les opérations logiques sur chaque élément des chaînes de bits, donnant une valeur chaîne de bits;
- 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 de bits et la **longueur effective** du *sous-opérande-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 une valeur booléenne qui est *FALSE*, il s'agit du résultat, sinon le résultat est *l'opérande-2*;
- des valeurs chaîne de bits, auquel cas **AND** désigne l'opération logique sur chaque élément des chaînes de bits donnant une valeur chaîne de bits;
- 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 de **bits** 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 ou chaîne. 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 ou chaîne;
- 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 et la valeur entière résultante est la somme (+) ou différence (−) des deux valeurs.

Si l'*opérateur-4* est un opérateur de concaténation de chaîne, les deux opérandes donnent soit des valeurs chaîne de bits 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 de bits (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.*

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 de bits 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 un *opérateur arithmétique additif*, les classes des deux opérandes doivent être **compatibles** et avoir toutes les deux un mode **racine** entier;
- 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 de **bits** ou un mode **racine** chaîne de **caractères**, ou
  - les classes des deux opérandes doivent être **compatibles** avec le mode *BOOL* ou avec le mode *CHAR*, ou
  - la classe d'un opérande doit avoir un mode **racine** chaîne de **bits (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:

Dans le cas d'un *opérande-3* qui n'est pas **constant**, 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*.

exemples:

1.6	*j*	(1.2)
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 un opérateur arithmétique multiplicatif est spécifié, *sous-opérande-4* et *opérande-5* donnent des valeurs entières et la valeur entière résultante est soit le produit (*), le quotient (/), modulo (**MOD**), soit le reste de la division (**REM**) des deux valeurs.

L'opération modulo est définie de telle manière que *i* **MOD** *j* donne l'entier unique k, $0 \leqslant k < j$ tel qu'il existe un entier 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.

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, les classes de l'*opérande-5* et du *sous-opérande-4* doivent être **compatibles** et toutes deux doivent avoir un mode **racine** entier.

conditions dynamiques:

Dans le cas d'un *opérande-4*, qui n'est pas **constant**, l'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.-à-d. division ou reste avec un *opérande-5* donnant 0 ou une opération modulo avec un *opérande-5* donnant une valeur entière non positive.

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)
[*<opérateur unaire>*] *<opérande-6>*	(1.1)
<opérateur unaire> ::=	(2)
– | **NOT**	(2.1)
| *<opérateur de répétition de chaîne>*	(2.2)
<opérateur de répétition de chaîne> ::=	(3)
(<expression littérale entière>)	(3.1)

sémantique:

Si l'opérateur unaire est l'opérateur changer-le-signe (–), l'*opérande-6* donne une valeur entière et la valeur entière résultante est la valeur entière précédente changée de signe.

Si l'opérateur unaire est **NOT**, l'*opérande-6* donne soit une valeur booléenne soit une valeur chaîne de bits, soit une valeur ensembliste. Dans les deux premiers cas, la négation logique de la valeur booléenne ou chaîne de bits est donnée; dans le dernier cas, la valeur ensembliste complémentaire, c.-à-d. 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-6* est un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits.* Si *l'expression littérale entière* 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 littérale moins 1.

propriétés statiques:

Si *l'opérande-5* est un *opérande-6*, la classe de *l'opérande-5* est la classe de *l'opérande-6.*

Si un *opérateur unaire* est spécifié, la classe de *l'opérande-5* est:

- si *l'opérateur unaire* est – ou **NOT**, alors la **classe résultante** de celle de l'*opérande-6*;
- 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 de chaîne de caractères* ou un *littéral de chaîne de bits*) où $n = r * l$, où r est la valeur donnée par *l'expression entière littérale* et l est la **longueur de chaîne** du littéral chaîne.

Un *opérande-5* est **constant** si et seulement si *l'opérande-6* est **constant**. Un *opérande-5* est **littéral** si et seulement si *l'opérande-6* est **littéral** et que *l'opérateur unaire* est – ou *NOT.*

conditions statiques:

Si *l'opérateur unaire* est –, la classe de l'*opérande-6* doit avoir un mode **racine** entier.

Si *l'opérateur unaire* est **NOT**, la classe de *l'opérande-6* doit avoir un mode **racine** booléen, chaîne **de bits** ou ensembliste.

Si *l'opérateur unaire* est *l'opérateur de répétition de chaîne, l'opérande-6* doit être un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits.* L'*expression littérale entière* doit donner une valeur entière non négative.

conditions dynamiques :

Si *l'opérande-5* 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-5.*

exemples :

5.10	**NOT** *k2*	(1.1)
7.54	*(6)' '*	(1.1)
7.54	*(6)*	(2.2)

5.3.9 Opérande-6

syntaxe :

<opérande-6> ::= (1)
  <locus repéré> (1.1)
  | *<expression recevoir>* (1.2)
  | *<valeur primitive>* (1.3)

<locus repéré> ::= (2)
  − > *<locus>* (2.1)

<expression recevoir> ::= (3)
  RECEIVE *<locus tampon>* (3.1)

sémantique :

Un locus repéré donne un repère au locus spécifié.

L'expression recevoir reçoit une valeur du *locus tampon.* Le processus exécutant peut être mis en attente et peut réactiver un autre processus, mis en attente à l'envoi vers le locus tampon spécifié (voir la section 6.19.3 pour tous les détails).

propriétés statiques :

La classe de *l'opérande-6* est la classe du *locus repéré,* de *l'expression recevoir* ou de la *valeur primitive,* respectivement. La classe du *locus repéré* est la M-classe par repérage, où M est le mode du *locus.* La classe de *l'expression recevoir* est la M-classe par valeur, où M est le mode **des éléments de tampon** du mode du *locus tampon.*

Un *opérande-6* est **constant** si et seulement si la *valeur primitive* est **constante** ou si le *locus repéré* est **constant.** Un *locus repéré* est **constant** si et seulement si le *locus* est un locus **statique.** Un *opérande-6* est **littéral** si et seulement si la *valeur primitive* est **littérale.**

conditions statiques :

Le *locus* doit être **repérable.**

conditions dynamiques :

La durée de vie du *locus tampon* dénoté ne doit pas se terminer pendant que le processus exécutant est en attente pour ce locus tampon.

exemples :

8.25	− > *c*	(2.1)
16.51	**RECEIVE** *user_buffer*	(3.1)

6 ACTIONS

6.1 GÉNÉRALITÉS

syntaxe :

<énoncé d'action> ::= (1)
[*<définition>* :] *<action>* [*<filet>*] [*<représentation textuelle 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 causer>* (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 et choisir>* (3.5)
| *<action recevoir et choisir>* (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 *définition* apparaissant dans un *énoncé d'action* définit un nom **d'étiquette**.

conditions statiques :

La *représentation textuelle 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 *définition* est spécifiée. La *représentation textuelle de nom simple* doit être la même représentation textuelle que la *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 arithmétque 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(s) locus spécifié(s) 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 (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 **marqueur** 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/ou d'une valeur de classe dynamique.

La *valeur* doit être **régionalement sûre** pour chaque *locus* (voir la section 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.

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 et si la valeur fournie par l'évaluation de *valeur* n'est ni l'une des valeurs définies par le mode intervalle ni la valeur **indéfinie**.

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).

exemples:

4.12	*a := b+c*	(1.1)
10.25	*stackindex– := 1*	(2.1)
19.19	*x →.prex, 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    IFn > = 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 DE CAS

syntaxe:

<action de cas> ::= (1)
CASE *<liste de sélecteurs de cas>* **OF** [*<liste d'intervalles>* ;] { *<cas à choisir>* }+
[**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 à choisir> ::= (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 à choisir). 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 à choisir 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 la section 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 littérale discrète* ou définie par un *intervalle littéral* ou un nom de *mode discret* dans une *étiquette de cas* (voir la section 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 la section 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 de commande>* ;] *<liste d'énoncés d'action>* **OD**	(1.1)
<partie de 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)
<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 composé>*	(11.1)
<objet composé> ::=	(12)
<locus rangée>	(12.1)
| *<expression rangée>*	(12.2)
| *<locus chaîne>*	(12.3)
| *<expression chaîne>*	(12.4)

Note: Si l'*objet composé* et un *locus* (*chaîne, rangée*), on résout l'ambiguïté syntactique en interprétant l'*objet composé* 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. **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 termaison 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 rangée dénoté par le *locus rangée* ou les composants du locus chaîne désigné par le *locus chaîne*. Si une *expression rangée* ou *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 de loc-identité:

DCL *<compteur de boucle> <mode>* **LOC** := *<objet composé> (<indice>)*;

était rencontrée, où *mode* est le mode des éléments du mode du *locus rangée*, où *&nom(1)* tel que *&nom* est un nom de **synmode** virtuel **synonyme** du mode du locus chaîne, s'il s'agit d'un mode chaîne **fixe**, sinon du mode **composant**, et où *l'indice* est réglé initialement sur la **borne inférieure (borne supérieure)** du mode du 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 du *locus*. Quand l'action faire est terminée, *l'objet composé* 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 sa *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 de *l'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 **des éléments** du mode du *locus rangée* ou *expression rangée* ou le mode chaîne *&nom(1)*, où *&nom* est un nom de **synmode** virtuel **synonyme** du mode du *locus chaîne* ou du *mode* **racine** de *l'expression chaîne.*

Un nom **d'énumération de locus** est **repérable** si l'implantation d'élément du mode du *locus rangée* 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.**

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.

exemples:

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 des 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 définition de nom de champ était introduite pour chaque nom **de champ** attaché au mode du 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 du *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 de la *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 et les évaluations peuvent éventuellement se mélanger.

propriétés statiques :

La définition (virtuelle) introduite pour un nom de **champ** a la même chaîne de noms que la *définition de noms de champ* de ce nom de **champ**.

Si une *valeur primitive structure* est spécifiée, une 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 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 du *locus structure*, qui est rendu disponible comme nom de **locus faire-avec**. Un nom de **locus faire-avec** est **repérable** si l'implantation de champ du nom de **champ** associé est **NOPACK**.

exemples :

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 la *définition* qui précède a la même représentation textuelle 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.-à-d. que l'action sortir ne peut s'employer pour quitter des procédures ou des processus).

Aucun *filet* ne peut terminer une *action sortir*.

exemples:

15.62 **EXIT** *find_counter* (1.1)

6.7 ACTION APPELER

syntaxe:

<action appeler> ::= (1)
  <appel de procédure> (1.1)
  | *<appel d'opération prédéfinie>* (1.2)

<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 d'opération prédéfinie> ::= (5)
  <nom d'opération prédéfinie> *(* [*<liste de paramètres d'opération prédéfinie>*] *)* (5.1)

<liste de paramètres d'opération prédéfinie> ::= (6)
  <paramètre d'opération prédéfinie> {, *<paramètre d'opération prédéfinie>*}* (6.1)

<paramètre d'opération prédéfinie> ::= (7)
  <valeur> (7.1)
  | *<locus>* (7.2)
  | *<nom non réservé>* [(*<liste de paramètres d'opération prédéfinie>*)] (7.3)

Note: Si le *paramètre effectif* ou le *paramètre d'opération prédéfinie* est un *locus*, on élimine l'ambiguïté syntaxique en l'interprétant comme *un locus* et non comme une *valeur*.

sémantique:

Une action appeler cause l'appel soit d'une procédure soit d'une opération prédéfinie. 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*. Les valeurs et les locus effectifs spécifiés dans la liste des paramètres effectifs sont transmis à la procédure.

Un *appel d'opération prédéfinie* respectivement est soit un *appel d'opération prédéfinie* CHILL, soit un *appel d'opération prédéfinie* par l'implémentation (voir les sections 6.20 et 13.1).

Une valeur, un locus ou tout nom défini de programme qui n'est pas une représentation textuelle de nom simple **réservée** peut être passé comme un *paramètre d'opération prédéfinie*. L'appel d'opération prédéfinie peut envoyer une valeur ou un locus.

Une opération prédéfinie peut être générique, c'est-à-dire que sa classe (si c'est un appel d'opération prédéfinie rendant **valeur**) ou son mode (si c'est un appel d'opération prédéfinie rendant **locus**) peut dépendre non seulement du *nom d'opération prédéfinie* mais aussi des propriétés statiques des paramètres effectifs passés et du contexte statique de l'appel.

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 la section 11.2.2). Ces propriétés sont héritées du *nom de procédure* 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 d'opération prédéfinie* est un nom défini par 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 la section 10.8).

Un *appel d'opération prédéfinie* est un *appel d'opération prédéfinie* **rendant locus** s'il livre un locus; c'est un *appel d'opération prédéfinie* **rendant valeur** s'il livre une valeur.

conditions statiques:

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 **repérable** et tel que le mode dans la spec de paramètre soit **compatible en lecture** avec le mode de ce *locus* (effectif), ou soit une *valeur* qui n'est pas un *locus* mais dont la classe est **compatible** avec le mode dans la spec de paramètre.
- Si la spec de paramètre a l'attribut **LOC**, **DYNAMIC** étant spécifié, le *paramètre effectif* doit être un *locus* qui est à la fois **repérable** et tel que le mode dans la spec de paramètre soit **compatible en lecture dynamique** avec le mode de ce *locus* (effectif), ou soit 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* ou un *appel d'opération prédéfinie* 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 la section 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 la section 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 la section 6.2).

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.

exemples:

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 **repérable** 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.-à-d. 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.*

exemples:

4.7	**ASSERT** $b>0$ **AND** $c>0$ **AND** $oder>0$	(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 CAUSER

syntaxe :

<*action causer*> ::= (1)
CAUSE <*nom d'exception*> (1.1)

sémantique :

Une action causer cause l'exception dont le nom est indiqué par *nom d'exception.*

conditions statiques :

Un *filet* ne doit pas terminer une *action causer.*

exemples :

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 la section 5.2.14), éventuellement sans employer la valeur exemplaire donnée par cette expression.

exemples :

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 la section 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.

exemples :

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 littérale entière>* (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 attente associés au locus événement spécifié. La priorité est celle qui est spécifiée, le cas échéant, sinon elle est égale à 0 (c.-à-d. 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 littérale entière* ne peut pas donner une valeur négative.

conditions dynamiques :

L'exception *DELAYFAIL* est causée si le mode du *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 en attente sur lui.

exemples :

13.18 **DELAY** *resource_freed* (1.1)

6.17 ACTION METTRE EN ATTENTE ET CHOISIR

syntaxe :

<action mettre en attente et choisir> ::= (1)
  DELAY CASE [**SET** *<locus exemplaire>* [*<priorité>*]; | *<priorité>* ;]
  { *<événement à choisir>* }+
  ESAC (1.1)

<événement à choisir> ::= (2)
  (<liste d'événements>) : <liste d'énoncés d'action> (2.1)

<liste d'événements> ::= (3)
  <locus événement> {, *<locus événement>* }* (3.1)

sémantique :

Une action mettre en attente et choisir évalue, dans un ordre non spécifié ou éventuellement mélangé, un *locus exemplaire*, s'il y en a un, et tous les *locus événement* spécifiés dans un *choix de mise en attente.*

Ensuite, une exception DELAYFAIL se produit (voir ci-dessous) ou le processus exécutant est mis en attente.

Si le processus exécutant est mis en attente, il devient membre, avec une certaine priorité, de l'ensemble des processus mis 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 mis 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 *choix de mise en attente* spécifient le même locus événement, le choix entre ces choix n'est pas spécifié. Avant d'entamer, si un *locus exemplaire* est spécifié, la valeur exemplaire identifiant le processus qui a exécuté l'action continuer est mémorisée dans ce locus.

propriétés dynamiques :

Un processus qui exécute une action mettre en attente et choisir 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 :

Le mode du *locus exemplaire* ne doit pas avoir la **propriété de protection**. L'*expression littérale entière* 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 mettre en attente et choisir est en attente sur lui.

L'exception SPACEFAIL est causée lorsque les besoins de mémoire ne peuvent pas être satisfaits.

exemples :

```
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 processus 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 exemplaire*>] [<*priorité*>]	(1.1)

sémantique :

Une action envoyer signal évalue dans un ordre non spécifié et éventuellement mélangé, la liste des *valeurs*, s'il y en a une, et la *valeur primitive exemplaire*, s'il y en a une.

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 exemplaire* est spécifiée, seul ce processus peut recevoir le signal; sinon tout processus 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 mis en attente auxquels il appartenait.

conditions statiques :

Le nombre d'occurrences de *valeur* doit être égal au nombre de modes du *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 la section 11.2.2). L'*expression littérale entière* 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 exemplaire* donne *NULL*.

La durée de vie du processus indiqué par la valeur donnée par la *valeur primitive exemplaire* 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 exemplaire*.

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 mis 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 mis en attente, il devient membre de l'ensemble des processus envoyants mis en attente associé au locus tampon. Si un processus est réactivé, il ne fait plus partie de tous les ensembles de processus mis 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 mis en attente. Il cesse d'être **temporisable** lorsqu'il quitte ce point.

conditions statiques:

La classe de *valeur* doit être **compatible** avec le mode **des éléments de tampon** du mode du *locus tampon*. La *valeur* ne doit pas être **intrarégionale** (voir la section 11.2.2). L'*expression littérale entière* dans *priorité* ne doit pas donner une valeur négative.

conditions dynamiques:

Les conditions d'affectation de la *valeur* en tenant compte du mode **des éléments de tampon** du mode du *locus tampon* doivent être respectées. Les exceptions possibles sont causées avant que le processus ne soit mis 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.

exemples:

16.119 **SEND** *user* – >([*ready*, – > *counter_buffer*]) *k* (1.1)

6.19 ACTION RECEVOIR ET CHOISIR

6.19.1 Généralités

syntaxe:

<action recevoir et choisir> ::= (1)
<action recevoir signal et choisir> (1.1)
| *<action recevoir tampon et choisir>* (1.2)

sémantique:

Une action recevoir et choisir 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 et choisir ne se traduit pas nécessairement par la mise en attente du processus exécutant (voir le chapitre 11 pour plus de détails).

6.19.2 Action recevoir signal et choisir

syntaxe:

<*action recevoir signal et choisir*> ::= (1)
  RECEIVE CASE [**SET** <*locus exemplaire*> ;]
  { <*signal à chosir*>) }+
  [**ELSE** <*liste d'énoncés d'action*>] **ESAC** (1.1)

<*signal à choisir*> ::= (2)
  (<*nom de signal*> [**IN** <*liste de définitions*>]) : <*liste d'énoncés d'action*> (2.1)

sémantique:

Une action recevoir signal et choisir évalue le *locus exemplaire*, s'il est présent.

Ensuite, le processus exécutant reçoit (immédiatement) un signal ou, si **ELSE** est spécifié, entame *la liste d'énoncés d'action* correspondante, sinon il est mis en attente. Le processus exécutant reçoit immédiatement un signal si un *nom de signal* spécifié dans un *choix de signal à recevoir* 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 mis en attente, il devient membre de l'ensemble des processus mis en attente attachés à chacun des signaux spécifiés. Si le processus mis 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 exemplaire* est spécifié, la valeur exemplaire 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 *définition* de la *liste de définitions* d'un *signal à choisir* 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 et choisir 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:

Le mode du *locus exemplaire* ne doit pas avoir la **propriété de protection**.

Toutes les occurrences de *nom de signal* doivent être différentes.

Le **IN** facultatif et la *liste de définitions* dans le *signal à choisir* ne doivent être spécifiés que si le *nom de signal* a un ensemble de modes non vide. Le nombre de noms dans la *liste de définitions* doit être égal au nombre de modes du *nom de signal*.

conditions dynamiques:

L'exception *SPACEFAIL* est causée si les besoins de mémoire ne peuvent pas être satisfaits.

exemples:

```
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 et choisir

syntaxe:

<*action recevoir tampon et choisir*> ::= (1)

RECEIVE CASE [**SET** <*locus exemplaire*> ;]
{ <*tampon à choisir*> }+
[**ELSE** <*liste d'énoncés d'action*>]
ESAC (1.1)

<*tampon à choisir*> ::= (2)

(<*locus tampon*> **IN** <*définition*>) : <*liste d'énoncés d'action*> (2.1)

sémantique:

Une action recevoir tampon et choisir évalue, dans un ordre non spécifié et éventuellement mélangé, le *locus exemplaire*, s'il est présent, et tous les *locus tampon* spécifiés dans un *choix recevoir 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 mis en attente. Le processus exécutant reçoit immédiatement une valeur s'il en a une qui est stockée dans, ou un processus envoyer mis 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 mis en attente, il devient membre de l'ensemble des processus mis en attente attachés à chacun des locus tampon spécifiés. Si le processus mis 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 *tampon à choisir* spécifient le même locus tampon, le choix entre ces choix n'est pas spécifié. Avant d'entamer, si un *locus exemplaire* est spécifié, la valeur exemplaire 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 mis 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 mis en attente attachés au locus tampon.

propriétés statiques:

Une *définition* dans un *tampon à choisir* définit un nom de **valeur reçue**. Sa classe est la M-classe par valeur, où M est le mode des **éléments de tampon** du mode du *locus tampon* qualifiant le *tampon à choisir.*

propriétés dynamiques:

Un processus qui exécute une action recevoir tampon et choisir 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:

Le mode de *locus exemplaire* ne doit pas avoir la **propriété de protection.**

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 D'OPÉRATION PRÉDÉFINIE CHILL

syntaxe :

<appel d'opération prédéfinie CHILL> ::=	(1)
<appel d'opération prédéfinie simple CHILL>	(1.1)
| *<appel d'opération prédéfinie rendant locus CHILL>*	(1.2)
| *<appel d'opération prédéfinie rendant valeur CHILL>*	(1.3)

noms prédéfinis :

Les noms d'opération prédéfinie CHILL sont prédéfinis comme des noms **d'opération prédéfinie** (voir la section 6.7).

sémantique :

Un *appel d'opération prédéfinie CHILL* est un *appel d'opération prédéfinie simple CHILL* qui ne fournit pas de résultat (voir la section 6.20.1), ou un *appel d'opération prédéfinie rendant locus CHILL* qui donne un locus (voir la section 6.20.2), ou un *appel d'opération prédéfinie rendant valeur CHILL* qui donne une valeur (voir la section 6.20.3).

propriétés statiques :

Un *appel d'opération prédéfinie CHILL* est un *appel d'opération prédéfinie* de **locus** si c'est un *appel d'opération prédéfinie rendant locus CHILL*; c'est un *appel d'opération prédéfinie* de **valeur** si c'est un *appel d'opération prédéfinie rendant valeur CHILL*.

6.20.1 Appels d'opération prédéfinie simple CHILL

syntaxe :

<appel d'opération prédéfinie simple CHILL> ::=	(1)
<appel d'opération prédéfinie terminer>	(1.1)
| *<appel d'opération prédéfinie simple d'e/s>*	(1.2)
| *<appel d'opération prédéfinie simple de temporisation>*	(1.3)

sémantique :

Un *appel d'opération prédéfinie simple CHILL* est un *appel d'opération prédéfinie* qui ne donne ni valeur ni locus. Les opérations prédéfinies simples pour l'entrée-sortie sont décrites au chapitre 7. Les opérations prédéfinies simples pour la temporisation sont décrites au chapitre 9.

6.20.2 Appels d'opération prédéfinie rendant locus CHILL

syntaxe :

<appel d'opération prédéfinie rendant locus CHILL> ::=	(1)
<appel d'opération prédéfinie rendant locus d'e/s>	(1.1)

sémantique :

Un *appel d'opération prédéfinie rendant locus CHILL* est un *appel d'opération prédéfinie* qui donne un locus. Les opérations prédéfinies rendant locus pour l'entrée-sortie sont décrites au chapitre 7.

6.20.3 **Appels d'opération prédéfinie rendant valeur CHILL**

syntaxe :

<appel d'opération prédéfinie rendant valeur CHILL> ::= (1)
NUM (<expression discrète>) (1.1)
| *PRED (<expression discrète>)* (1.2)
| *SUCC (<expression discrète>)* (1.3)
| *ABS (<expression entière>)* (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 upper lower>)* (1.9)
| *LOWER (<argument pour upper lower>)* (1.10)
| *LENGTH (<argument de longueur>)* (1.11)
| *<appel d'opération prédéfinie affecter>* (1.12)
| *<appel d'opération prédéfinie rendant valeur d'e/s>* (1.13)
| *<appel d'opération prédéfinie de valeur temps>* (1.14)

<argument de mode> ::= (2)
<nom de mode> (2.1)
| *<nom de mode rangée> (<expression>)* (2.2)
| *<nom de mode chaîne> (<expression entière>)* (2.3)
| *<nom de mode structure variable> (<liste d'expressions>)* (2.4)

<argument pour upper lower> ::= (3)
<locus rangée> (3.1)
| *<expression rangée>* (3.2)
| *<nom de mode rangée>* (3.3)
| *<locus chaîne>* (3.4)
| *<expression chaîne>* (3.5)
| *<nom de mode chaîne>* (3.6)
| *<locus discret>* (3.7)
| *<expression discrète>* (3.8)
| *<nom de mode discret>* (3.9)

<argument longueur> ::= (4)
<locus chaîne> (4.1)
| *<expression chaîne>* (4.2)

Note: Si *l'argument* pour *upper lower* est un *locus* (*rangée*, *chaîne*, *discret*), l'ambiguïté syntaxique est résolue comme suit: on interprète *argument pour upper lower* 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 d'opération prédéfinie rendant valeur CHILL* est un *appel d'opération 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 donne la valeur absolue de son argument.

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 **repérables** 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 **repérable** 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 rangée, chaîne et discrets, donnant la **borne supérieure** et la **borne inférieure** du mode du locus,
- expressions rangée et chaîne, donnant la **borne supérieure** et la **borne inférieure** du mode de la classe de valeur,
- expressions discrètes **fortes**, donnant la **borne supérieure** et la **borne inférieure** du mode de la classe de valeur,
- noms de **mode** rangée, chaîne et discrets, donnant la **borne supérieure** et la **borne inférieure** du mode.

LENGTH donne la **longueur effective** de l'argument de longueur.

propriétés statiques:

La classe d'un appel d'opération prédéfinie *NUM* est la *INT*-classe par dérivation. L'appel d'opération prédéfinie est **constant** si et seulement si l'argument est **constant** ou **littéral.**

La classe d'un appel d'opération prédéfinie *PRED* ou *SUCC* est la **classe résultante** de l'argument. L'appel d'opération prédéfinie est **constant (littéral)** si et seulement si l'argument est **constant (littéral)**.

La classe d'un appel d'opération prédéfinie *ABS* est la **classe résultante** de l'argument. L'appel d'opération prédéfinie est **constant (littéral)** si et seulement si l'argument est **constant (littéral)**.

La classe d'un appel d'opération prédéfinie *CARD* est la *INT*-classe par dérivation. L'appel d'opération prédéfinie est **constant** si et seulement si l'argument est **constant.**

La classe d'un appel d'opération prédéfinie *MAX* ou *MIN* est la M-classe par valeur, où M est le mode **primitif** du mode de *l'expression ensembliste*. L'appel d'opération prédéfinie est **constant** si et seulement si l'argument est **constant.**

La classe d'un appel d'opération prédéfinie *SIZE* est la *INT*-classe par dérivation. L'appel d'opération prédéfinie est **constant** si le mode de l'argument est statique.

La classe d'un appel d'opération prédéfinie *UPPER* et *LOWER* est:

- la M-classe par valeur si *l'argument pour upper lower* est un *locus rangée*, une *expression rangée* ou un *nom de mode rangée*, où M est respectivement le mode **d'indice** du *locus rangée*, d'une *expression rangée* ou un *nom de mode rangée*;
- la *INT*-classe par dérivation si *l'argument pour upper lower* est un *locus chaîne*, une *expression chaîne* ou un *nom de mode chaîne*;
- la M-classe par valeur si *l'argument pour upper lower* 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*.

Un appel d'opération prédéfinie *UPPER* et *LOWER* est **constant** si *l'argument pour upper lower* est un *nom de mode* (*rangée*, *chaîne* ou *discret*), si le mode du *locus rangée* ou *chaîne* est statique, si *l'expression rangée* ou *chaîne* a une classe statique, ou si *l'argument pour upper lower* est une *expression discrète* ou un *locus discret*.

La classe d'un appel d'opération prédéfinie LENGTH est la INT-classe par dérivation.

conditions statiques:

Si l'argument d'un appel d'opération 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 **sans numéros.**

Si l'argument d'un appel d'opération prédéfinie *MAX* ou *MIN* est **constant**, il ne doit pas donner la valeur ensembliste vide.

L'argument pour *locus* de *SIZE* doit être **repérable.**

L'*expression discrète* en tant qu'argument de *UPPER* et *LOWER* doit être **forte.**

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 **d'indice** du mode du *nom de mode rangée.*
- 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 la liste de classes du *nom de mode structure variable* et la classe de chaque expression doit être **compatible** avec la classe correspondante de la liste de classes.

conditions dynamiques:

PRED et *SUCC* 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* 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 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 **d'indice** du *nom de mode rangée*;
- *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.-à-d. est **forte**), donne une valeur qui est en dehors de l'ensemble de valeurs définies par M.

exemples:

9.12	*MIN (sieve)*	(1.7)
11.47	*PRED (col_1)*	(1.2)
11.47	*SUCC (col_1)*	(1.3)

6.20.4 Opérations prédéfinies de traitement de mémoire dynamique

syntaxe:

```
<appel d'opération prédéfinie affecter> ::=                          (1)
        GETSTACK ( <argument de mode> [, <valeur> ])                 (1.1)
      | ALLOCATE ( <argument de mode> [, <valeur> ])                 (1.2)

<appel d'opération prédéfinie terminer> ::=                          (2)
        TERMINATE ( <valeur primitive repère> )                      (2.1)
```

sémantique:

GETSTACK et *ALLOCATE* créent un locus du mode spécifié et donnent une valeur repère pour le locus créé. *GETSTACK* crée ce locus sur la pile (voir la section 10.9). Un locus dont le mode est celui de *l'argument de mode* est créé et une valeur se repérant à lui est donnée. Le locus créé est initialisé avec la valeur *de valeur,* si présente; sinon, avec la valeur **indéfinie** (voir la section 4.1.2).

TERMINATE met fin à la durée de vie du locus repéré par la valeur donnée par la *valeur primitive repère.* Une implémentation pourrait en conséquence libérer la mémoire occupée par ce locus.

propriétés statiques:

La classe d'un appel d'opération prédéfinie *GETSTACK* ou *ALLOCATE* est la M-classe par repère, 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 rangée> (<expression>) ou

& <nom de mode chaîne> (<expression entière>) ou

& <nom de mode structure variable> (<liste d'expressions>),

respectivement.

Un appel d'opération 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 d'opération 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 premier argument de *GETSTACK* ou *ALLOCATE* a la **propriété d'être protégé**, le second argument doit être présent.

La *valeur,* si elle est présente, dans l'appel d'opération prédéfinie *GETSTACK* et *ALLOCATE,* doit être **régionalement sûre** pour le locus créé.

propriétés dynamiques:

Une valeur repère est une valeur repère **affectée** si, et seulement si elle est envoyée par un appel d'opération 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 repère* donne la valeur NULL.

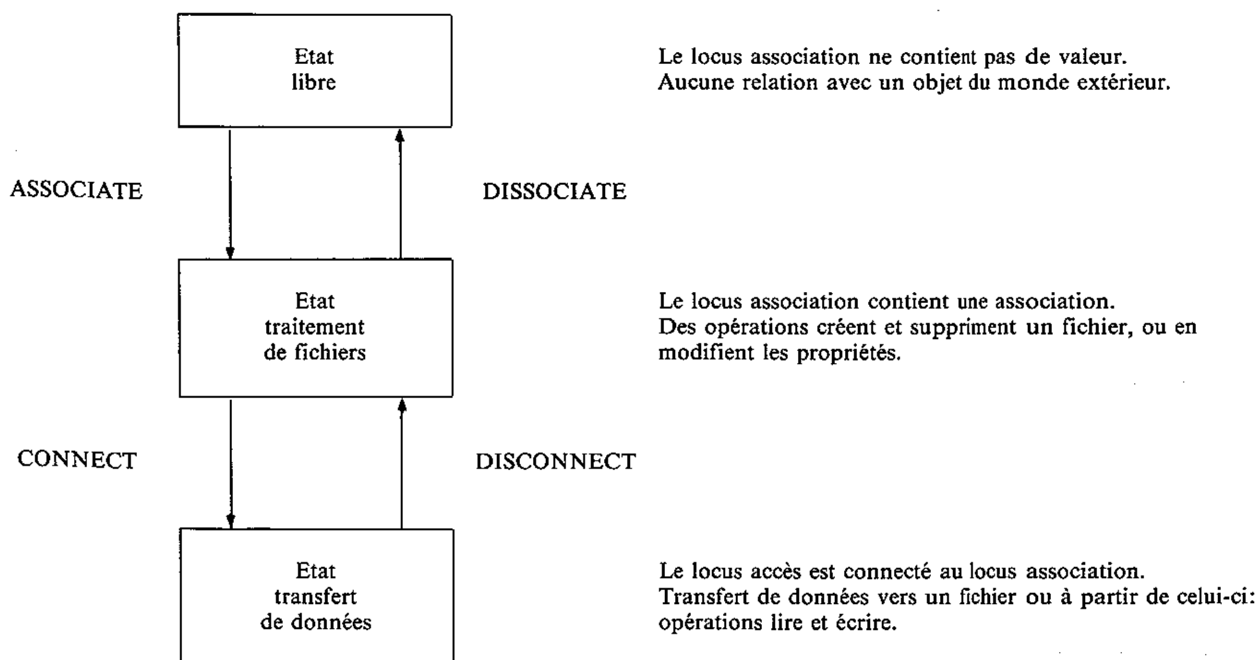
La valeur *primitive repère* doit donner une valeur repère **affectée.** La durée de vie du locus repéré ne doit pas être terminée.

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 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.-à-d. 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/ou 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.-à-d. 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/ou 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 rangée de valeurs; chaque élément de cette rangée se rapporte à un enregistrement du fichier. Le mode des éléments de cette rangée 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 opérations 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 opérations 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.

CHILL définit les attributs suivants:

- **usage** : indiquant pour quelle(s) opération(s) 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 OPÉRATIONS PRÉDÉFINIES POUR ENTRÉE-SORTIE

7.4.1 Généralités

Les opérations 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 opérations prédéfinies sont décrites dans les sections ci-après:

syntaxe:

<appel d'opération prédéfinie rendant valeur d'e/s> ::= (1)
    <appel d'opération prédéfinie attribut d'association> (1.1)
  | *<appel d'opération prédéfinie est associé>* (1.2)
  | *<appel d'opération prédéfinie attribut d'accès>* (1.3)
  | *<appel d'opération prédéfinie lire article>* (1.4)
  | *<appel d'opération prédéfinie obtenir texte>* (1.5)

<appel d'opération prédéfinie simple d'e/s> ::= (2)
    <appel d'opération prédéfinie dissocier> (2.1)
  | *<appel d'opération prédéfinie modification>* (2.2)
  | *<appel d'opération prédéfinie connecter>* (2.3)
  | *<appel d'opération prédéfinie déconnecter>* (2.4)
  | *<appel d'opération prédéfinie écrire article>* (2.5)
  | *<appel d'opération prédéfinie texte>* (2.6)
  | *<appel d'opération prédéfinie fixer texte>* (2.7)

<appel d'opération prédéfinie rendant locus d'e/s> ::= (3)
    <appel d'opération prédéfinie associer> (3.1)

conditions statiques:

Un *paramètre d'opération prédéfinie* dans une opération prédéfinie d'e/s qui est un *locus* *association*, *accès* ou *texte* doit être **repérable**.

7.4.2 Association avec un objet du monde extérieur

syntaxe :

<appel d'opération prédéfinie associer> ::=	(1)
ASSOCIATE (<locus association> [, *<liste de paramètres pour associer>*] *)*	(1.1)
<appel d'opération 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/ou 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 d'opération prédéfinie *ISASSOCIATED* est la *BOOL*-classe par dérivation. Le mode de l'appel d'opération prédéfinie *ASSOCIATE* est le mode du *locus association.*

La **régionalité** d'un appel d'opération 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.

exemples :

20.21	*ASSOCIATE (file_association,"DSK:RECORDS.DAT");*	(1.1)

7.4.3 Dissociation d'un objet du monde extérieur

syntaxe :

<appel d'opération 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.

exemples :

22.38	*DISSOCIATE (association);*	(1.1)

7.4.4 Accès aux attributs association

syntaxe :

<appel d'opération 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 d'opération prédéfinie attribut d'association* est la *BOOL*-classe par dérivation.

conditions dynamiques :

L'appel d'opération 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 d'opération 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/ou 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 d'opération 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 l'opération 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.-à-d. 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 **d'indice**, tandis que l'attribut **séquençable** doit être *TRUE* si le locus n'a pas de mode **d'indice**.

CONNECT (re)place le fichier qui est rattaché à l'association désignée, c.-à-d. 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.-à-d. 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.-à-d. 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.-à-d. 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 **d'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.-à-d. que le locus accès n'a pas de mode **d'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.-à-d. que le fichier peut être tronqué ou vidé par *CONNECT*.

Un locus accès qui n'a pas de mode d'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 **d'indice** si une *expression indice* est spécifiée; la classe de la valeur donnée par *l'expression indice* doit être **compatible** avec ce mode **d'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 **d'indice**;
- l'attribut **séquençable** de l'association est *FALSE* et le locus accès n'a pas de mode **d'indice**;
- *l'expression positionnnement* 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 **d'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 **d'indice** du locus accès est un mode intervalle et que *l'expression indice* donne une valeur extérieure aux bornes de ce mode intervalle.

L'exception *EMPTY* est causée si le **repère 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 d'opération 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 d'opération 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 repère 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.-à-d. *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.-à-d. *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 d'opération prédéfinie *GETASSOCIATION* est la *ASSOCIATION*-classe par repère. La **régionalité** d'un appel d'opération prédéfinie *GETASSOCIATION* est celle du *locus transfert*.

La classe d'un appel d'opération prédéfinie *OUTOFFILE* est la *BOOL*-classe par dérivation.

La classe d'un appel d'opération 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.

exemples:

21.47 *OUTOFFILE (infiles (FALSE))* (1.3)

7.4.9 Opérations de transfert de données

syntaxe:

<appel d'opération prédéfinie lire article> ::= (1)
READRECORD (<locus accès> [, *<expression indice>*] [, *<locus de lecture>*]) (1.1)

<appel d'opération 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 **d'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 opérations 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 **d'indice**, l'indice de **transfert** est l'indice **actuel** augmenté de 1; si le *locus accès* a un mode **d'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 **d'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 repéré 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 repère 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 d'opération 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 rangée lue dans le cas d'un mode rangée;
- 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 d'opération prédéfinie *READRECORD* est la M-classe par repère si le *locus de lecture* n'est pas spécifié, sinon c'est la S-classe par repère où S est le mode du *locus de lecture.*

La **régionalité** d'un appel d'opération 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 **d'indice** et doit être spécifiée si le *locus accès* a un mode **d'indice**. La classe de la valeur donnée par *l'expression indice* doit être **compatible** avec ce mode **d'indice**.

Le *locus de lecture* doit être **repérable**.

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 **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 d'opération prédéfinie *READRECORD* et *WRITERECORD* causent l'exception *NOTCONNECTED* si le *locus accès* n'est pas connecté à une association.

Les appels d'opération prédéfinie *READRECORD* ou *WRITERECORD* causent l'exception *RANGEFAIL* si le mode d'**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 d'opération 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 repérée dans les implémentations comme des caractères de commande de chariot ou de commande.

La manipulation d'un fichier de texte dans 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 opérations 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 INT (0:*L*), où *L* est la **longueur de texte** du mode de la valeur. Il est initialisé à 0 quand un locus texte est créé;
- **repère d'enregistrement de texte**: indique une valeur repère 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;
- **repère d'accès**: indique une valeur repère 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 d'opération 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>*],	(2.1)
<argument de format> [, *<liste d'e/s>*]	(2.2)
<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 chaîne>*	(7.2)
<argument de valeur> ::=	(8)
<expression discrète>	(8.1)
| *<expression chaîne>*	(8.2)

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 d'opération 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,* ou
- 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 d'opération 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** une tranche de chaîne égale 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'*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 la section 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 repère à la *liste d'e/s* (voir les sections 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 <u>chaîne</u>*, 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 <u>texte</u>* ou s'il l'est et son mode **accès** n'a pas de mode d'**indice** et doit être spécifié si le mode **accès** a un mode d'**indice**; la classe de la valeur livrée par l'*expression indice* doit être **compatible** avec ce mode d'**indice**.

Un *argument de texte* dans un appel d'opération prédéfinie *WRITETEXT* doit être un locus.

Un *locus <u>chaîne</u>* dans un *argument de texte* doit être **repérable**.

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*, ou si
- intervient une tentative d'affecter à l'**indice effectif** une valeur inférieure à 0 ou supérieure à la **longueur de texte**, ou si
- 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**, ou si
- une *clause d'e/s* est rencontrée et si l'*argument de texte* n'est pas un *locus <u>texte</u>*, ou si
- *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 d'opération 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.

exemples :

26.18 *WRITETEXT (sortie, «%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-pourcent>* | *<pourcent>* } (2.1)

<pourcent> ::= (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 «%3*D4*» équivaut à «%*D4*%*D4*%*D4*».

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.

exemples:

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 (2.1)

<qualificatif de conversion> ::= (3)
L / E / P <caractère> (3.1)

<largeur de clause> ::= (4)
{ *<chiffre>* }+ | *V* (4.1)

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 d'opération prédéfinie *READTEXT* transforme une chaîne qui est une représentation externe en une valeur CHILL. Une conversion dans un appel d'opération 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, une tranche de taille **largeur** commençant à la position d'**indice effective** est lue à partir d'un **enregistrement de texte** ou écrite 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 la tranche (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 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 la plus longue tranche de caractères qui commence à l'**indice effectif** et qui est constituée 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 livrée 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).

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 mis au rebut 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) *booléens*

Les valeurs booléennes sont représentées lexicalement comme une *représentation textuelle 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 représentations textuelles de nom simple **spéciales**). Le *code de conversion* suivant est disponible: *C*.

c) *caractères*

Les valeurs de caractère sont représentées lexicalement comme des chaînes de longueur *1*. Le *code de conversion* suivant est disponible: *C*.

d) *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*.

e) *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 appartiennent à l'ensemble de représentations externes associées au mode intervalle.

f) *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 (la tranche) 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*.

g) *chaînes de bits*

Les valeurs de chaîne de bits 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**.

conditions dynamiques:

L'exception *TEXTFAIL* se produit si:

- dans *READTEXT*, l'**enregistrement de texte** ne contient pas de tranche de chaîne commençant à l'**indice effectif** qui peut (après enlèvement ou omission des caractères de remplissage, voir ci-dessus) être interprétée 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**), ou
- dans *WRITETEXT*, une tranche de chaîne qui est la représentation externe de l'*argument de valeur* actuel ne peut pas être transférée à l'**enregistrement de texte** commençant à l'**indice effectif**, ou
- 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**, ou
- 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.

exemples:

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: largeur de clause espaces sont insérés ou omis.

>: omettre à droite: l'**indice effectif** est déplacé vers la droite de largeur de clause positions.

<: omettre à gauche: l'**indice effectif** est déplacé vers la gauche de largeur de clause positions.

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**, ou si
- 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.**

exemples:

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.

exemples:

26.21 / (1.1)

7.5.8 Accès aux attributs d'un locus texte

syntaxe:

```
<appel d'opération 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 d'opération 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 le **repère d'enregistrement de texte** de *locus texte.*

GETTEXTINDEX renvoie l'**indice effectif** du *locus texte.*

GETTEXTACCESS renvoie le **repère 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 un repère pour le locus donné par le *locus chaîne de caractères* dans le **repère 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 un repère du locus donné par le *locus accès* dans le **repère accès** du *locus texte*.

propriétés statiques:

La classe de l'appel d'opération prédéfinie *GETTEXTRECORD* est la M-classe par **repère**, où M est le mode **enregistrement de texte** du *locus texte*.

La classe de l'appel d'opération prédéfinie *GETTEXTINDEX* est la *INT*-classe par dérivation.

La classe de l'opération prédéfinie *GETTEXTACCESS* est la M-classe par repère, où M est le mode **accès** du *locus texte*.

La classe de l'appel de l'opération prédéfinie *EOLN* est la *BOOL*-classe par dérivation.

Un appel d'opération prédéfinie *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*.

exemples:

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 causer.

Quand une exception est causée, elle peut être traitée, c.-à-d. 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.-à-d. 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 { *<choix d'exceptions>* }* **[ELSE** *<liste d'énoncés d'action>*] **END**	(1.1)
<choix d'exceptions> ::=	(2)
(*<liste d'exceptions>*) : *<liste d'énoncés d'action>*	(2.1)

sémantique:

Un filet est entamé s'il convient pour une exception E conformément à la section 8.3. Si E est mentionné dans une *liste d'exceptions* dans un *choix d'exceptions* 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.

exemples:

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 parenthésés, le filet qui convient (si présent) est le filet qui convient pour l'action, le module ou la région parenthésés 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;

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; cependant, dans ce cas, un filet défini par l'implémentation peut être utilisé (voir la section 13.4);

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 tranfert 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). 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. 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. A cause de l'action de temporisation cyclique et de l'imbrication des actions de temporisation, plusieurs temporisations peuvent être associées à un même processus.

Une interruption se produit quand un processus est **temporisable** et qu'une au moins des temporisations associées a expiré. L'apparition 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 différé, il devient réactivé.

Les temporisations cessent aussi d'exister quand la commande quitte l'action de temporisation qui les a déclenchées.

9.3.1 Action de temporisation relative

syntaxe:

<action de temporisation relative> ::=	(1)
AFTER *<valeur primitive durée>* [**DELAY**] **IN**	(1.1)
<liste d'énoncés d'action> *<filet de temporisation>* **END**	
<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** n'est pas spécifié, la temporisation est déclenchée avant que la *liste d'énoncés d'action* soit entamée, sinon elle est déclenchée quand le processus d'exécution devient **temporisable** au point d'exécution spécifié par l'*énoncé d'action* dans 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 OPÉRATIONS PRÉDÉFINIES POUR LE TEMPS

syntaxe:

<appel d'opération prédéfinie de valeur temps> ::=	(1)
<appel d'opération prédéfinie de durée>	(1.1)
| *<appel d'opération 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 Opérations prédéfinies de durée

syntaxe:

<appel d'opération 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 d'opération 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.

propriétés statiques:

La classe d'un appel d'opération 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 Opération prédéfinie de temps absolu

syntaxe:

<appel d'opération 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 d'opération prédéfinie *ABSTIME* livre une valeur de temps absolu désignant le moment dans le calendrier grégorien indiqué dans la liste des paramètres. 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 d'opération 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 d'opération prédéfinie de temporisation

syntaxe:

<appel d'opération 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.

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.

conditions statiques :

Tous les locus entiers spécifiés doivent être **repérables** et leur mode ne peut 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, mettre en attente* et *choisir, bloc début-fin, module, région, module de spec, région de spec, contexte, recevoir et choisir, définition de procédure* et *définition de processus* déterminent la structure du programme, c-à-d. 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 à choisir* dans une *action de cas*;
  - la *liste d'énoncés d'action* dans un *événement à choisir* dans une *action mettre en attente et choisir*;
  - 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 un *tampon à choisir* ou dans un *signal à choisir*, y compris une *définition* ou *liste de définitions* après **IN**;
  - la *liste d'énoncés d'action* après **ELSE** dans une *action conditionnelle, de cas, recevoir et choisir* ou un *filet*;
  - le *choix d'exceptions* 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 *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.*
- 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 englobée (voir la section 10.2) par un groupe interne.

Un groupe influence la portée de chacun des noms créés dans son domaine.

Des noms peuvent être créés par des *définitions*:

- Une *définition* qui apparaît dans la *liste 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 *définition* qui apparaît dans un *mode ensemble* crée un nom dans le domaine qui englobe immédiatement le *mode ensemble.*
- Une *définition* qui apparaît dans la *liste 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 *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 *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 *définition* de la *liste de définitions* d'un *tampon à choisir* ou d'un *signal à choisir* crée un nom, respectivement, dans le domaine du bloc du *tampon à choisir* ou du *signal à choisir* correspondants.
- Une *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 la section 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 la section 12.2.2).

Un nom a une certaine portée, c.-à-d. 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.-à-d. 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 neumode>* (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)
  | *<vide>* ; (12.7)

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 de loc-identité, 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 de loc-identité, 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'une initialisation domaniale, un énoncé de loc-identité, 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 les sections 6.5, 6.6, 6.8, 6.9);
- sinon, s'il s'agit d'une procédure, la commande est renvoyée au point d'appel (voir la section 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 la section 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 de cas* ou d'une *action mettre en attente et choisir*, 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 de cas* ou l'*action mettre en attente et choisir*, et seulement par ce groupe.
- Si le domaine est la *liste d'énoncés d'action* ou un *tampon à choisir* ou un *signal à choisir*, ou la *liste d'énoncés d'action* suivant **ELSE** dans une *action recevoir tampon et choisir* ou une *action recevoir signal et choisir*, il est alors immédiatement englobé par le groupe dans le domaine duquel se trouve placé l'*action recevoir tampon et choisir* ou l'*action recevoir signal et choisir*, et seulement par ce groupe.
- Si le domaine est la *liste d'énoncés d'action* d'une liste de *choix 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 un *choix d'exceptions* 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.-à-d. 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.-à-d. 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 tampon à choisir, domaine signal à choisir: le choix est exécuté à la réception d'une valeur tampon ou d'un signal.
- Domaine choix d'exceptions: le choix d'exceptions 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 *définition* est une **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* et n'est pas la *définition* d'un nom d'**élément d'ensemble**;

  sinon, c'est une *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 les sections 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)

<définition> : *<définition de procédure>*
[*<filet>*] [*<représentation textuelle de nom simple>*]; (1.1)

<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 de définitions> *<spec de paramètre>* (4.1)

<liste d'attributs de procédure> ::= (5)

[*<généralité>*] [**RECURSIVE**] (5.1)

<généralité> ::= (6)

GENERAL (6.1)
| **SIMPLE** (6.2)
| **INLINE** (6.3)

syntaxe dérivée :

Un *paramètre formel* où la *liste de définitions* comporte plus d'une *définition* est dérivé de plusieurs occurrences de *paramètre formel* séparées par des virgules, une pour chaque *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.-à-d. 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 **in-situ (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** peuvent être spécifiées comme (mutuellement) **récursives.** Quand aucun attribut de procédure n'est spécifié, un défaut de l'implémentation s'appliquera.

Une procédure peut retourner une valeur ou elle peut retourner un locus (indiqué par l'attribut **LOC** dans la spec de résultat).

La *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 *<définition>* *<mode>* ::= ; *<paramètre effectif>* ;

était rencontrée pour les *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 **indéfinie**. 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 **repérables** peuvent être passés de cette manière. Tout se passe comme si, au point d'entrée de la procédure, la déclaration de loc-identité:

DCL *<définition>* *<mode>*

LOC [**DYNAMIC**] ::= *<paramètre effectif>* ;

était rencontrée pour les *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 la section 6.8). Si l'attribut **NONREF** n'est pas donné dans la *spec de résultat,* le *locus* doit être **repérable**. 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 la section 4.2.11), ni comme appel de procédure rendant valeur (voir la section 5.2.12), mais seulement comme action appeler (section 6.7).

propriétés statiques:

Une *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 **in-situ**, 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'une région, sa **généralité** est **simple**.
- Il a une **récursivité** qui est **récursive** si **RECURSIVE** est spécifié; sinon, un défaut défini par l'implémentation spécifie soit **récursive,** soit **non récursive**. Cependant, si la **généralité** est **in-situ**, ou si le nom de **procédure** est **critique** (voir la section 11.2.1) la **récursivité** est **non récursive**.

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>* *)*] [**RECURSIVE**]

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 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 de **loc-identité**. De tels noms de **locus** ou noms de **loc-identité** sont **repérables**.

conditions statiques:

Si un nom de **procédure** est **intrarégional** (voir la section 11.2.2), sa définition de procédure ne doit pas spécifier **GENERAL**.

Si un nom de **procédure** est une procédure **critique** (voir la section 11.2.1), sa définition ne peut spécifier ni **GENERAL** ni **RECURSIVE**.

Aucune définition de procédure ne peut spécifier à la fois **INLINE** et **RECURSIVE**.

Si on le spécifie, la *représentation textuelle de nom simple* doit être égale à la *définition* devant la *définition de procédure.*

Seulement si **LOC** est spécifié dans la *spec de paramètre* ou *spec de résultat*, le mode contenu peut avoir la **propriété de non-valeur**.

Tous les noms d'exception mentionnés dans la *liste d'exceptions* doivent être différents.

exemples:

1.4 *add:*
PROC *(i, j INT)* **RETURNS** *(INT)* **EXCEPTIONS** *(OVERFLOW)*;
RESULT *i + j*;
END *add*; (1.1)

10.5 DÉFINITIONS DE PROCESSUS

syntaxe:

<énoncé de définition de processus> ::= (1)
<définition> : *<définition de processus>*
[*<filet>*] [*<représentation textuelle de nom simple>*]; (1.1)

<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 concurrente à partir de différents endroits du programme (voir le chapitre 11).

propriétés statiques:

Une *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 *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 *représentation textuelle de nom simple* doit être égale à la *représentation textuelle de nom de la 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 la section 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.**

exemples:

```
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> ] [ <définition> :]
      MODULE [ BODY ] <corps de module> END
      [ <filet> ] [ <représentation textuelle de nom simple> ];   (1.1)
    | <modulion distant>                                      (1.2)
```

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 représentations textuelles de nom; il n'influence pas la durée de vie des locus créés localement.

Les règles de visibilité détaillées pour les modules sont données dans la section 12.2.

propriétés statiques:

Une *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.-à-d. en excluant la *liste de contextes* et la *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 *représentation textuelle de nom simple* doit être égale à la représentation textuelle de nom de la *définition.*

Un *modulion distant* dans un *module* doit repérer 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> ] [ <définition> :]
      REGION [ BODY ] <corps de région> END                        (1.1)
      [ <filet> ] [ <représentation textuelle de nom simple> ];
    | <modulion distant>                                           (1.2)
```

sémantique :

Une région est un moyen de réaliser l'exclusion mutuelle, pour accéder aux objets informatifs créés localement, lors de l'exécution concurrente des processus (voir le chapitre 11). Elle détermine la visibilité des noms créés localement de la même manière qu'un module.

propriétés statiques :

Une *définition* dans une *région* définit un nom de **région.** Elle est rattachée à la région (considérée comme un modulion, c.-à-d. en excluant la *liste de contextes* et la *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 *représentation textuelle de nom simple* doit être égale à la représentation textuelle de nom de la *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 repérer une *région.*

exemples :

Voir 13.1 - 13.28.

10.8 PROGRAMME

syntaxe :

```
<programme> ::=                                                    (1)
      { <module> | <module de spec> | <région> | <région de spec> }+  (1.1)
```

sémantique :

Les programmes consistent en une liste de modules ou de régions, englobés par une définition de processus imaginaire le plus externe.

Les définitions de noms prédéfinis par CHILL (voir l'Appendice C.2) et les opérations 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 la section 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 d'opération 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 d'opération prédéfinie *GETSTACK* s'achève lorsque le bloc immédiatement englobant se termine.

La durée de vie d'un locus créé par un appel d'opération 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 d'opération prédéfinie *TERMINATE* est exécuté sur une valeur repère **affectée** repérant le locus.

La durée de vie d'un accès, créé dans une déclaration de loc-identité est le bloc englobant du plus près la déclaration de loc-identité.

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 une *tranche 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 la *tranche de chaîne* sont **constants**.
- Un *élément de rangée* où le *locus rangée* est **statique** et l'expression est **constante**.
- Une *tranche de rangée* où le *locus rangée* est **statique** et soit l'*élément inférieur* et l'*élément supérieur*, soit le *premier élément* et la *taille de tranche* 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 la section 10.10.1). 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 repéré.

10.10.1 Fragments distants

syntaxe:

<modulion distant> ::=	(1)
[*<représentation textuelle de nom simple>* :]	
REMOTE *<indicateur de fragment>*;	(1.1)
<spec distante> ::=	(2)
[*<représentation textuelle 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 repère de texte>* (5.2)
| *<vide>* (5.3)

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* et *modules de contexte* 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 de chaîne de caractères,* celui-ci est utilisé pour extraire le texte source.
- Si l'*indicateur de fragment* contient un *nom de repère 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,* est équivalent au programme formé en remplaçant chaque 1) *modulion distant,* 2) *spec 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 repéré 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, 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,* 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.*

Lorsque le texte source mentionné par l'*indicateur de fragment* dans un *modulion distant* commence par une *définition,* le *modulion distant* doit alors commencer par une *représentation textuelle de nom simple* qui est la représentation textuelle du nom de cette *définition.*

Lorsque le texte source mentionné par l'*indicateur de fragment* dans une *spec distante* commence par une *représentation textuelle de nom simple,* la *spec distante* doit alors commencer par la *représentation textuelle de nom simple.*

exemples:

25.9 *stack*: **REMOTE** *"example 27 ou 28";* (1.1)
25.9 *"example 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>*] [*<représentation textuelle de nom simple>* :] **SPEC MODULE** *<corps de module de spec>* **END** [*<représentation textuelle de nom simple>*];	(2.1)
<spec de module> ::=	(3)
[*<liste de contextes>*] *<représentation textuelle de nom simple>* : **MODULE SPEC** *<corps de module de spec>* **END** [*<représentation textuelle 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>*] [*<représentation textuelle de nom simple>* :] **SPEC REGION** *<corps de région de spec>* **END** [*<représentation textuelle de nom simple>*];	(5.1)
<spec de région> ::=	(6)
[*<liste de contextes>*] *<représentation textuelle de nom simple>* : **REGION SPEC** *<corps de région de spec>* **END** [*<représentation textuelle 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 *représentations textuelles 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 *représentation textuelle de nom OP ! NS* **visible** dans le domaine d'une 1) *spec de module,* 2) *spec de région* et reliée ici à une **quasi-***définition* et qui est octroyée dans un domaine **réel** tel *NP ! NS,* un énoncé (virtuel) d'octroi avec la même *représentation textuelle* **ancienne** *OP ! NS* et la **nouvelle** *représentation textuelle 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 *représentation textuelle de nom simple* optionnelle suivant **END** ne peut être présente que si *la représentation textuelle de nom simple* optionnelle avant **SPEC** est présente. Quand les deux sont présentes, elles doivent avoir des représentations textuelles 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** 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 *représentations textuelles de nom simple* qui les précèdent doivent avoir des représentations textuelles 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*.

exemples :

```
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)

24.1    CONTEXT
            count: PROC (ROW CHARS (max) IN,
            ARRAY ('A':'Z') INT OUT) END;
        FOR                                                 (8.1)
```

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 de loc-identité>* (3.2)

<quasi-déclaration de locus> ::= (4)
  <liste de définitions> *<mode>* [**STATIC**] (4.1)

<quasi-déclaration de loc-identité> ::= (5)
  <liste de définitions> *<mode>* **LOC** [**NONREF**] [**DYNAMIC**] (5.1)

<quasi-énoncé définissant> ::= (6)
  <énoncé de définition de synmode> (6.1)
  | *<énoncé de définition de neumode>* (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)
  | *<vide>* ; (6.8)

<*quasi-énoncé de définition de synonyme*> ::= (7)
SYN <*quasi-définition de synonyme*> {, <*quasi-définition de synonyme*> }*; (7.1)

<*quasi-définition de synonyme*> ::= (8)
<*liste de définitions*> { <*mode*> = [<*valeur constante*>] | [<*mode*>] = <*expression littérale*> } (8.1)

<*quasi-énoncé de définition de procédure*> ::= (9)
<*définition*> : **PROC** ([<*quasi-liste de paramètres formels*>])
[<*spec de résultat*>] [**EXCEPTIONS** (<*liste d'exceptions*>)]
<*liste d'attributs de procédure*> **END** [<*représentation textuelle de nom simple*>]; (9.1)

<*quasi-liste de paramètres formels*> ::= (10)
<*quasi-paramètre formel*> {, <*quasi-paramètre formel*> }* (10.1)

<*quasi-paramètre formel*> ::= (11)
<*représentation textuelle de nom simple*> {, <*représentation textuelle de nom simple*> }*
<*spec de paramètre*> (11.1)

<*quasi-énoncé de définition de processus*> ::= (12)
<*définition*> : **PROCESS** ([<*quasi-liste de paramètres formels*>]) **END**
[<*représentation textuelle 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)
<*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. Ces spécifications sont redondantes mais des quasi-énoncés peuvent être 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 CHILL, il n'existe pas de **quasi**-*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 *définition* dans une *quasi-déclaration de loc-identité* est **repérable** 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* 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* 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-définitions et définitions

Deux *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 repère, **nulle ou toute**, et si celle qui est quasi est **littérale**, l'autre doit l'être aussi;
- si elles sont des noms d'élément d'ensemble, les modes ensemble attachés doivent être **semblables**;
- si elles sont des noms de **neumode** ou de **synmode**, leurs modes doivent être **semblables**;
- si elles sont des noms de **locus** ou des noms de **loc-identité**, elles doivent avoir la même **régionalité**, elles doivent être, ou ne pas être, toutes deux **repérables**, elles doivent être ou ne pas être toutes deux **statiques** 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 *représentations textuelles 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 *représentation textuelle 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**-*définition*, elle doit être aussi **liée** à une *définition* qui n'est pas une **quasi**-*définition* et de plus:

  soit une *représentation textuelle de nom* **liée** à une **quasi**-*définition* QD et **liée** aussi à une *définition* **réelle** RD dans le domaine R; dans ces conditions:

  1) QD et RD doivent **correspondre** comme défini plus haut, et

  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 *représentation textuelle de nom* dans un domaine **réel** R est **liée** à une **quasi**-*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 *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**-*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**.

Si une *représentation textuelle de nom* dans un domaine **réel** R est **liée** à une *définition* **réelle** qui est englobée dans le groupe d'un *module* ou d'une *région* qui sont indiqués par un *modulion distant* immédiatement englobé dans R, elle doit aussi être **liée** à une **quasi**-*définition* qui est englobée dans le groupe de R (c'est-à-dire entourée par un modulion de spec. Informellement, si l'implémentation octroie, il doit en être de même pour l'interface).

Pour chaque *représentation textuelle 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 *définition* non entourée par Q, il doit y avoir une *représentation textuelle 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 *définition* (informellement, si l'interface saisit, il doit en être de même pour l'implémentation).

- Si deux *représentations textuelles de nom* sont **liées** à la même 1) *définition* **réelle**, 2) **quasi**-*définition* dans un domaine, les deux *représentations textuelles de nom* doivent être **liées** à la même 1) **quasi**-*définition*, 2) *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 *région* **correspondante**.

11 EXÉCUTION CONCURRENTE

11.1 LES PROCESSUS ET LEURS DÉFINITIONS

Un processus est l'exécution séquentielle d'une série d'énoncés. Il peut être exécuté en parallèle avec d'autres processus. Le comportement d'un processus est décrit par une définition de processus (voir la section 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 la section 5.2.14). Il devient actif (c.-à-d. en exécution) et il est considéré être exécuté en parallèle avec d'autres processus. 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 exemplaire, 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 la section 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 la section 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 processus du programme seront terminés.

Un processus est, au niveau du programme CHILL, toujours dans l'un des deux états: il est soit actif (c.-à-d. en exécution) ou en attente (c.-à-d. attendant une condition à satisfaire). La transition d'actif à en attente est appelée la mise en attente du processus, la transition d'en attente à actif est appelée la réactivation du processus.

11.2 EXCLUSION MUTUELLE ET RÉGIONS

11.2.1 Généralités

Les régions (voir la section 10.7) sont un moyen de fournir aux processus un accès mutuellement exclusif aux locus déclarés à l'intérieur d'elles. Les conditions de contexte statiques (voir la section 11.2.2) sont telles que des accès par un processus (qui n'est pas le processus imaginaire le plus externe) aux locus déclarés dans une région ne peuvent se faire qu'en appelant des procédures qui sont définies à l'intérieur de la région et octroyées par la région.

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.

Une région est dite être **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.

La région sera verrouillée (pour empêcher l'exécution concurrente) si:

- La région est entamée (à noter que parce que les régions ne sont pas englobées par un bloc, plusieurs essais pour entamer la région ne peuvent être réalisés en parallèle).
- Une procédure **critique** de la région est appelée.
- Un processus, mis en attente sur la région, est réactivé.

La région sera libérée et devient à nouveau libre, si:

- La région est quittée.
- Une procédure **critique** revient.
- Une procédure **critique** exécute une action qui cause la mise en attente du processus exécutant (voir la section 11.3). Dans le cas d'appels de procédures **critiques** imbriquées dynamiquement, seule la région verrouillée le plus tard 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.

Si, pendant qu'une région est verrouillée, un processus essaye d'appeler une de ses procédures **critiques** ou qu'un processus mis en attente dans la région est réactivé, ce processus est suspendu jusqu'à ce que la région soit libérée. (A noter que le processus qui essaye reste actif dans le sens de CHILL.)

Quand une région est libérée et que plus d'un processus a été suspendu en essayant d'appeler une de ses procédures **critiques** ou d'être réactivé dans une de ses procédures **critiques**, un processus seulement sera sélectionné pour verrouiller la 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 la section 10.4);
- les procédures **critiques** ne sont ni **générales**, ni **récursives** (voir la section 10.4).

Un *locus* et un *appel de procédure* peuvent être **intrarégionaux** ou **extrarégionaux**. Une *valeur* a une **régionalité** qui est **intrarégionale** ou **extrarégionale** ou **nulle**. Ces propriétés se définissent comme suit:

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 loc-identité*, dans la déclaration de laquelle le *locus* est **intrarégional** ou qui est défini dans le *paramètre formel* d'une procédure **intrarégionale**,
     - soit un *nom d'énumération de locus*, dont le *locus rangée* 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 un *repère lié dérepéré*, contenant une *valeur primitive repère lié* qui est **intrarégionale**.
   - C'est un *repère libre dérepéré*, contenant une *valeur primitive repère libre* qui est **intrarégionale**.
   - C'est un *descripteur dérepéré*, contenant une *valeur primitive descripteur* qui est **intrarégionale**.
   - C'est un *élément de rangée* ou une *tranche de rangée*, contenant un *locus rangée* qui est **intrarégional**.
   - C'est un *élément de chaîne* ou une *tranche 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 d'opération 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 repère 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 repérer**, alors la **régionalité** est **nulle**; sinon, la *valeur* est un *opérande-6* (et a la **propriété de repérer**) ou une *expression conditionnelle*:

Si c'est une *valeur primitive*, alors:

- Si c'est un *contenu de locus* qui est un *locus*, alors, c'est celle du *locus*.
- Si c'est un *nom de valeur*, alors:
  - si c'est un *nom de synonyme*, alors c'est celui de la *valeur constante* dans sa définition;
  - si c'est un *nom de valeur faire-avec*, alors c'est celui de la *valeur primitive structure* de l'action faire associée;
  - si c'est un *nom de valeur reçue*, alors elle est **extrarégionale**.
- Si c'est un *multiplet*, alors si l'une de ses occurrences de *valeur* a une **régionalité** non **nulle**, alors c'est celle de cette *valeur* (peu importe le choix qui est fait, voir la section 5.2.5, conditions statiques); sinon, il est **nul**.
- Si c'est une *valeur élément de rangée*, ou une *valeur tranche de rangée*, alors c'est celle de la *valeur primitive rangée* qu'elle contient.
- Si c'est une *valeur champ de structure*, alors c'est celle de la *valeur primitive structure* qu'elle contient.
- Si c'est une *conversion d'expression*, alors c'est celle de l'*expression* qu'elle contient.
- Si c'est un *appel de procédure rendant valeur*, alors c'est celle de l'*appel de procédure* qu'elle contient.
- Si c'est un *appel d'opération prédéfinie rendant valeur* que la définition CHILL ou l'implémentation spécifie comme étant **intrarégional** ou **extrarégional**.

Si c'est un *locus repéré*, c'est celui du *locus* qu'elle contient.

Si c'est une *expression recevoir*, alors, elle est **extrarégionale**.

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 la section 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.-à-d. 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.**

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 PROCESSUS

Un processus actif peut être mis en attente en exécutant (évaluant) l'une des actions (expressions) suivantes:

Action mettre en attente (voir la section 6.16).

Action mettre en attente et choisir (voir la section 6.17).

Expression recevoir (voir la section 5.3.9).

Action recevoir signal et choisir (voir la section 6.19.2).

Action recevoir tampon et choisir (voir la section 6.19.3).

Action envoyer tampon (voir la section 6.18.3).

Quand un processus est mis en attente pendant que le contrôle réside dans une procédure **critique**, la région associée sera libérée. Le contexte dynamique du processus sera retenu jusqu'à ce que celui-ci soit réactivé. Le processus essaye alors de verrouiller à nouveau la région, ce qui peut provoquer son interruption.

11.4 RÉACTIVATION D'UN PROCESSUS

Un processus en attente peut être réactivé lorsqu'il est soumis à une supervision temporelle et qu'une interruption se produit (voir le chapitre 9). Il peut également être réactivé si un autre processus exécute (évalue) une des actions (expressions) ci-après:

- **Action continuer** (voir la section 6.15).
- **Action envoyer signal** (voir la section 6.18.2).
- **Action envoyer tampon** (voir la section 6.18.3).
- **Expression recevoir** (voir la section 5.3.9).
- **Action recevoir tampon et choisir** (voir la section 6.19.3).

Quand un processus, qui a verrouillé une région, réactive un autre processus, il reste actif, c'est-à-dire qu'il ne libérera pas la région à cet endroit.

11.5 ÉNONCÉS DE DÉFINITION DE SIGNAL

syntaxe:

<énoncé de définition de signal> ::= (1)
  SIGNAL *<définition de signal>* {, *<définition de signal>* }*; (1.1)

<définition de signal> ::= (2)
  <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 et choisir, les valeurs sont gardées jusqu'à ce qu'un processus les reçoive.

propriétés statiques:

Une *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**.

exemples:

15.27 **SIGNAL** *initiate = (INSTANCE),*
  terminate; (1.1)

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 un composant, ou un sous-composant, etc. qui est un mode **protégé**.

Définition

Un mode a la **propriété de protection** si et seulement si c'est:

- un mode rangée ayant un mode des **éléments** qui a la **propriété de protection**;
- un mode structure dont au moins un des modes **de champ** a la **propriété de protection**, où le champ n'est pas un champ **marqueur** 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 rangée;
- un mode structure **variable paramétrable**.

12.1.1.3 Propriété de repérer

Informel

Un mode a la **propriété de repérer** si c'est un mode repère ou s'il contient un composant ou un sous-composant, etc. qui est un mode repère.

Définition

Un mode a la **propriété de repérer** si et seulement si c'est:

- un mode repère;
- un mode rangée avec un mode des **éléments** qui a la **propriété de repérer**;
- un mode structure dont au moins un des modes de **champ** a la **propriété de repérer**.

12.1.1.4 Propriété de marquage et de paramétrage

Informel

Un mode a la **propriété de marquage et de paramétrage** si c'est un mode structure **paramétré avec marqueurs** ou s'il contient un composant ou un sous-composant, etc. qui est un mode structure **paramétré avec marqueurs**.

Définition

Un mode a la **propriété de marquage et de paramétrage** si et seulement si c'est:

- un mode rangée ayant un mode des **éléments** qui a la **propriété de marquage et de paramétrage**;
- un mode structure dont au moins un des modes de **champ** a la **propriété de marquage et de paramétrage**;
- un mode structure **paramétré avec marqueurs**.

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 rangée ayant un mode des **éléments** qui a la **propriété de non-valeur**;
- un mode structure dont au moins un des modes de **champ** a la **propriété de non-valeur**.

12.1.1.6 Mode racine

Tout mode M a un mode **racine** défini de la façon suivante:

- M, si M n'est pas un mode intervalle;
- le mode **parent** de M, si M est un mode intervalle.

Toute M-classe par valeur ou M-classe par définition a un mode **racine** qui est le mode **racine** de M.

12.1.1.7 Classe résultante

Etant donné deux classes **compatibles** (voir la section 12.1.2.16), qui sont soit la classe **toute**, soit une M-classe par valeur, soit une M-classe par dérivation, où M est un mode discret, un mode ensembliste, ou un mode chaîne, la **classe résultante** est définie en fonction de la notion de mode **résultant** R de M et N et le mode **racine** P 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.

La **classe résultante** se définit ainsi:

- 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 de la classe **toute** est la P-classe par valeur;
- la **classe résultante** de la M-classe par dérivation et de 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** de la classe **toute** et de la classe **toute** est la classe **toute**.

Etant donné une liste C_i de classes **compatibles** deux à deux ($i = 1, \ldots, n$), 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 ($i = 1, \ldots, n - 1$) 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 sections 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 ce document pour définir les conditions statiques.

Les relations de compatibilité elles-mêmes sont définies en termes d'autres relations qui, dans le présent chapitre, 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.-à-d. 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.-à-d. 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 sections 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 sections appropriées).

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 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 représentation textuelle 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 qui ont des modes **parents similaires;**
- l'un est un mode intervalle dont le mode **parent** est **similaire** à l'autre;
- ce sont des modes ensembliste dont les modes **primitifs** sont **équivalents;**
- ce sont des modes repère lié tels que leurs modes **repérés** sont **équivalents;**

- ce sont des modes repère libre;
- ce sont des modes descripteur dont les modes **repérés 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 exemplaire;
- 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 **des éléments 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 **d'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 qui sont tous les deux des modes chaîne **de bits** ou des modes chaîne **de caractères**;
- ce sont des modes rangée du type suivant:
  1) leurs modes **d'indice** sont **v-équivalents**,
  2) leurs modes **des éléments** 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) mode(s) est (sont) dynamique(s). 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 est (sont) dynamique(s). Une détection d'anomalie donnera lieu à l'exception *TAGFAIL*.

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, l'autre mode doit aussi être un mode intervalle et les deux **bornes supérieures** doivent être égales ainsi que les deux **bornes inférieures**;

- 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 de **protection**, l'autre doit l'avoir aussi, et:

- si les deux sont des modes repère lié, leurs modes **repérés** doivent être **l-équivalents**;
- si les deux sont des modes descripteur, leurs modes **repérés originels** doivent être **l-équivalents**;
- si les deux sont des modes rangée, leurs modes **des éléments** 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 *choix de champs* 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 *choix de champs* et des *champs à choisir* 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**.
- *Choix de champs*
  1) Les deux *choix de champs* ont des *listes de marqueurs* ou les deux n'en ont pas. Dans le premier cas, les *listes de marqueurs* doivent avoir le même nombre de noms de **champs marqueurs** et les noms de **champs marqueurs** correspondants (par position) doivent dénoter des *champs fixes* correspondants.
  2) Les deux doivent avoir le même nombre de *champs à choisir* et les *champs à choisir* 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**.
- *Champs à choisir*
  1) Les deux *champs à choisir* 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 *champs à choisir* 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>)*

La section 3.12.5 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 ayant des **bornes supérieures** égales et des **bornes inférieures** égales;
- ce sont des modes ensembliste dont les modes **primitifs** sont **semblables**;
- ce sont des modes repère lié dont les modes **repère** sont **semblables**;
- ce sont des modes repère libre;
- ce sont des modes descripteur dont les modes **repérés 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 exemplaire;
- ce sont des modes événement qui ont tous deux la même **longueur d'événement** ou n'en 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 **des éléments de tampon** qui sont **semblables**;
- ce sont des modes association;
- ce sont des modes accès du type suivant:
  1) tous deux ont des modes **d'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) tous deux sont des modes chaîne **de bits** ou des modes chaîne **de caractères,**
  2) ils ont la même **longueur de chaîne,**
  3) ils sont tous deux des modes chaîne **fixe** ou **variable;**
- ce sont des modes rangée du type suivant:
  1) leurs modes **d'indice** sont **semblables,**
  2) leurs modes des **éléments** 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 choix de *champs* qui sont **semblables.**

La relation **semblable** est définie récursivement pour, respectivement, des *champs fixes,* des *champs récurrents,* des *choix de champs* et des *champs à choisir* 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 *représentation textuelle de nom.*
- *Choix de champs*
  1) Les *choix de champs* doivent avoir tous deux des *listes de marqueurs* ou ne pas en avoir. Dans le premier cas, les *listes de marqueurs* doivent avoir le même nombre de noms de **champ marqueur** et les noms de **champ marqueur** correspondants (par position) doivent dénoter des *champs fixes* correspondants.
  2) Tous deux doivent avoir le même nombre de *champs à choisir* et les *champs à choisir* 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.**
- *Champs à choisir*
  1) Les deux *champs à choisir* 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 *champs à choisir* 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**-neumode doit représenter au plus un neumode **réel.** Cela s'établit ainsi: quand une *représentation textuelle de nom* est **liée** à la fois à une définition **réelle** et à une **quasi**-*définition,* tous les neumodes 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 *représentation textuelle de nom* **liée** dans un domaine R à la fois à une définition **réelle** et à une **quasi**-*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 d'**élément d'ensemble**, les modes des modes **ensemble** attachés sont **appariés par la nouveauté** dans R;
- si ce sont des noms de **locus** ou de **loc-identité**, 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 repère lié, leurs modes **repérés** sont **appariés par la nouveauté** dans R;
- si ce sont des modes descripteur, leurs modes **repérés 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 des **éléments de tampon** sont **appariés par la nouveauté** dans R;
- si ce sont des modes accès, leurs modes d'**indice**, si présents, et modes **enregistrement**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des modes texte, leurs modes d'**indice**, si présents, sont **appariés par la nouveauté** dans R;
- si ce sont des modes rangée, leurs modes d'**indice** et modes **des éléments** sont **appariés par la nouveauté** dans R;
- si ce sont des modes structure, leurs modes de **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-)composants éventuels ont des spécifications de **protection** égales ou plus restrictives et, si ce sont des modes repère, 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 repère lié, le mode **repéré** de M doit être **l-équivalent** avec le mode **repéré** de N;
- si M et N sont des modes descripteur, le mode **repéré originel** de M doit être **l-équivalent** avec le mode **repéré originel** de N;

- si M et N sont des modes rangée, le mode des **éléments** de M doit être **compatible en lecture** avec le mode des **éléments** de N;
- si M et N sont des modes structure qui ne sont pas des modes structure **paramétrés**, tout mode de **champ** de M doit être **compatible en lecture** avec le mode de **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, rangée 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 rangée 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($p_1, \ldots, p_n$) est 1. **équivalent**, 2. **compatible en lecture** avec N, où $p_1, \ldots, p_n$ 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 repérer**. On dit qu'un mode M est **limitable à** un mode N si lui ou ses (sous)-composants éventuels repèrent des locus qui ont des spécifications de **protection** égales ou plus restrictives que ceux repérés 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 repère lié, le mode **repéré** de M doit être **compatible en lecture** avec le mode **repéré** de N;
- si M et N sont des modes descripteur, le mode **repéré originel** de M doit être **compatible en lecture** avec le mode **repéré originel** de N;
- si M et N sont des modes rangée, le mode des **éléments** de M doit être **limitable au** mode des **éléments** de N;
- si M et N sont des modes structure, chaque mode de **champ** de M doit être **limitable au** mode de **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 repère, un mode procédure ou un mode exemplaire;
- un mode M est **compatible** avec la N-classe par repère si et seulement si c'est un mode repère et l'une des conditions suivantes est satisfaite:
  1) N est un mode statique et M est un mode repère lié dont le mode **repéré** est **compatible en lecture** avec N;
  2) N est un mode statique et M est un mode repère libre;
  3) M est un mode rangée dont le mode **repéré 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 repérer**, M et N doivent être **v-équivalents**;
  2) si M a la **propriété de repérer**, 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 repère.
- 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 repère, un mode procédure ou un mode exemplaire.
- La M-classe par repère est **compatible** avec la N-classe par repère 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.-à-d. aucune exception ne peut être causée.
- La M-classe par repère est **compatible** avec la N-classe par valeur si et seulement si N est un mode repère et l'une des conditions suivantes est satisfaite:
  1) M est un mode statique et N est un mode repère lié dont le mode **repéré** est **équivalent** à M.
  2) M est un mode statique et N est un mode repère libre.
  3) N est un mode descripteur dont le mode **repéré 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.**

12.2 VISIBILITÉ ET IDENTIFICATION

La définition de la visibilité et de l'identification repose sur la terminologie suivante:

- *représentation textuelle de nom*: désigne un symbole terminal auquel est attachée une représentation textuelle de nom **canonique** (voir la section 2.7) et des propriétés de visibilité;
- nom: désigne une *représentation textuelle de nom simple* associée à la *définition* qui l'a créée (voir la section 10.1);
- *nom*: désigne une occurrence d'utilisation d'un nom (avec éventuellement une représentation textuelle de nom préfixe).

12.2.1 Degrés de visibilité

Les règles d'identification sont fondées sur la visibilité des *représentations textuelles de nom* dans les domaines d'un programme. Dans un domaine, chaque *représentation textuelle de nom* possède l'un des quatre degrés de visibilité suivants:

TABLEAU 1 – Degrés de visibilité

Visibilité	**Propriétés** (informel)
Directement fortement visible	La *représentation textuelle de nom* est **visible** par création, octroi, saisie ou héritage de la spec au corps
Indirectement fortement visible	La *représentation textuelle de nom* est prédéfinie ou héritée via une imbrication de bloc
Faiblement visible	La *représentation textuelle de nom* est **impliquée** par une *représentation textuelle* de nom **fortement visible**
Invisible	La *représentation textuelle de nom* ne peut pas être utilisée

Une *représentation textuelle de nom* est dite être **fortement visible** dans un domaine si elle est soit **directement fortement visible** soit **indirectement fortement visible** dans ce domaine. Une *représentation textuelle de nom* est dite être **visible** si elle est soit **faiblement**, soit **fortement 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 *représentation textuelle de nom* appartient.

Lorsqu'une *représentation textuelle de nom* est **visible** dans un domaine, elle peut être **directement liée** à une autre *représentation textuelle de nom* dans un autre domaine, ou **directement liée** à une *définition* dans le programme. Les règles de **liaison directe** sont énoncées dans la section 12.2.3. On notera que toute utilisation d'une règle introduit une nouvelle **liaison directe** pour une *représentation textuelle de nom.*

Sur la base de la **liaison directe**, la notion de **liaison** (non nécessairement **directe**) se définit comme suit:

Une *représentation textuelle de nom* N1, **visible** dans le domaine R1, est dite être **liée** à une *représentation textuelle de nom* N2 dans le domaine R2 ou à une *définition* D, si et seulement si l'une des conditions suivantes se vérifie:

- N1 dans R1 est **directement lié** à N2 en R2 ou à D. Toutefois, si N1 est **directement lié** à plusieurs *définitions* dans R1, alors toutes ces *définitions* sauf une sont superflues et N1 est **lié** arbitrairement à une de ces définitions dans R1.
- N1 dans R1 est **directement lié** à un N dans un R et, dans R, N est **lié** à N2 en R2, ou à D.

12.2.2 Conditions de visibilité et identification

Dans chaque domaine d'un programme, les conditions suivantes doivent être safisfaites:

- si une *représentation textuelle de nom* est **fortement visible** dans un domaine et si elle a plus d'une **liaison directe**, alors:
  - elle ne doit être **directement liée** qu'aux *définitions*, et ces *définitions* doivent définir les mêmes éléments d'ensembles de modes d'ensembles qui sont **similaires**, ou
  - elle doit être **liée** à exactement une seule *définition* **réelle** et une **quasi**-*définition.*

Une *représentation textuelle de nom* **faiblement visible** dans un domaine et **liée** comme une *représentation textuelle de nom* **faiblement visible** dans ce domaine à des *définitions* qui ne définissent pas le même élément d'ensemble de modes ensemble **similaires** est dite avoir une **faible discordance** dans ce domaine.

Une *représentation textuelle de nom* NS, **visible** dans le domaine R, est dite **liée** dans R à plusieurs *définitions* selon les règles suivantes:

- si NS est **fortement visible** dans R, NS est **liée** aux *définitions* auxquelles elle est **liée** dans R (comme une *représentation textuelle de nom* **fortement visible**). Si elle est **liée** à la fois à une **quasi**-*définition* et à une *définition* **réelle**, la **quasi**-*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, héritée et n'introduit pas de noms **impliqués**);
- autrement, si NS est **faiblement visible** dans R, elle est **liée** aux *définitions* auxquelles elle est **liée** dans R (comme une *représentation textuelle de nom* **faiblement visible**), à condition que NS n'ait pas de **faible discordance** dans R (les **discordances faibles** sont autorisées dans un domaine s'il n'existe dans celui-ci aucun *nom* avec une *représentation textuelle de nom* donnant lieu à une **faible discordance**);
- sinon, NS n'est pas **liée** dans R.

conditions statiques:

La *représentation textuelle de nom* attachée à chaque *nom* immédiatement englobé dans un domaine doit être **liée** dans ce domaine.

identification:

Un *nom* N ayant une *représentation textuelle de nom* NS dans un domaine R est **lié** aux *définitions* auxquelles NS est **lié** dans R.

12.2.3 Visibilité dans les domaines

12.2.3.1 Généralités

Une *représentation textuelle de nom* est **directement fortement visible** dans un domaine, selon les règles suivantes:

- cette *représentation textuelle de nom* est saisie dans le domaine (voir la section 12.2.3.5);
- cette *représentation textuelle de nom* est octroyée dans le domaine (voir la section 12.2.3.4);
- il existe une *définition* ayant cette *représentation textuelle de nom* dans le domaine. En pareil cas, la *représentation textuelle de nom* dans le domaine est **directement liée** à la *définition*. (A noter que la *représentation textuelle de nom* peut être **directement liée** à plusieurs *définitions* dans le domaine.);
- le domaine est 1. un *corps de module*, 2. un *corps de région* et la *représentation textuelle de nom* est **directement fortement visible** dans le domaine 1. *d'un module de spec*, 2. *d'une région de spec* **correspondante**. La *représentation textuelle de nom* est **directement liée** à la *représentation textuelle de nom* dans le domaine correspondant.

Une *représentation textuelle de nom* qui n'est pas **directement fortement visible** dans un domaine y est **indirectement fortement visible**, selon les règles suivantes:

- le domaine est un bloc et la *représentation textuelle de nom* est **fortement visible** dans le domaine immédiatement englobant. La *représentation textuelle de nom* est dite être héritée par le bloc et elle est **directement liée** à la même *représentation textuelle de nom* dans le domaine immédiatement englobant;
- le domaine n'est pas un bloc dans lequel la *représentation textuelle de nom* est héritée et la *représentation textuelle de nom* est une *représentation textuelle de nom* définie par le langage (voir l'Appendice C.2) ou par l'implémentation. La *représentation textuelle de nom* est considérée être **directement liée** à une *définition* dans le domaine de la définition de processus imaginaire la plus externe pour sa signification prédéfinie.

Une *représentation textuelle de nom* qui n'est pas **fortement visible** dans un domaine y est **faiblement visible** si elle est **impliquée** par une *représentation textuelle de nom* qui est **fortement visible** dans le domaine. La *représentation textuelle de nom* dans un domaine est **directement liée** à une *définition* **impliquée** (voir la section 12.2.4).

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 contrôlent la visibilité *des représentations textuelles de nom* qui y sont mentionnées et, implicitement, de leurs *représentations textuelles de nom* **impliquées.**

propriétés statiques :

Un *énoncé de visibilité* a un ou deux domaines **originels** (voir la section 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 immédiatement englobant 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*, alors, son domaine **originel** est le domaine immédiatement englobant l'*énoncé d'octroi*, et ses domaines de **destination** sont les domaines immédiatement englobant 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 représentations textuelles 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 *représentations textuelles de nom* qui est l'ensemble de *représentations textuelles de nom* attaché à son *postfixe de saisie* ou à l'ensemble de *représentations textuelles de nom* attaché à son *postfixe d'octroi*. Ces *représentations textuelles de nom* sont les *représentations textuelles de nom* de postfixe de la *clause renommer préfixe*.

Une *clause renommer préfixe* a un ensemble d'**anciennes** *représentations textuelles de nom* et un ensemble de **nouvelles** *représentations textuelles de nom*. Chaque *représentation textuelle de nom* de postfixe attachée à une *clause renommer préfixe* donne à la fois une **ancienne** *représentation textuelle de nom* et une **nouvelle** *représentation textuelle de nom* attachées à cette *clause*, comme suit: on obtient la **nouvelle** *représentation textuelle de nom* en préfixant la *représentation textuelle de nom* de postfixe avec le *nouveau préfixe*; on obtient l'**ancienne** *représentation textuelle de nom* en préfixant la représentation textuelle de nom de postfixe avec l'*ancien préfixe*.

Quand une **nouvelle** *représentation textuelle de nom* et une **ancienne** *représentation textuelle de nom* sont obtenues à partir de la même *représentation textuelle de nom* de postfixe, l'**ancienne** *représentation textuelle de nom* est dite être la source de la **nouvelle** *représentation textuelle de nom*.

règles de visibilité:

Les **nouvelles** *représentations textuelles 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 *représentations textuelles de nom* sont saisies (octroyées) dans leurs domaines de **destination**.

Une *représentation textuelle de nom* NS est dite être **saisissable** par le modulion M immédiatement englobé dans le domaine R si et seulement si elle est **fortement visible** dans R et si elle n'est ni **liée** dans R à une *représentation textuelle de nom* quelconque dans le domaine de M ni **directement liée** à la *définition* d'une *représentation textuelle de nom* prédéfinie.

Une *représentation textuelle de nom* NS est dite être **octroyable** par le modulion M immédiatement englobé dans le domaine R si et seulement si elle est **fortement visible** dans le domaine de M et si elle n'est ni **liée** dans M à une *représentation textuelle de nom* quelconque dans R ni **directement liée** dans M à la *définition* d'une *représentation textuelle 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** *représentations textuelles de nom* doit être:

- **liée** dans le domaine immédiatement englobant le domaine de M et
- **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** *représentations textuelles de nom* doit être:

- **liée** 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)*.

exemples:

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>*]; (1.2)

<fenêtre d'octroi> ::= (2)
<postfixe d'octroi> {, *<postfixe d'octroi>* }* (2.1)

<postfixe d'octroi> ::= (3)
<représentation textuelle de nom> (3.1)
| *<représentation textuelle de nom* de neumode*>* *<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)

sémantique:

Les énoncés d'octroi sont un moyen d'étendre aux domaines immédiatement englobants la visibilité des représentations textuelles de nom d'un domaine de modulion. **FORBID** ne peut être spécifié que pour des noms **de neumode** 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 (des) *clause(s) renommer préfixe*, l'*énoncé d'octroi* a l'effet de sa (ses) *clause(s) renommer préfixe* (voir la section 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:
  - A 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 **neumode** (voir la section 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**

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 *représentation textuelle de nom* est déterminée comme suit:
  - Si le domaine immédiatement englobant le *préfixe* est un *module* ou une *région*, alors, la *représentation textuelle de nom* est la même que celle du nom de modulion de ce modulion.
  - Si le domaine immédiatement englobant le *préfixe* est une *région de spec* ou un *module de spec*, alors, la représentation textuelle de nom est la représentation textuelle de nom qui précède **SPEC**.

Un *postfixe d'octroi* a un ensemble de *représentations textuelles de nom* qui lui est attaché, défini comme suit:

- Si c'est une *représentation textuelle de nom*, ou s'il contient une *représentation textuelle de nom* de neumode, alors, l'ensemble ne contient que cette *représentation textuelle 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 *représentations textuelles de nom* de la forme *OP ! N* (c.-à-d. obtenues en préfixant *N* par *OP*) pour toute *représentation textuelle de nom N* telle que *OP ! N* est **fortement visible** dans le domaine du modulion dans lequel le *postfixe d'octroi* est placé et **octroyable** par ce modulion.

conditions statiques:

La *représentation textuelle de nom* de **neumode** avec *clause d'interdiction* doit être **fortement visible** dans le domaine *R* du modulion dans lequel se trouve l'*énoncé d'octroi.* La *représentation textuelle de nom* de **neumode** doit être **liée** dans *R* à la *définition* d'un neumode, qui doit être un mode structure, et chaque *nom de champ* dans la *liste de noms de champ* doit être un *nom de champ* de ce mode. La *définition* de neumode doit être immédiatement englobée par *R.* Tous les noms de **champ** dans une *liste de noms d'interdiction* doivent avoir des représentations textuelles 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 *représentation textuelle de nom* qui est **liée** dans le domaine de la *définition d':*

- un nom **de locus**, ou
- un nom de **loc-identité**, 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* et,

- si son modulion immédiatement englobant est un *module* ou une *région*, alors, il doit être nommé (c.-à-d. qu'il doit être précédé par une *définition* suivie par 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 *représentation textuelle de nom simple.*

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)
<représentation textuelle de nom>	(3.1)
| [*<préfixe>* !] **ALL**	(3.2)

sémantique:

Les énoncés de saisie sont un moyen d'étendre la visibilité des représentations textuelles 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 (des) *clause(s) renommer préfixe*, il a l'effet de sa (ses) *clause(s) renommer préfixe* (voir la section 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 *représentations textuelles de nom*, défini comme suit:

- Si le *postfixe de saisie* est une *représentation textuelle de nom*, l'ensemble ne contient que la *représentation textuelle 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 *représentations textuelles de nom* de la forme *OP ! S*, pour toutes les *représentations textuelles de nom S*, telles que *OP ! S* est **fortement visible** dans le domaine immédiatement englobant le modulion dans lequel se trouve l'*énoncé de saisie* et **peut être saisi** par ce modulion.

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* et

- si son modulion immédiatement englobant est un *module* ou une *région*, alors, il doit être nommé (c.-à-d. qu'il doit être précédé par une *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 *représentation textuelle de nom simple*.

exemples:

25.35 **SEIZE** *(stack ! int -> stack) !* **ALL**; (1.1)

12.2.4 Représentations textuelles de nom impliquées

Chaque *représentation textuelle de nom* **fortement visible** dans un domaine R a un ensemble de *représentations textuelles de nom* **impliquées**, qui peuvent être **faiblement visibles** dans R.

Chaque *mode* a un ensemble éventuellement vide de *définitions* **impliquées** attachées à un domaine, comme indiqué dans le Tableau 2.

Chaque *représentation textuelle de nom NS*, **fortement visible** dans le domaine R, a un ensemble de *définitions* **impliquées**, définies comme suit, où D est l'une des *définitions* auxquelles NS est **lié** dans R:

- Si D définit un nom **d'accès** de mode M, les *définitions* **impliquées** de NS dans R sont celles qui sont **impliquées** dans R par M.
- Si D définit un nom de **mode**, les *définitions* **impliquées** de NS dans R sont les définitions **impliquées** dans R par le mode définissant du *nom* de **mode**.

- Si D définit un nom de **procédure**, les *définitions* **impliquées** de NS dans R sont les définitions **impliquées** dans R par les modes des **specs de paramètre** et des **specs de résultat** de la procédure, si elles existent.
- Si D définit un nom de **processus**, les *définitions* **impliquées** de NS dans R sont les définitions **impliquées** dans R par les modes des **specs de paramètre**, si elles existent.
- Si D définit un nom de **signal**, les *définitions* **impliquées** de NS dans R sont toutes les *définitions* **impliquées** dans R par tous les modes attachés au signal.
- Sinon, l'ensemble est vide.

TABLEAU 2 — Définitions impliquées de modes dans le domaine R

Modes	**Ensembles de définitions impliquées**
INT, BOOL, CHAR, **RANGE** *(...),* **BIN** *(n), PTR, INSTANCE,* **EVENT**, *ASSOCIATION, TIME, DURATION,* **BOOLS** (n), **CHARS** (n)	Vide
nom de mode	L'ensemble de *définitions* **impliquées** dans R par son mode **définissant**
nom de mode (...) (paramétré)	L'ensemble de *définitions* **impliquées** dans R par *nom de mode*
M(m:n), **REF** *M,* **ROW** *M* **READ** *M,* **POWERSET** *M* **BUFFER** *M* **TEXT** (...) *M*	L'ensemble de *définitions* **impliquées** dans R par *M*
SET *(...)*	L'ensemble de *définitions* **d'éléments d'ensemble** dans le mode
PROC $(M_1, \ldots, M_n)$ (M_{n+1})	L'union des ensembles de *définitions* **impliquées** dans R par M_1 jusqu'à M_{n+1}
ARRAY *(M) N* **ACCESS** *(M) N*	L'union des ensembles de *définitions* **impliquées** dans R par *M* et *N*
STRUCT $(N_1\ M_1, \ldots, N_n\ M_n)$	L'union des ensembles *de définitions* **impliquées** dans R par M_i pour des champs qui sont **visibles** dans R. Pour les structures variables, c'est l'union des *définitions* **impliquées** dans R par les champs de la structure variable qui sont **visibles** dans R

Si une *représentation textuelle de nom* NS, **fortement visible** dans un domaine R, a des *définitions* **impliquées**, chacune de ces *définitions* spécifie une *représentation textuelle de nom* **impliquée** pour NS dans R: soit D une *définition* **impliquée** par NS dans R et soit Ni la *représentation textuelle de nom* de D. Deux cas peuvent se présenter:

- NS est une *représentation textuelle de nom simple.* Alors, Ni est une *représentation textuelle de nom* **impliquée** de NS.
- NS a la forme P ! S, où S est une *représentation textuelle de nom simple.* Alors, P ! Ni est une *représentation textuelle de nom* **impliquée** de NS.

exemples:

```
m: MODULE
    DCL x SET (on, off);
    GRANT x PREFIXED;
END;
/* m ! x visible here with implied m ! on, m ! off */
```

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.*

Dans ces cas, la *représentation textuelle de nom* du *nom* de *champ* peut être liée à une *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 la *définition* à laquelle la *représentation textuelle de nom de neumode* est **liée** dans le domaine dans lequel se trouve la *clause d'interdiction.*

Cependant, si la **nouveauté** de M est une *définition* qui définit un nom de **neumode** 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.

12.3 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 choix de champs (voir la section 3.12.4), auquel cas une liste de champs récurrents est sélectionnée,
- des multiplets de rangée avec indices (voir la section 5.2.5), auquel cas une valeur élément de rangée est sélectionnée,
- des expressions conditionnelles (voir la section 5.3.2), auquel cas une expression est sélectionnée,
- des actions de cas (voir la section 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 de rangée avec indices, chaque valeur est étiquetée avec une liste d'étiquettes de cas. Pour faciliter l'explication, la liste d'étiquettes de cas pour le multiplet de rangée avec indices sera considérée dans cette section 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.-à-d. appartenant à la même valeur de sélecteur) dans toute spécification d'étiquettes de cas.

propriétés statiques:

- A un *choix de champs* avec *spécification d'étiquettes de cas*, un *multiplet de rangée avec indices*, une *expression conditionnelle*, ou une *action de cas* on attache une liste de spécifications d'étiquettes de cas, formée en prenant, respectivement, la *spécification d'étiquettes de cas* précédant chaque *champ à choisir*, *valeur*, ou *cas à choisir*.
- A 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**.
- A 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*.
- A une *spécification d'étiquettes de cas* on attache une liste des classes qui sont les classes de chaque liste d'étiquettes de cas.
- A 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 marqueurs**, c'est l'ensemble des valeurs défini par le mode du champ **marqueur** correspondant.
- Pour un mode structure **variable sans marqueurs**, c'est l'ensemble des valeurs défini par le mode **racine** de la **classe résultante** correspondante (qui n'est jamais la classe **toute**, voir la section 3.12.4).
- Pour un multiplet de rangée, c'est l'ensemble des valeurs défini par le mode **d'indice** du mode du multiplet de rangée.
- 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 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.-à-d. que chaque liste de valeurs des sélecteurs possible ne correspond qu'à une spécification d'étiquettes 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

Cette section 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 la section appropriée, la définition est donnée ici, sinon la section appropriée est référencée.

12.4.1 Noms

Noms de mode

nom de mode:	voir la section 3.2.1.
nom de mode temps absolu:	un *nom* défini par un mode temps absolu.
nom de mode accès:	un *nom* défini par un mode accès.
nom de mode association:	un *nom* défini par un mode association.
nom de mode booléen:	un *nom* défini par un mode booléen.
nom de mode caractère:	un *nom* défini par un mode caractère.
nom de mode chaîne:	un *nom* défini par un mode chaîne.
nom de mode chaîne paramétré:	un *nom* défini par un mode chaîne **paramétré**.
nom de mode descripteur:	un *nom* défini par un mode descripteur.
nom de mode discret:	un *nom* défini par un mode discret.
nom de mode durée:	un *nom* défini par un mode durée.
nom de mode ensemble:	un *nom* défini par un mode ensemble.
nom de mode ensembliste:	un *nom* défini par un mode ensembliste.
nom de mode entier:	un *nom* défini par un mode entier.
nom de mode événement:	un *nom* défini par un mode événement.
nom de mode exemplaire:	un *nom* défini par un mode exemplaire.
nom de mode intervalle:	un *nom* défini par un mode intervalle.
nom de mode procédure:	un *nom* défini par un mode procédure.
nom de mode rangée:	un *nom* défini par un mode rangée.
nom de mode rangée paramétré:	un *nom* défini par un mode rangée **paramétré**.
nom de mode repère libre:	un *nom* défini par un mode repère libre.
nom de mode repère lié:	un *nom* défini par un mode repère lié.
nom de mode structure:	un *nom* défini par un mode structure.
nom de mode structure paramétré:	un *nom* défini par un mode structure **paramétré**.
nom de mode structure variable:	un *nom* défini par un mode structure.
nom de mode tampon:	un *nom* défini par un mode tampon.
nom de neumode:	voir la section 3.2.3
nom de synmode:	voir la section 3.2.2

Noms d'accès

nom de loc-identité:	voir la section 4.1.3
nom de locus:	voir la section 4.1.2
nom de locus faire-avec:	voir la section 6.5.4
nom d'énumération de locus:	voir la section 6.5.2

Noms de valeur

nom de littéral de booléen:	voir la section 5.2.4.3
nom de littéral de vide:	voir la section 5.2.4.6
nom d'énumération de valeur:	voir la section 6.5.2
nom de synonyme:	voir la section 5.1
nom de valeur faire-avec:	voir la section 6.5.4
nom de valeur reçue:	voir les sections 6.19.2, 6.19.3

Noms divers

nom de champ marqueur:	voir la section 3.12.4
nom d'élément d'ensemble:	voir la section 3.4.5
nom de locus repère lié:	un *nom de locus* dont le mode est un mode repère lié.
nom de locus repère libre:	un *nom de locus* dont le mode est un mode repère libre.
nom de procédure:	voir la section 10.4
nom de procédure générale:	un *nom de procédure* dont la généralité est un nom de procédure **générale.**
nom de processus:	voir la section 10.5
nom de signal:	voir la section 11.5
nom de synonyme indéfini:	voir la section 5.1
nom d'étiquette:	voir les sections 6.1 et 10.6
nom d'opération prédéfinie:	un nom défini par CHILL ou par l'implémentation dénotant une opération prédéfinie
nom non réservé:	un *nom* qui n'est aucun des noms réservés mentionnés à l'Appendice C1.
représentation textuelle de nom de neumode:	une *représentation textuelle de nom* **liée** à la *définition* d'un nom de **neumode**

12.4.2 Locus

locus accès:	un *locus* qui a un mode accès.
locus association:	un *locus* qui a un mode association.
locus chaîne:	un *locus* qui a un mode chaîne.
locus chaîne de caractères:	un *locus* qui a un mode chaîne de **caractères.**
locus discret:	un *locus* qui a un mode discret.
locus événement:	un *locus* qui a un mode événement.
locus exemplaire:	un *locus* qui a un mode exemplaire.
locus de mode statique:	un *locus* qui a un mode statique.
locus rangée:	un *locus* qui a un mode rangée.
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.

12.4.3 Expressions et valeurs

expression booléenne:	une *expression* dont la classe est **compatible** avec un mode booléen.
valeur primitive temps absolu:	une *valeur primitive* dont la classe est **compatible** avec un mode temps absolu.
expression chaîne de caractères:	une *expression* dont la classe est **compatible** avec un mode chaîne de **caractères.**
expression rangée:	une *expression* dont la classe est **compatible** avec un mode rangée.
valeur primitive rangée:	une *valeur primitive* dont la classe est **compatible** avec un mode rangée.
valeur primitive repère lié:	une *valeur primitive* dont la classe est **compatible** avec un mode repère lié.

valeur primitive chaîne: une *valeur primitive* dont la classe est **compatible** avec un mode chaîne.

valeur constante: une *valeur* qui est **constante.**

valeur primitive descripteur: une *valeur primitive* dont la classe est **compatible** avec un mode descripteur.

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.

valeur primitive durée: une *valeur primitive* dont la classe est **compatible** avec un mode durée.

valeur primitive exemplaire: une *valeur primitive* dont la classe est **compatible** avec un mode exemplaire.

expression chaîne: une *expression* dont la classe est **compatible** avec un mode chaîne.

expression littérale discrète: une *expression discrète* qui est **littérale.**

expression littérale entière: une *expression entière* qui est **littérale.**

valeur primitive procédure: une *valeur primitive* dont la classe est **compatible** avec un mode procédure.

valeur primitive repère: une *valeur primitive* dont la classe est **compatible** avec un mode repère lié, un mode repère libre ou un mode descripteur.

valeur primitive repère libre: une *valeur primitive* dont la classe est **compatible** avec un mode repère libre.

valeur primitive structure: une *valeur primitive* dont la classe est **compatible** avec un mode structure.

12.4.4 Catégories sémantiques diverses

mode chaîne: un *mode* dans lequel le *mode composé* est un *mode chaîne.*

mode discret: un *mode* dans lequel le *mode non composé* est un *mode discret.*

appel de procédure rendant locus: voir la section 6.7.

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 ouverture de parenthèse (( ).

mode rangée: un *mode* dans lequel le *mode composé* est un *mode rangée.*

appel de procédure rendant valeur: voir la section 6.7.

appel d'opération prédéfinie rendant locus: voir la section 6.7.

appel d'opération prédéfinie rendant valeur: voir la section 6.7.

13 OPTIONS POUR L'IMPLÉMENTATION

13.1 OPÉRATIONS PRÉDÉFINIES PAR L'IMPLÉMENTATION

sémantique :

Une implémentation peut fournir un ensemble d'opérations prédéfinies par l'implémentation en plus de l'ensemble des opérations prédéfinies par le langage.

Le mécanisme de passage de paramètres est défini par l'implémentation.

noms prédéfinis :

Le nom d'une opération prédéfinie par l'implémentation est prédéfini comme un nom **d'opération prédéfinie.**

propriétés statiques :

A nom d'**opération prédéfinie** peut être attaché un ensemble de noms d'exception définis par l'implémentation. Un *appel d'opération prédéfinie* est un *appel d'opération 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 d'opération 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,* c.-à-d. 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 **neumode, 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 NOMS DE PROCESSUS DÉFINIS PAR L'IMPLÉMENTATION

Une implémentation peut définir un ensemble de noms de **processus** définis par l'implémentation, c.-à-d. 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 exemplaire les dénotant peuvent être manipulées.

13.4 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.

13.5 NOMS D'EXCEPTION DÉFINIS PAR L'IMPLÉMENTATION

Une implémentation peut définir un ensemble de noms d'exception.

13.6 AUTRES CARACTÉRISTIQUES DÉFINIES PAR L'IMPLÉMENTATION

- vérification statique des conditions dynamiques (voir la section 2.1.2)
- *directive d'implémentation* (voir la section 2.6)
- *nom de repère de texte* (voir les sections 2.7 et 10.10.1)
- **récursivité** et **généralité** par défaut (voir les sections 3.7 et 10.4)
- ensemble de valeurs de modes durée (voir la section 3.11.2)
- ensemble de valeurs de modes temps absolu (voir la section 3.11.3)
- **implantation d'élément** par défaut (voir la section 3.12.3)

- comparaison de valeurs de structures **variables sans marqueurs** (voir la section 3.12.4)
- nombre de bits dans un mot (voir la section 3.12.5)
- occupation minimale de bits (voir la section 3.12.5)
- autres (sous-)locus **repérables** (voir la section 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 marqueurs** (voir les sections 4.2.2 et 5.2.3)
- sémantique de champs **récurrents** à structure **variable sans marqueurs** (voir les sections 4.2.10, 5.2.13 et 6.2)
- sémantique de la *conversion de locus* (voir la section 4.2.13)
- sémantique de la *conversion d'expression* et autres conditions (voir la section 5.2.11)
- autres *paramètres réels* dans une *expression démarrer* (voir la section 5.2.14)
- intervalles de valeurs pour des expressions **littérales** et **constantes** (voir la section 5.3.1)
- algorithme de séquencement (voir les sections 6.15, 6.18.2, 6.18.3, 6.19.2 et 6.19.3)
- libération de la mémoire dans *TERMINATE* (voir la section 6.20.4)
- dénotation des fichiers (voir la section 7.1)
- opérations sur les associations (voir les sections 7.1 et 7.2.1)
- associations non exclusives (voir la section 7.1)
- autres attributs des valeurs d'association (voir la section 7.2.2)
- sémantique de *paramètres pour associer* (voir la section 7.4.2)
- exception *ASSOCIATEFAIL* (voir la section 7.4.2)
- sémantique des *paramètres pour modifier* (voir la section 7.4.5)
- exceptions *CREATEFAIL*, *DELETEFAIL* et *MODIFYFAIL* (voir la section 7.4.5)
- exception *CONNECTFAIL* (voir la section 7.4.6)
- sémantique de la lecture d'enregistrements qui ne sont pas des valeurs autorisées par le mode enregistrement (voir la section 7.4.9)
- autres actions **temporisables** (voir la section 9.2)
- exception *TIMERFAIL* (voir les sections 9.3.1, 9.3.2 et 9.3.3)
- précision des valeurs de durée (voir les sections 9.4.1 et 9.4.2)
- indication des valeurs **constantes** dans des *quasi-définitions de synonyme* (voir la section 10.10.3)
- **régionalité** des opérations prédéfinies (voir la section 11.2.2).

APPENDICE A

Ensemble de caractères pour le langage CHILL

L'ensemble de caractères du langage CHILL est une extension de l'alphabet n°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 moindre 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 B

Symboles spéciaux et combinaisons de caractères

	Nom	Usage
;	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	repérage ou dérepérage, 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
“	citation	symbole de début ou de fin dans les littéraux de chaîne de caractères
“ “	double citation	citation dans les littéraux de chaîne de caractères
!	opérateur préfixant	préfixage de noms
B’	qualification littérale	base binaire pour littéral
D’	qualification littérale	base décimale pour littéral
H’	qualification littérale	base hexadécimale pour littéral
O’	qualification littérale	base octale pour littéral
−−	fin de ligne	délimiteur de fin de ligne des commentaires in-situ

APPENDICE C

Représentations textuelles de nom simple spéciales

C.1 Représentations textuelles de nom simple réservées

ACCESS
AFTER
ALL
AND
ANDIF
ARRAY
ASSERT
AT

BEGIN
BIN
BODY
BOOLS
BUFFER
BY

CASE
CAUSE
CHARS
CONTEXT
CONTINUE
CYCLE

DCL
DELAY
DO
DOWN
DYNAMIC

ELSE
ELSIF
END
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MOD
MODULE

NEWMODE
NONREF
NOPACK
NOT

OD
OF
ON
OR
ORIF
OUT

PACK
POS
POWERSET
PREFIXED
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
REM
REMOTE
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
SPEC
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

TEXT
THEN
THIS
TIMEOUT
TO

UP

VARYING

WHILE
WITH

XOR

C.2 Représentations textuelles de nom simple prédéfinies

ABS
ABSTIME
ALLOCATE
ASSOCIATE
ASSOCIATION

BOOL

CARD
CHAR
CONNECT
CREATE

DAYS
DELETE
DISCONNECT
DISSOCIATE
DURATION

EOLN
EXISTING
EXPIRED

FALSE
FIRST

GETASSOCIATION
GETSTACK
GETTEXTACCESS
GETTEXTINDEX
GETTEXTRECORD
GETUSAGE

HOURS

INDEXABLE
INSTANCE
INT
INTTIME
ISASSOCIATED

LAST
LENGTH
LOWER

MAX
MILLISECS
MIN
MINUTES
MODIFY

NULL
NUM

OUTOFFILE

PRED
PTR

READABLE
READONLY
READRECORD
READTEXT
READWRITE

SAME
SECS
SEQUENCIBLE
SETTEXTACCESS
SETTEXTINDEX
SETTEXTRECORD
SIZE
SUCC

TERMINATE
TIME
TRUE

UPPER
USAGE

VARIABLE

WAIT
WHERE
WRITEABLE
WRITEONLY
WRITERECORD
WRITETEXT

C.3 Noms d'exception

ALLOCATEFAIL
ASSERTFAIL
ASSOCIATEFAIL

CONNECTFAIL
CREATEFAIL

DELAYFAIL

DELETEFAIL

EMPTY

MODIFYFAIL

NOTCONNECTED
NOTASSOCIATED

OVERFLOW

RANGEFAIL
READFAIL

SENDFAIL
SPACEFAIL

TAGFAIL
TEXTFAIL
TIMERFAIL

WRITEFAIL

APPENDICE D

Exemples de programmes

1 Opérations sur des 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;
```

2 Mêmes opérations sur des 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;
```

3 Mêmes opérations sur des nombres complexes

```
 1  complex_operations:
 2   MODULE
 3       NEWMODE complex= STRUCT (re,im INT);
 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 ],
19            neutral_for_mult=complex [ 1,0 ];
20       GRANT neutral_for_add,
21               neutral_for_mult;
22
23   END complex_operations;
```

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) RECURSIVE;
 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;
```

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       DCL a STRUCT (a2,a1 BOOL),
28             b STRUCT (b2,b1 BOOL);
29       SYNMODE res= ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
30       DCL r INT, results res;
31       DO WITH a,b;
32          r := 0;
33          DO FOR a2 IN BOOL;
34              DO FOR a1 IN BOOL;
35                  DO FOR b2 IN BOOL;
36                      DO FOR b1 IN BOOL;
37                          r+ := 1;
38                          results (r) := adder (a,b);
39                      OD;
40                  OD;
41              OD;
42          OD;
43       OD;
44       ASSERT
45          results=res [[FALSE, FALSE, FALSE], [FALSE, FALSE, TRUE],
46                       [FALSE, TRUE, FALSE], [FALSE, TRUE, TRUE],
47                       [FALSE, FALSE, TRUE], [FALSE, TRUE, FALSE],
48                       [FALSE, TRUE, TRUE], [TRUE, FALSE, FALSE],
49                       [FALSE, TRUE, FALSE], [FALSE, TRUE, TRUE],
50                       [TRUE, FALSE, FALSE], [TRUE, FALSE, TRUE],
51                       [FALSE, TRUE, TRUE], [TRUE, FALSE, FALSE],
52                       [TRUE, FALSE, TRUE], [TRUE, TRUE, FALSE]];
53   END exhaustive_checker;
```

```
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<100 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+c)/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;
```

```
 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;
```

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;
```

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;
```

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                 RETURNstack (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                     THEN first := 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;
```

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      initialise:
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 : [white,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 initialise;
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;
```

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;
```

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;
```

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;
```

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;
```

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_buffers user_buffers,
48            allocator_buf REF allocator_buffers,
49            counter_buf REF counters_buffers;
50      START allocator (-> user_buffer);
51     allocator_buf:=(RECEIVE user_buffer).allocator;
52      END user_world;
53     counter_manager:
54      MODULE
55      SEIZE user_buffers, user_messages,
56              allocator_messages, allocator_buffers,
57              counter_messages, counters_buffers,
58      GRANT allocator;
59
60     allocator:
61      PROCESS (starter REF user_buffers);
62          DCL allocator_buffer allocator_buffers;
63          NEWMODE no_of_counters = INT (1:10);
64          DCL counters ARRAY (no_of_counters)
65                              STRUCT(counter REF counters_buffers,
66                                      status SET (busy, idle)),
67              message allocator_messages;
68          SEND starter → ([allocator_id, → allocator_buffer]);
69          DO FOR each IN counters;
70             START counter(->allocator_buffer);
71             each := [(RECEIVE allocator_buffer).counter, idle];
72          OD;
73          DO FOR EVER;
74              BEGIN
75              DCL user REF user_buffers;
76              message := RECEIVE allocator_buffer;
77              handle_messages:
78              CASE message, type OF
79              (acquire):
80                user := message, user;
81                 DO FOR each IN counters;
82                     DO WITH each;
83                         IF status = idle
84                             THEN status := busy;
85                                 SEND counter->([initiate, user]);
86                                 EXIT handle_messages;
87                         FI;
88                     OD;
89                 OD;
90                 SEND user->([congested]);
91             (release):
92                 SEND message,counter->([terminate]);
```

```
93              find_counter:
94              DO FOR each IN counters;
95                  DO WITH each;
96                      IF message.counter = counter
97                          THEN status := idle;
98                                  EXIT find_counter;
99                      FI;
100                 OD;
101             OD find_counter;
102         (counter_id);
103         ESAC handle_messages;
104         END;
105     OD;
106  END allocator;
107 counter:
108  PROCESS (starter REF allocator_buffers);
109      DCL counter_buffer counters_buffers;
110      SEND starter->([counter_id,->counter_buffer]);
111      DO FOR EVER;
112         BEGIN
113             DCL user REF user_buffers,
114                     count INT := 0,
115                     message counter_messages;
116             message:= RECEIVE counter_buffer;
117             CASE message.type OF
118                     (initiate ): user := message.user;
119                                 SEND user->([ready,->counter_buffer]);
120                     ELSE/* some error action */
121             ESAC;
122         work_loop:
123         DO FOR EVER;
124             message:= RECEIVE counter_buffer;
125             CASE message.type OF
126                 (advance): count + := 1;
127                 (terminate): SEND user->([readout, count]);
128                             EXIT work_loop;
129                 ELSE /* some error action */
130             ESAC;
131         OD work_loop;
132      END;
133      OD;
134  END counter;
135  END counter_manager;
```

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;
```

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;
```

19 Enlever un élément d'une liste doublement chaînée circulairement

```
1   queue: MODULE
2       SYNMODE info=INT;
3       queue_removal:
4       MODULE
5           SEIZE info;
6           GRANT remove;
7           remove:
8           PROC(p PTR) RETURNS (info) EXCEPTIONS (EMPTY);
9                   /* This procedure removes the item referred to
10                  by p from a queue and returns the information
11                  contents of that queue element */
12                  SYNMODE element = STRUCT (
13                              i info POS (0,8:31),
14                              prev PTR POS (1,0:15),
15                              next PTR POS (1,16:31));
16                  DCL x REF element LOC := element (p), prev, next PTR;
17                  prev := x->.prev;
18                  next := x->.next;
19                  x->prev, x->.next := NULL;
20                  RESULT x->.i;
21                  p := prev;
22                  x->.next := next;
23                  p := next;
24                  x->.prev := prev;
25          END remove;
26      END queue_removal;
27  END queue;
```

20 Mettre à jour un fichier

```
1   read_modify_write:
2    MODULE
3
4     /* this example indicates how the CHILL i/o concepts can be used  */
5     /* to write an application where a record of a random accessible  */
6     /* file can be updated or added if not yet in use                 */
7
8     NEWMODE
9       index_set = INT (1:1000),
10      record_type = STRUCT (
11                          free BOOL,
12                          count INT,
13                          name CHARS (20));
```

```
14
15   DCL
16       curindex          index_set,
17       file_association  ASSOCIATION,
18       record_file       ACCESS (index_set) record_type,
19       record_buffer     record_type;
20
21   ASSOCIATE (file_association, "DSK:RECORDS.DAT");     /* create association    */
22   CONNECT (record_file,file_association,READWRITE);    /* connect to file       */
23   curindex := 123;                                     /* position record       */
24   READRECORD (record_file,curindex,record_buffer);     /* read the record       */
25       IF record_buffer.free                            /* if record is free     */
26          THEN                                          /* the claim and         */
27             record_buffer.free := FALSE                /* initialize it         */
28             record_buffer.count := 0;
29             record_buffer.name := "CHILL I/O concept";
30       FI;
31  record_buffer.count + := 1;                           /* increment its count   */
32  WRITERECORD (record_file, curindex, record_buffer);   /* write the record      */
33  DISSOCIATE (file_association);                        /* end the association   */
34
35   END read_modify_write;
```

21 Fusionner deux fichiers assortis

```
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 outpout                         */
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
67         DO
68            FOR curassoc IN inassocs;
69                DELETE (curassoc);                  /* delete the file              */
70                DISSOCIATE (curassoc);              /* and terminate association    */
71         OD;
72         DISSOCIATE (outassoc);                     /* disconnect and terminate     */
73
74     END merge_sorted_files;
```

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 = CHAR (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       ACCESS 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 pushdownlist->;                             /* 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;
```

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;
```

24 Exemple d'un contexte

```
 1  CONTEXT
 2     /* This 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;
```

25 L'emploi du préfixage 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";
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";
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;
```

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) DYNAMIC := "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%Ci/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;
```

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;
```

28 Un type de données abstrait

```
 1      /* This example implements the functionality of example 27   */
 2      /* demonstrating how an abstract data type can be            */
 3      /* 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             stackindex INT INIT := min-1;
11      push:
12        PROC (e element) EXCEPTIONS (overflow)
13            IF stackindex = max THEN
```

```
14                  CAUSE overflow;
15              FI;
16              stackindex +:= 1;
17              stack(stackindex) := e;
18              RETURN;
19          END push;
20         pop:
21          PROC () EXCEPTIONS (underflow)
22              IF stackindex = min THEN
23                  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 THEN
32                  CAUSE bounds;
33              FI;
34              RETURN stack(i);
35          END elem;
36
37          GRANT push,pop,elem;
38      END stacks;
```

29 Exemple d'un module de spec

```
1          /* This SPEC MODULE defines the interface of example 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;
```

APPENDICE E

Caractéristiques retirées

Les caractéristiques décrites ci-dessous ne font pas partie de la présente Recommandation Z.200, mais elles faisaient partie de cette Recommandation dans le Livre rouge (tome VI, fascicule VI.12, 1984). On en trouvera ci-après une brève description; leur définition complète se trouve dans les sections correspondantes de la Recommandation Z.200 de 1984, qui sont mentionnées ci-après. Ces caractéristiques peuvent être acceptées par une implémentation.

1 *Directive de libération* (voir la section 2.6)

Une directive de libération a libéré les représentations textuelles de nom simple **réservées** spécifiées dans la *liste de représentations textuelles de nom simple réservées*, de sorte qu'elles ont pu être redéfinies.

2 *Syntaxe de mode entier* (voir la section 3.4.2)

BIN était la syntaxe dérivée pour *INT*.

3 *Modes ensemble avec des trous* (voir la section 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.

4 *Syntaxe des modes procédure* (voir la section 3.7)

Une *spec de résultat* sans la représentation textuelle de nom simple **réservée** optionnelle **RETURNS** était la syntaxe dérivée pour la *spec de résultat* avec **RETURNS**.

5 *Syntaxe des modes rangée* (voir la section 3.11.3)

La représentation textuelle de nom simple **réservée ARRAY** était optionnelle.

6 *Notation étagée de structures* (voir la section 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 rangées 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.

7 *Noms de référence d'implantation* (voir la section 3.11.6)

Les noms de référence d'implantation pouvaient être utilisés pour spécifier l'implantation d'une manière définie par l'implémentation.

8 *Déclarations de locus avec base* (voir la section 4.1.4)

Une déclaration de locus avec base sans nom de *locus repère lié ou libre* était la syntaxe dérivée pour un énoncé de définition de synmode. Une déclaration de locus avec base avec un nom de *locus repère lié ou libre* définissait un ou plusieurs noms d'accès. Ces noms constituaient un autre moyen pour accéder à un locus en dérepérant la valeur repère contenue dans le locus repère spécifié. Cette opération de dérepérage était accomplie chaque fois que, et seulement quand un accès était obtenu via un nom déclaré avec base.

9 *Littéraux de chaîne de caractères* (voir la section 5.2.4.6)

Les littéraux de chaîne de caractères étaient délimités par des caractères apostrophe. Outre la représentation imprimable, la représentation hexadécimale pouvait être utilisée. Les littéraux de chaîne de caractères de longueur un servaient de littéraux de caractère.

10 *Notation Addr* (voir la section 5.3.8)

ADDR (<*locus*>) était la syntaxe dérivée pour − > <*locus*>.

11 *Syntaxe d'affectation* (voir la section 6.2)

Le symbole = était la syntaxe dérivée pour le symbole := .

12 *Syntaxe d'action de cas* (voir la section 6.4)

La *liste d'intervalles* d'une *action de cas* pouvait être spécifiée plus généralement par un mode discret et pas seulement par un nom de mode discret.

13 *Syntaxe action faire-pour* (voir la section 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.

14 *Compteurs de boucles explicites* (voir la section 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.

15 *Syntaxe d'action appeler* (voir la section 6.7)

La représentation textuelle de nom simple **réservée CALL** était facultative. Une *action appeler* avec **CALL** était dérivée d'une *action appeler* sans **CALL**.

16 *Exception* **RECURSEFAIL** (voir la section 6.7)

L'exception *RECURSEFAIL* était causée quand une procédure **non récursive** s'appelait elle-même récursivement.

17 *Syntaxe d'action démarrer* (voir la section 6.13)

L'*action démarrer* avec l'option **SET** était la syntaxe dérivée pour l'action d'affectation simple: <*locus exemplaire*> ::= <*expression démarrer*>.

18 *Noms explicites de valeur reçue* (voir la section 6.19)

Une action recevoir signal et choisir et une action recevoir tampon et choisir pouvaient introduire des noms de **valeur reçue**. Si un nom était visible dans le domaine où l'*action recevoir signal et choisir* 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.

19 *Blocs* (voir la section 8.1)

L'*action conditionnelle*, l'*action de cas*, l'*action faire* et l'*action mettre en attente et choisir* n'étaient pas définies comme des blocs.

20 *Enoncé d'entrée* (voir la section 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.

21 *Noms de registre* (voir la section 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.

22 *Noms faiblement visibles et énoncés de visibilité* (voir *la section 10.2.4.3)*

Une *représentation textuelle 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 *définition* non englobée dans le domaine de M. Une *représentation textuelle 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 *définition* englobée dans R.

23 *Saisie par nom de modulion* (voir la section 10.2.4.5)

Si une *clause renommer préfixe* d'un *énoncé de saisie* avait un *postfixe de saisie* contenant une *représentation textuelle de nom* de modulion et **ALL**, la *clause renommer préfixe* était équivalente à un ensemble *d'énoncés de saisie*, pour toute représentation textuelle 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.

24 *Représentations textuelles de nom simple prédéfinies (voir la section C.2)*

AND, NOT, OR, REM, MOD, THIS et *XOR* étaient des représentations textuelles de nom simple prédéfinies.

APPENDICE F

Ensemble de règles de production

2 PRÉLIMINAIRES

<représentation textuelle de nom simple> ::=
*<lettre> { <lettre> | <chiffre> | – }**

<lettre> ::=

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

<chiffre> ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<commentaire> ::=
<commentaire parenthésé>
| <commentaire fin de ligne>

<commentaire parenthésé> ::=
/ <chaîne de caractères> */*

<commentaire fin de ligne> ::=
– – <chaîne de caractères> <fin de ligne>

<chaîne de caractères> ::=
*{ <caractère> }**

<clause de directive> ::=
< > <directive> {, <directive> } < >*

<directive> ::=
<directive d'implémentation>

<nom> ::=
<représentation textuelle de nom>

<représentation textuelle de nom> ::=
<représentation textuelle de nom simple>
| <représentation textuelle de nom préfixe>

<représentation textuelle de nom préfixe> ::=
<préfixe> ! <représentation textuelle de nom simple>

<préfixe> ::=
*<préfixe simple> {! <préfixe simple> }**

<préfixe simple> ::=
<représentation textuelle de nom simple>

<définition> ::=
<représentation textuelle de nom simple>

<*liste de définitions*> ::=
<*définition*> {, <*définition*> }*

<*nom de champ*> ::=
<*représentation textuelle de nom simple*>

<*définition de nom de champ*> ::=
<*représentation textuelle de nom simple*>

<*liste de définitions de nom de champ*> ::=
<*définition de nom de champ*> {, <*définition de nom de champ*> }*

<*nom d'exception*> ::=
<*représentation textuelle de nom simple*>
| <*représentation textuelle de nom préfixe*>

<*nom de repère de texte*> ::=
<*représentation textuelle de nom simple*>
| <*représentation textuelle de nom préfixe*>

3 MODES ET CLASSES

<*définition de mode*> ::=
<*liste de définitions*> = <*mode définissant*>

<*mode définissant*> ::=
<*mode*>

<*énoncé de définition de synmode*> ::=
SYNMODE <*définition de mode*> {, <*définition de mode*> }*;

<*énoncé de définition de neumode*> ::=
NEWMODE <*définition de mode*> {, <*définition de mode*> }*;

<*mode*> ::=
[**READ**] <*mode non composé*>
| [**READ**] <*mode composé*>

<*mode non composé*> ::=
<*mode discret*>
| <*mode ensembliste*>
| <*mode repère*>
| <*mode procédure*>
| <*mode exemplaire*>
| <*mode de synchronisation*>
| <*mode d'entrée-sortie*>
| <*mode temporisation*>

<*mode discret*> ::=
<*mode entier*>
| <*mode booléen*>
| <*mode caractère*>
| <*mode ensemble*>
| <*mode intervalle*>

<*mode entier*> ::=
| <*nom de mode entier*>

<*mode booléen*> ::=
| <*nom de mode booléen*>

<*mode caractère*> ::=
| <*nom de mode caractère*>

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

<extension d'ensemble> ::=
<extension d'ensemble avec numéros>
| <extension d'ensemble sans numéros>

<extension d'ensemble avec numéros> ::=
<élément d'ensemble avec numéros> {, <élément d'ensemble avec numéros> }*

<élément d'ensemble avec numéros> ::=
<définition> = <expression littérale entière>

<extension d'ensemble sans numéros> ::=
<élément d'ensemble> {, <élément d'ensemble> }*

<élément d'ensemble> ::=
<définition>

<mode intervalle> ::=
<nom de mode discret> (<intervalle littéral>)
| **RANGE** (<intervalle littéral>)
| **BIN** (<expression littérale entière>)
| <nom de mode intervalle>

<intervalle littéral> ::=
<borne inférieure> : <borne supérieure>

<borne inférieure> ::=
<expression littérale discrète>

<borne supérieure> ::=
<expression littérale discrète>

<mode ensembliste> ::=
POWERSET <mode primitif>
| <nom de mode ensembliste>

<mode primitif> ::=
<mode discret>

<mode repère> ::=
<mode repère lié>
| <mode repère libre>
| <mode descripteur>

<mode repère lié> ::=
REF <mode repéré>
| <nom de mode repère lié>

<mode repéré> ::=
<mode>

<mode repère libre> ::=
| <nom de mode repère libre>

<mode descripteur> ::=
ROW <mode chaîne>
| **ROW** <mode rangée>
| **ROW** <nom de mode structure variable>
| <nom de mode descripteur>

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

<liste de paramètres> ::=
<spec de paramètre> {, <spec de paramètre> }*

<spec de paramètre> ::=
    <mode> [<attribut de paramètre>]

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

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

<attribut de résultat> ::=
    [**NONREF**] **LOC** [**DYNAMIC**]

<liste d'exceptions> ::=
    <nom d'exception> {, <nom d'exception> }*

<mode exemplaire> ::=
    | <nom de mode exemplaire>

<mode de synchronisation> ::=
    <mode événement>
    | <mode tampon>

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

<longueur d'événement> ::=
    <expression littérale entière>

<mode tampon> ::=
    BUFFER [(<longueur de tampon>)] <mode des éléments de tampon>
    | <nom de mode tampon>

<longueur de tampon> ::=
    <expression littérale entière>

<mode des éléments de tampon> ::=
    <mode>

<mode d'entrée-sortie> ::=
    <mode association>
    | <mode accès>
    | <mode texte>

<mode association> ::=
    | <nom de mode association>

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

<mode enregistrement> ::=
    <mode>

<mode d'indice> ::=
    <mode discret>
    | <intervalle de littéral>

<mode texte> ::=
    TEXT (<longueur de texte>) [<mode d'indice>] [**DYNAMIC**]

<longueur de texte> ::=
    <expression littérale entière>

<mode temporisation> ::=
    <mode durée>
    | <mode temps absolu>

<mode durée> ::=
<nom de mode durée>

<mode temps absolu> ::=
<nom de mode temps absolu>

<mode composé> ::=
<mode chaîne>
| <mode rangée>
| <mode structure>

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

<mode chaîne paramétré> ::=
<nom de mode chaîne originel> (<longueur de chaîne>)
| <nom de mode chaîne paramétré>

<nom de mode chaîne originel> ::=
<nom de mode chaîne>

<type de chaîne> ::=
BOOLS
| **CHARS**

<longueur de chaîne> ::=
<expression littérale entière>

<mode rangée> ::=
ARRAY (<mode d'indice> {, <mode d'indice> }*)
<mode des éléments> { <implantation d'élément> }*
| <mode rangée paramétré>
| <nom de mode rangée>

<mode rangée paramétré> ::=
<nom de mode rangée originel> (<indice supérieur>)
| <nom de mode rangée paramétré>

<nom de mode rangée originel> ::=
<nom de mode rangée>

<indice supérieur> ::=
<expression littérale discrète>

<mode des éléments> ::=
<mode>

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

<champ> ::=
<champ fixe>
| <choix de champs>

<champ fixe> ::=
<liste de définitions de noms de champ> <mode>
[<implantation de champ>]

<choix de champs> ::=
CASE [<liste de marqueurs>] **OF**
<champ à choisir> {, <champ à choisir> }*
[**ELSE** [<champ récurrent> {, <champ récurrent> }*]] **ESAC**

```
<champ à choisir> ::=
      [ <spécification d'étiquettes de cas> ]:
              [ <champ récurrent> {, <champ récurrent> }* ]

<liste de marqueurs> ::=
      <nom de champ marqueur> {, <nom de champ marqueur> }*

<champ récurrent> ::=
      <liste de définitions de noms de champ> <mode>
      [ <implantation de champ> ]

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

<nom de mode structure variable originel> ::=
      <nom de mode structure variable>

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

<implantation d'élément> ::=
      PACK | NOPACK | <pas>

<implantation de champ> ::=
      PACK | NOPACK | <pos>

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

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

<mot> ::=
      <expression littérale entière>

<taille de pas> ::=
      <expression littérale entière>

<bit initial> ::=
      <expression littérale entière>

<bit final> ::=
      <expression littérale entière>

<longueur> ::=
      <expression littérale entière>
```

4 LOCUS ET LEURS ACCÈS

```
<énoncé déclaratif> ::=
      DCL <déclaration> {, <déclaration> }*;

<déclaration> ::=
      <déclaration de locus>
    | <déclaration de loc-identité>

<déclaration de locus> ::=
      <liste de définitions> <mode> [ STATIC ] [ <initialisation> ]

<initialisation> ::=
      <initialisation domaniale>
    | <initialisation viagère>

<initialisation domaniale> ::=
      <symbole d'affectation> <valeur> [ <filet> ]
```

```
<initialisation viagère> ::=
      INIT <symbole d'affectation> <valeur constante>

<déclaration de loc-identité> ::=
      <liste de définitions> <mode> LOC [ DYNAMIC ] <symbole d'affectation>
            <locus> [ <filet> ]

<locus> ::=
      <nom d'accès>
    | <repère lié dérepéré>
    | <repère libre dérepéré>
    | <rangée dérepérée>
    | <élément de chaîne>
    | <tranche de chaîne>
    | <élément de rangée>
    | <tranche de rangée>
    | <champ de structure>
    | <appel de procédure rendant locus>
    | <appel d'opération prédéfinie rendant locus>
    | <conversion de locus>

<nom d'accès> ::=
      <nom de locus>
    | <nom de loc-identité>
    | <nom d'énumération de locus>
    | <nom de locus faire-avec>

<repère lié dérepéré> ::=
      <valeur primitive repère lié> -> [ <nom de mode> ]

<repère libre dérepéré> ::=
      <valeur primitive repère libre> -> <nom de mode>

<rangée dérepérée> ::=
      <valeur primitive rangée> ->

<élément de chaîne> ::=
      <locus chaîne> ( <élément de début> )

<élément de début> ::=
      <expression entière>

<tranche de chaîne> ::=
      <locus chaîne> ( <élément de gauche> : <élément de droite> )
    | <locus chaîne> ( <élément de début> UP <taille de tranche> )

<élément de gauche> ::=
      <expression entière>

<élément de droite> ::=
      <expression entière>

<taille de tranche> ::=
      <expression entière>

<élément de rangée> ::=
      <locus rangée> ( <liste d'expressions> )

<liste d'expressions> ::=
      <expression> {, <expression> }*

<tranche de rangée> ::=
      <locus rangée> ( <élément inférieur> : <élément supérieur> )
    | <locus rangée> ( <premier élément> UP <taille de rangée> )
```

```
<élément inférieur> ::=
      <expression>

<élément supérieur> ::=
      <expression>

<premier élément> ::=
      <expression>

<champ de structure> ::=
      <locus structure> . <nom de champ>

<appel de procédure rendant locus> ::=
      <appel de procédure rendant locus>

<appel d'opération prédéfinie rendant locus> ::=
      <appel d'opération prédéfinie rendant locus>

<conversion de locus> ::=
      <nom de mode> ( <locus de mode statique> )
```

5 VALEURS ET LEURS OPÉRATIONS

```
<énoncé de définition de synonyme> ::=
      SYN <définition de synonyme> {, <définition de synonyme> }*;

<définition de synonyme> ::=
      <liste de définitions> [ <mode> ] = <valeur constante>

<valeur primitive> ::=
      <contenu de locus>
    | <nom de valeur>
    | <littéral>
    | <multiplet>
    | <valeur élément de chaîne>
    | <valeur tranche de chaîne>
    | <valeur élément de rangée>
    | <valeur tranche de rangée>
    | <valeur champ de structure>
    | <conversion d'expression>
    | <appel de procédure rendant valeur>
    | <appel d'opération prédéfinie rendant valeur>
    | <expression démarrer>
    | <opérateur nullaire>
    | <expression parenthésée>

<contenu de locus> ::=
      <locus>

<nom de valeur> ::=
      <nom de synonyme>
    | <nom d'énumération de valeur>
    | <nom de valeur faire-avec>
    | <nom de valeur reçue>
    | <nom de procédure générale>

<littéral> ::=
      <littéral d'entier>
    | <littéral de booléen>
    | <littéral de caractère>
    | <littéral d'ensemble>
    | <littéral de vide>
    | <littéral de chaîne de caractères>
    | <littéral de chaîne de bits>
```

```
<littéral d'entier> ::=
        <littéral décimal d'entier>
      | <littéral binaire d'entier>
      | <littéral octal d'entier>
      | <littéral hexadécimal d'entier>

<littéral décimal d'entier> ::=
        [ { D | d }' ] <chiffre> | _ }+

<littéral binaire d'entier> ::=
        { B | b }' { 0 | 1 | _ }+

<littéral octal d'entier> ::=
        { O | o }' { <chiffre octal> | _ }+

<littéral hexadécimal d'entier> ::=
        { H | h }' { <chiffre hexadécimal> | _ }+

<chiffre hexadécimal> ::=
        <chiffre> | A | B | C | D | E | F | a | b | c | d | e | f

<chiffre octal> ::=
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<littéral de booléen> ::=
        <nom littéral de booléen>

<littéral de caractère> ::=
        '<caractère> | <séquence de contrôle>'

<littéral d'ensemble> ::=
        <nom d'élément d'ensemble>

<littéral de vide> ::=
        <nom littéral de vide>

<littéral de chaîne de caractères> ::=
        "{ <caractère non réservé> | <citation> | <séquence de contrôle> }*"

<citation> ::=
        " "

<séquence de contrôle> ::=
        ^( <expression littérale entière> {, <expression littérale entière> }* )
      | ^ <caractère non spécial>
      | ^ ^

<littéral de chaîne de bits> ::=
        <littéral binaire de chaîne de bits>
      | <littéral octal de chaîne de bits>
      | <littéral hexadécimal de chaîne de bits>

<littéral binaire de chaîne de bits> ::=
        { B | b }' { 0 | 1 | _ }*'

<littéral octal de chaîne de bits> ::=
        { O | o }' { <chiffre octal> | _ }*'

<littéral hexadécimal de chaîne de bits> ::=
        { H | h }' { <chiffre hexadécimal> | _ }*'

<multiplet> ::=
        [ <nom de mode> ] (: { <multiplet ensembliste> | <multiplet de rangée>
      | <multiplet de structure> } :)

<multiplet ensembliste> ::=
        [ { <expression> | <intervalle> } {, { <expression> | <intervalle> } }* ]

<intervalle> ::=
        <expression> : <expression>
```

<multiplet de rangée> ::=
<multiplet de rangée sans indices>
| *<multiplet de rangée avec indices>*

<multiplet de rangée sans indices> ::=
*<valeur> {, <valeur> }**

<multiplet de rangée avec indices> ::=
*<liste d'étiquettes de cas> : <valeur> {, <liste d'étiquettes de cas> : <valeur> }**

<multiplet de structure> ::=
<multiplet de structure sans noms de champ>
| *<multiplet de structure avec noms de champ>*

<multiplet de structure sans noms de champ> ::=
*<valeur> {, <valeur> }**

<multiplet de structure avec noms de champ> ::=
*<liste de noms de champ> : <valeur> {, <liste de noms de champ> : <valeur> }**

<liste de noms de champ> ::=
*.<nom de champ> {, .<nom de champ> }**

<valeur élément de chaîne> ::=
<valeur primitive chaîne> (<élément de début>)

<valeur tranche de chaîne> ::=
<valeur primitive chaîne> (<élément de gauche> : <élément de droite>)
| *<valeur primitive chaîne> (<élément de début>* **UP** *<taille de tranche>)*

<valeur élément de rangée> ::=
<valeur primitive rangée> (<liste d'expressions>)

<valeur tranche de rangée> ::=
<valeur primitive rangée> (<élément inférieur> : <élément supérieur>)
| *<valeur primitive rangée> (<premier élément>* **UP** *<taille de tranche>)*

<valeur champ de structure> ::=
<valeur primitive structure> . <nom de champ>

<conversion d'expression> ::=
<nom de mode> (<expression>)

<appel de procédure rendant valeur> ::=
<appel de procédure rendant valeur>

<appel d'opération prédéfinie rendant valeur> ::=
| *<appel d'opération prédéfinie rendant valeur>*

<expression démarrer> ::=
START *<nom de processus> ([<liste de paramètres effectifs>])*

<opérateur nullaire> ::=
THIS

<expression parenthésée> ::=
(<expression>)

<valeur> ::=
<expression>
| *<valeur indéfinie>*

<valeur indéfinie> ::=
*
| *<nom de synonyme indéfini>*

<expression> ::=
<opérande-0>
<expression conditionnelle>

<expression conditionnelle> ::=
| **IF** *<expression booléenne>* *<solution alors>*
<solution sinon> **FI**
| **CASE** *<liste de sélecteurs de cas>* **OF** { *<solution cas de valeur>* }+
[**ELSE** *<sous-expression>*] **ESAC**

<solution alors> ::=
THEN *<sous-expression>*

<solution sinon> ::=
ELSE *<sous-expression>*
| **ELSIF** *<expression booléenne>*
<solution alors> *<solution sinon>*

<sous-expression> ::=
<expression>

<solution cas de valeur> ::=
<spécification d'étiquettes de cas> : *<sous-expression>*;

<opérande-0> ::=
<opérande-1>
| *<sous-opérande-0>* { **OR** | **ORIF** | **XOR** } *<opérande-1>*

<sous-opérande-0> ::=
<opérande-0>

<opérande-1> ::=
<opérande-2>
| *<sous-opérande-1>* { **AND** | **ANDIF** } *<opérande-2>*

<sous-opérande-1> ::=
<opérande-1>

<opérande-2> ::=
<opérande-3>
| *<sous-opérande-2>* *<opérateur-3>* *<opérande-3>*

<sous-opérande-2> ::=
<opérande-2>

<opérateur-3> ::=
<opérateur relationnel>
| *<opérateur d'appartenance>*
| *<opérateur d'inclusion ensembliste>*

<opérateur relationnel> ::=
= | /= | > | >= | < | <=

<opérateur d'appartenance> ::=
IN

<opérateur d'inclusion ensembliste> ::=
<= | >= | < | >

<opérande-3> ::=
<opérande-4>
| *<sous-opérande-3>* *<opérateur-4>* *<opérande-4>*

<sous-opérande-3> ::=
<opérande-3>

<opérateur-4> ::=
<opérateur arithmétique additif>
| *<opérateur de concaténation de chaîne>*
| *<opérateur de différence ensembliste>*

<opérateur arithmétique additif> ::=
+ | −

<opérateur de concaténation de chaîne> ::=
    / /

<opérateur de différence ensembliste> ::=
    –

<opérande-4> ::=
    <opérande-5>
  | <sous-opérande-4> <opérateur arithmétique multiplicatif> <opérande-5>

<sous-opérande-4> ::=
    <opérande-4>

<opérateur arithmétique multiplicatif> ::=
    * | / | **MOD** | **REM**

<opérande-5> ::=
    [<opérateur unaire>] <opérande-6>

<opérateur unaire> ::=
    – | **NOT**
  | <opérateur de répétition de chaîne>

<opérateur de répétition de chaîne> ::=
    (<expression littérale entière>)

<opérande-6> ::=
    <locus repéré>
  | <expression recevoir>
  | <valeur primitive>

<locus repéré> ::=
    – > <locus>

<expression recevoir> ::=
    RECEIVE <locus tampon>

6 ACTIONS

<énoncé d'action> ::=
    [<définition> :] <action> [<filet>] [<représentation textuelle de nom simple>];
  | <module>
  | <module de spec>
  | <module de contexte>

<action> ::=
    <action parenthésée>
  | <action d'affectation>
  | <action appeler>
  | <action sortir>
  | <action revenir>
  | <action résulter>
  | <action aller>
  | <action affirmer>
  | <action vide>
  | <action démarrer>
  | <action arrêter>
  | <action mettre en attente>
  | <action continuer>
  | <action envoyer>
  | <action causer>

<action parenthésée> ::=
<action conditionnelle>
| *<action de cas>*
| *<action faire>*
| *<bloc début-fin>*
| *<action mettre en attente et choisir>*
| *<action recevoir et choisir>*
| *<action de temporisation>*

<action d'affectation> ::=
<action d'affectation simple>
| *<action d'affectation multiple>*

<action d'affectation simple> ::=
<locus> <symbole d'affectation> <valeur>
| *<locus> <opérateur affectant> <expression>*

<action d'affectation multiple> ::=
<locus> {, *<locus>* }+ *<symbole d'affectation> <valeur>*

<opérateur affectant> ::=
<opérateur binaire fermé> <symbole d'affectation>

<opérateur binaire fermé> ::=
OR | **XOR** | **AND**
| *<opérateur de différence ensembliste>*
| *<opérateur arithmétique additif>*
| *<opérateur arithmétique multiplicatif>*
| *<opérateur de concaténation de chaîne>*

<symbole d'affectation> ::=
:=

<action conditionnelle> ::=
IF *<expression booléenne> <clause alors>* [*<clause sinon>*] **FI**

<clause alors> ::=
THEN *<liste d'énoncés d'action>*

<clause sinon> ::=
ELSE *<liste d'énoncés d'action>*
| **ELSIF** *<expression booléenne> <clause alors>* [*<clause sinon>*]

<action de cas> ::=
CASE *<liste de sélecteurs de cas>* **OF** [*<liste d'intervalles>;*] { *<cas à choisir>* }+
[**ELSE** *<liste d'énoncés d'action>*] **ESAC**

<liste de sélecteurs de cas> ::=
<expression discrète> {, *<expression discrète>* }*

<liste d'intervalles> ::=
<nom de mode discret> {, *<nom de mode discret>* }*

<cas à choisir> ::=
<spécification d'étiquettes de cas> : <liste d'énoncés d'action>

<action faire> ::=
DO [*<partie de commande>* ;] *<liste d'énoncés d'action>* **OD**

<partie de commande> ::=
<commande pour> [*<commande tandis>*]
| *<commande tandis>*
| *<partie avec>*

<commande pour> ::=
FOR { *<itération>* {, *<itération>* }* | **EVER** }

<*itération*> ::=
<*énumération de valeur*>
| <*énumération de locus*>

<*énumération de valeur*> ::=
<*énumération par pas*>
| <*énumération par intervalle*>
| <*énumération ensembliste*>

<*énumération par pas*> ::=
<*compteur de boucle*> <*symbole d'affectation*>
<*valeur initiale*> [<*valeur de pas*>] [**DOWN**] <*valeur finale*>

<*compteur de boucle*> ::=
<*définition*>

<*valeur initiale*> ::=
<*expression discrète*>

<*valeur de pas*> ::=
BY <*expression entière*>

<*valeur finale*> ::=
TO <*expression discrète*>

<*énumération par intervalle*> ::=
<*compteur de boucle*> [**DOWN**] **IN** <*nom de mode discret*>

<*énumération ensembliste*> ::=
<*compteur de boucle*> [**DOWN**] **IN** <*expression ensembliste*>

<*énumération de locus*> ::=
<*compteur de boucle*> [**DOWN**] **IN** <*objet composé*>

<*objet composé*> ::=
<*locus rangée*>
| <*expression rangée*>
| <*locus chaîne*>
| <*expression chaîne*>

<*commande tandis*> ::=
WHILE <*expression booléenne*>

<*partie avec*> ::=
WITH <*commande avec*> {, <*commande avec*> }*

<*commande avec*> ::=
<*locus structure*>
| <*valeur primitive structure*>

<*action sortir*> ::=
EXIT <*nom d'étiquette*>

<*action appeler*> ::=
<*appel de procédure*>
| <*appel d'opération prédéfinie*>

<*appel de procédure*> ::=
{ <*nom de procédure*> | <*valeur primitive procédure*> }
([<*liste de paramètres effectifs*>] *)*

<*liste de paramètres effectifs*> ::=
<*paramètre effectif*> {, <*paramètre effectif*> }*

<*paramètre effectif*> ::=
<*valeur*>
| <*locus*>

```
<appel d'opération prédéfinie> ::=
      <nom d'opération prédéfinie> ([ <liste de paramètres d'opération prédéfinie> ] )

<liste de paramètres d'opération prédéfinie> ::=
      <paramètre d'opération prédéfinie> {, <paramètre d'opération prédéfinie> }*

<paramètre d'opération prédéfinie> ::=
      <valeur>
    | <locus>
    | <nom non réservé> [ ( <liste de paramètres d'opération prédéfinie> )]

<action revenir> ::=
      RETURN [ <résultat> ]

<action résulter> ::=
      RESULT <résultat>

<résultat> ::=
      <valeur>
    | <locus>

<action aller> ::=
      GOTO <nom d'étiquette>

<action affirmer> ::=
      ASSERT <expression booléenne>

<action vide> ::=
      <vide>

<vide> ::=

<action causer> ::=
      CAUSE <nom d'exception>

<action démarrer> ::=
      <expression démarrer>

<action arrêter> ::=
      STOP

<action continuer> ::=
      CONTINUE <locus événement>

<action mettre en attente> ::=
      DELAY <locus événement> [ <priorité> ]

<priorité> ::=
      PRIORITY <expression littérale entière>

<action mettre en attente et choisir> ::=
      DELAY CASE [ SET <locus exemplaire> [ <priorité> ]; | <priorité> ;]
      { <événement à choisir> }+
      ESAC

<événement à choisir> ::=
      ( <liste d'événements> ) : <liste d'énoncés d'action>

<liste d'événements> ::=
      <locus événement> {, <locus événement> }*

<action envoyer> ::=
      <action envoyer signal>
    | <action envoyer tampon>

<action envoyer signal> ::=
      SEND <nom de signal> [ ( <valeur> {, <valeur> }* )]
      [ TO <valeur primitive exemplaire> ] [ <priorité> ]
```

```
<action envoyer tampon> ::=
      SEND <locus tampon> ( <valeur> ) [ <priorité> ]

<action recevoir et choisir> ::=
        <action recevoir signal et choisir>
      | <action recevoir tampon et choisir>

<action recevoir signal et choisir> ::=
      RECEIVE CASE [ SET <locus exemplaire> ;]
      { <signal à choisir> }+
      [ ELSE <liste d'énoncés d'action> ] ESAC

<signal à choisir> ::=
      ( <nom de signal> [ IN <liste de définitions> ] ) : <liste d'énoncés d'action>

<action recevoir tampon et choisir> ::=
      RECEIVE CASE [ SET <locus exemplaire> ;]
      { <tampon à choisir> }+
      [ ELSE <liste d'énoncés d'action> ]
      ESAC

<tampon à choisir> ::=
      ( <locus tampon> IN <définition> ) : <liste d'énoncés d'action>

<appel d'opération prédéfinie CHILL> ::=
        <appel d'opération prédéfinie simple CHILL>
      | <appel d'opération prédéfinie rendant locus CHILL>
      | <appel d'opération prédéfinie rendant valeur CHILL>

<appel d'opération prédéfinie simple CHILL> ::=
        <appel d'opération prédéfinie terminer>
      | <appel d'opération prédéfinie simple d'e/s>
      | <appel d'opération prédéfinie simple de temporisation>

<appel d'opération prédéfinie rendant locus CHILL> ::=
        <appel d'opération prédéfinie rendant locus d'e/s>

<appel d'opération prédéfinie rendant valeur CHILL> ::=
        NUM ( <expression discrète> )
      | PRED ( <expression discrète> )
      | SUCC ( <expression discrète> )
      | ABS ( <expression entière> )
      | CARD ( <expression ensembliste> )
      | MAX ( <expression ensembliste> )
      | MIN ( <expression ensembliste> )
      | SIZE ( { <locus> | <argument de mode> } )
      | UPPER ( <argument pour upper lower> )
      | LOWER ( <argument pour upper lower> )
      | LENGTH ( <argument de longueur> )
      | <appel d'opération prédéfinie affecter>
      | <appel d'opération prédéfinie rendant valeur d'e/s>
      | <appel d'opération prédéfinie de valeur temps>

<argument de mode> ::=
        <nom de mode>
      | <nom de mode rangée> ( <expression> )
      | <nom de mode chaîne> ( <expression entière> )
      | <nom de mode structure variable> ( <liste d'expressions> )
```

<argument pour upper lower> ::=
<locus rangée>
| *<expression rangée>*
| *<nom de mode rangée>*
| *<locus chaîne>*
| *<expression chaîne>*
| *<nom de mode chaîne>*
| *<locus discret>*
| *<expression discrète>*
| *<nom de mode discret>*

<argument longueur> ::=
<locus chaîne>
| *<expression chaîne>*

<appel d'opération prédéfinie affecter> ::=
GETSTACK *(<argument de mode>* [, *<valeur>*] *)*
| ALLOCATE *(<argument de mode>* [, *<valeur>*] *)*

<appel d'opération prédéfinie terminer> ::=
TERMINATE *(<valeur primitive repère>)*

7 ENTRÉE ET SORTIE

<appel d'opération prédéfinie rendant valeur d'e/s> ::=
<appel d'opération prédéfinie attribut d'association>
| *<appel d'opération prédéfinie est associé>*
| *<appel d'opération prédéfinie attribut d'accès>*
| *<appel d'opération prédéfinie lire article>*
| *<appel d'opération prédéfinie obtenir texte>*

<appel d'opération prédéfinie simple d'e/s> ::=
<appel d'opération prédéfinie dissocier>
| *<appel d'opération prédéfinie modification>*
| *<appel d'opération prédéfinie connecter>*
| *<appel d'opération prédéfinie déconnecter>*
| *<appel d'opération prédéfinie écrire article>*
| *<appel d'opération prédéfinie texte>*
| *<appel d'opération prédéfinie fixer texte>*

<appel d'opération prédéfinie rendant locus d'e/s> ::=
<appel d'opération prédéfinie associer>

<appel d'opération prédéfinie associer> ::=
ASSOCIATE *(<locus association>* [, *<liste de paramètres associer>*] *)*

<appel d'opération prédéfinie est associé> ::=
ISASSOCIATED *(<locus association>)*

<liste de paramètres pour associer> ::=
<paramètre pour associer> {, *<paramètre pour associer>* }*

<paramètre pour associer> ::=
<locus>
| *<valeur>*

<appel d'opération prédéfinie dissocier> ::=
DISSOCIATE *(<locus association>)*

<appel d'opération prédéfinie attribut d'association> ::=
EXISTING *(<locus association>)*
| READABLE *(<locus association>)*
| WRITEABLE *(<locus association>)*
| INDEXABLE *(<locus association>)*
| SEQUENCIBLE *(<locus association>)*
| VARIABLE *(<locus association>)*

```
<appel d'opération prédéfinie modification> ::=
      CREATE ( <locus association> )
    | DELETE ( <locus association> )
    | MODIFY ( <locus association> [, <liste de paramètres pour modifier> ] )

<liste de paramètres pour modifier> ::=
      <paramètre pour modifier> {, <paramètre pour modifier> }*

<paramètre pour modifier> ::=
      <valeur>
    | <locus>

<appel d'opération prédéfinie connecter> ::=
      CONNECT ( <locus transfert>, <locus association>, <expression usage>
      [, <expression positionnement> [, <expression indice> ]] )

<locus transfert> ::=
      <locus accès>
    | <locus texte>

<expression usage> ::=
      <expression>

<expression positionnement> ::=
      <expression>

<expression indice> ::=
      <expression>

<appel d'opération prédéfinie déconnecter> ::=
      DISCONNECT ( <locus transfert> )

<appel d'opération prédéfinie attribut d'accès> ::=
      GETASSOCIATION ( <locus transfert> )
    | GETUSAGE ( <locus transfert> )
    | OUTOFFILE ( <locus transfert> )

<appel d'opération prédéfinie lire article> ::=
      READRECORD ( <locus accès> [, <expression indice> ] [, <locus de lecture> ] )

<appel d'opération prédéfinie écrire article> ::=
      WRITERECORD ( <locus accès> [, <expression indice> ] <expression écrire> )

<locus de lecture> ::=
      <locus de mode statique>

<expression écrire> ::=
      <expression>

<appel d'opération prédéfinie de texte> ::=
      READTEXT ( <liste d'arguments d'e/s de texte> )
    | WRITETEXT ( <liste d'arguments d'e/s de texte> )

<liste d'arguments d'e/s de texte> ::=
      <argument de texte> [, <expression indice> ],
      <argument de format> [, <liste d'e/s> ]

<argument de texte> ::=
      <locus texte>
    | <locus chaîne de caractères>
    | <expression chaîne de caractères>

<argument de format> ::=
      <expression chaîne de caractères>

<liste d'e/s> ::=
      <élément de liste d'e/s> {, <élément de liste d'e/s> }*
```

<élément de liste d'e/s> ::=
<argument de valeur>
| <argument de locus>

<argument de locus> ::=
<locus discret>
| <locus chaîne>

<argument de valeur> ::=
<expression discrète>
| <expression chaîne>

<chaîne de commande de format> ::=
[<texte de format>] { <spécification de format> [<texte de format>] }*

<texte de format> ::=
{ <caractère non-pourcent> | <pourcent> }

<pourcent> ::=
% %

<spécification de format> ::=
% [<facteur de répétition>] <élément de format>

<facteur de répétition> ::=
{ <chiffre> }+

<élément de format> ::=
<clause de format>
| <clause parenthésée>

<clause de format> ::=
<code de commande> [%.]

<code de commande> ::=
<clause de conversion>
| <clause d'édition>
| <clause d'e/s>

<clause parenthésée> ::=
(<chaîne de commande de format> %)

<clause de conversion> ::=
<code de conversion> { <qualificatif de conversion> }*
[<largeur de clause>]

<code de conversion> ::=
B | O | H | C

<qualificatif de conversion> ::=
L | E | P <caractère>

<largeur de clause> ::=
{ <chiffre> }+ | V

<clause d'édition> ::=
<code d'édition> [<largeur de clause>]

<code d'édition> ::=
X | < | > | T

<clause d'e/s> ::=
<code d'e/s>

<code d'e/s> ::=
/ | - | + | ? | ! | =

<appel d'opération prédéfinie obtenir texte> ::=
GETTEXTRECORD (<locus texte>)
| GETTEXTINDEX (<locus texte>)
| GETTEXTACCESS (<locus texte>)
| EOLN (<locus texte>)

<appel d'opération prédéfinie fixer texte> ::=
SETTEXTRECORD (<locus texte>, <locus chaîne de caractères>)
| SETTEXTINDEX (<locus texte>, <expression entière>)
| SETTEXTACCESS (<locus texte>, <locus accès>)

8 FILETS D'EXCEPTION

<filet> ::=
ON { <choix d'exceptions> }* [**ELSE** <liste d'énoncés d'action>] **END**

<choix d'exceptions> ::=
(<liste d'exceptions>) : <liste d'énoncés d'action>

9 TEMPORISATION

<action de temporisation> ::=
<action de temporisation relative>
| <action de temporisation absolue>
| <action de temporisation cyclique>

<action de temporisation relative> ::=
AFTER <valeur primitive durée> [**DELAY**] **IN**
<liste d'énoncés d'action> <filet de temporisation> **END**

<filet de temporisation> ::=
TIMEOUT <liste d'énoncés d'action>

<action de temporisation absolue> ::=
AT <valeur primitive temps absolu> **IN**
<liste d'énoncés d'action> <filet de temporisation> **END**

<action de temporisation cyclique> ::=
CYCLE <valeur primitive durée> **IN**
<liste d'énoncés d'action> **END**

<appel d'opération prédéfinie de valeur temps> ::=
<appel d'opération prédéfinie de durée>
| <appel d'opération prédéfinie de temps absolu>

<appel d'opération prédéfinie de durée> ::=
MILLISECS (<expression entière>)
| SECS (<expression entière>)
| MINUTES (<expression entière>)
| HOURS (<expression entière>)
| DAYS (<expression entière>)

<appel d'opération prédéfinie de temps absolu> ::=
ABSTIME ([[[[[<expression année> ,] <expression mois> ,]
<expression jour> ,] <expression heure> ,]
<expression minute> ,] <expression seconde>])

<expression année> ::=
<expression entière>

<expression mois> ::=
<expression entière>

```
<expression jour> ::=
        <expression entière>

<expression heure> ::=
        <expression entière>

<expression minute> ::=
        <expression entière>

<expression seconde> ::=
        <expression entière>

<appel d'opération prédéfinie simple de temporisation> ::=
        WAIT ( )
      | EXPIRED ( )
      | INTTIME <valeur primitive temps absolu> ,[[[[ <locus année>
        <locus mois> ,] <locus jour> ,]<locus heure> ,] <locus minute> ,] <locus seconde> )

<locus année> ::=
        <locus entier>

<locus mois> ::=
        <locus entier>

<locus jour> ::=
        <locus entier>

<locus heure> ::=
        <locus entier>

<locus minute> ::=
        <locus entier>

<locus seconde> ::=
        <locus entier>
```

10 STRUCTURE DE PROGRAMME

```
<corps début-fin> ::=
        <liste d'énoncés informatifs> <liste d'énoncés d'action>

<corps de procédure> ::=
        <liste d'énoncés informatifs> <liste d'énoncés d'action>

<corps de processus> ::=
        <liste d'énoncés informatifs> <liste d'énoncés d'action>

<corps de module> ::=
        { <énoncé informatif> | <énoncé de visibilité> | <région> | <région de spec> }*
                <liste d'énoncés d'action>

<corps de région> ::=
        { <énoncé informatif> | <énoncé de visibilité> }*

<corps de module de spec> ::=
        { <quasi-énoncé informatif> | <énoncé de visibilité> | <module de spec> |
        <région de spec>}*

<corps de région de spec> ::=
        { <quasi-énoncé informatif> | <énoncé de visibilité> }*

<corps de contexte> ::=
        { <quasi-énoncé informatif> | <énoncé de visibilité> | <module de spec> |
        <région de spec> }*
```

<liste d'énoncés d'action> ::=
{ *<énoncé d'action>* }*

<liste d'énoncés informatifs> ::=
{ *<énoncé informatif>* }*

<énoncé informatif> ::=
<énoncé déclaratif>
| *<énoncé définissant>*

<énoncé définissant> ::=
<énoncé de définition de synmode >
| *<énoncé de définition de neumode>*
| *<énoncé de définition de synonyme>*
| *<énoncé de définition de procédure>*
| *<énoncé de définition de processus>*
| *<énoncé de définition de signal>*
| *<vide>* ;

<bloc début-fin> ::=
BEGIN *<corps début-fin>* **END**

<énoncé de définition de procédure> ::=
<définition> : <définition de procédure>
[*<filet>*] [*<représentation textuelle de nom simple>*];

<définition de procédure> ::=
PROC *(* [*<liste de paramètres formels>*] *)* [*<spec de résultat>*]
[**EXCEPTIONS** *(<liste d'exceptions>)*] *<liste d'attributs de procédure>*
<corps de procédure> **END**

<liste de paramètres formels> ::=
<paramètre formel> {, *<paramètre formel>* }*

<paramètre formel> ::=
<liste de définitions> <spec de paramètre>

<liste d'attributs de procédure> ::=
[*<généralité>*] [**RECURSIVE**]

<généralité> ::=
GENERAL
| **SIMPLE**
| **INLINE**

<énoncé de définition de processus> ::=
<définition> : <définition de processus>
[*<filet>*] [*<représentation textuelle de nom simple>*];

<définition de processus> ::=
PROCESS *(* [*<liste de paramètres formels>*] *)* *<corps de processus>* **END**

<module> ::=
[*<liste de contextes>*] [*<définition>* :]
MODULE [**BODY**] *<corps de module>* **END** [*<filet>*] [*<représentation textuelle de nom simple>*];
| *<modulion distant>*

<région> ::=
[*<liste de contextes>*] [*<définition>* :] **REGION** [**BODY**] *<corps de région>* **END**
[*<filet>*] [*<représentation textuelle de nom simple>*];
| *<modulion distant>*

<programme> ::=
{ *<module>* | *<module de spec>* | *<région>* | *<région de spec>* }$^+$

<modulion distant> ::=
[*<représentation textuelle de nom simple>* :]
REMOTE *<indicateur de fragment>* ;

<spec distante> ::=
[*<représentation textuelle de nom simple>* :]
SPEC REMOTE *<indicateur de fragment>* ;

<contexte distant> ::=
CONTEXT REMOTE *<indicateur de fragment>*
[*<corps de contexte>*] **FOR**

<module de contexte> ::=
CONTEXT MODULE REMOTE *<indicateur de fragment>* ;

<indicateur de fragment> ::=
<littéral de chaîne de caractères>
| *<nom repère de texte>*
| *<vide>*

<module de spec> ::=
<module de spec simple>
| *<spec de module>*
| *<spec distante>*

<module de spec simple> ::=
[*<liste de contextes>*] [*<représentation textuelle de nom simple>* :] **SPEC MODULE**
<corps de module de spec> **END** [*<représentation textuelle de nom simple>*];

<spec de module> ::=
[*<liste de contextes>*] *<représentation textuelle de nom simple>* : **MODULE SPEC**
<corps de module de spec> **END** [*<représentation textuelle de nom simple>*];

<région de spec> ::=
<région de spec simple>
| *<spec de région>*
| *<spec distante>*

<région de spec simple> ::=
[*<liste de contextes>*] [*<représentation textuelle de nom simple>* :] **SPEC REGION**
<corps de région de spec> **END** [*<représentation textuelle de nom simple>*];

<spec de région> ::=
[*<liste de contextes>*] *<représentation textuelle de nom simple>* : **REGION SPEC**
<corps de région de spec> **END** [*<représentation textuelle de nom simple>*];

<liste de contextes> ::=
<contexte> { *<contexte>* }*
| *<contexte distant>*

<contexte> ::=
CONTEXT *<corps de contexte>* **FOR**

<quasi-énoncé informatif> ::=
<quasi-énoncé déclaratif>
| *<quasi-énoncé définissant>*

<quasi-énoncé déclaratif> ::=
DCL *<quasi-déclaration>* {, *<quasi-déclaration>* }*;

<quasi-déclaration> ::=
<quasi-déclaration de locus>
| *<quasi-déclaration de loc-identité>*

<quasi-déclaration de locus> ::=
<liste de définitions> *<mode>* [**STATIC**]

```
<quasi-déclaration de loc-identité> ::=
      <liste de définitions> <mode> LOC [ NONREF ] [ DYNAMIC ]

<quasi-énoncé définissant> ::=
      <énoncé de définition de synmode>
    | <énoncé de définition de neumode>
    | <énoncé de définition de synonyme>
    | <quasi-énoncé de définition de synonyme>
    | <quasi-énoncé de définition de procédure>
    | <quasi-énoncé de définition de processus>
    | <énoncé de définition de signal>
    | <vide> ;

<quasi-énoncé de définition de synonyme> ::=
      SYN <quasi-définition de synonyme> {, <quasi-définition de synonyme> }*;

<quasi-définition de synonyme> ::=
      <liste de définitions> { <mode> = [ <valeur constante>] | [ <mode> ] =
      <expression littérale> }

<quasi-énoncé de définition de procédure> ::=
      <définition> : PROC ( [ <quasi-liste de paramètres formels> ] )
      [ <spec de résultat> ] [ EXCEPTIONS ( <liste d'exceptions> ) ]
      <liste d'attributs de procédure>
      END [ <représentation textuelle de nom simple> ];

<quasi-liste de paramètres formels> ::=
      <quasi-paramètre formel> {, <quasi-paramètre formel> }*

<quasi-paramètre formel> ::=
      <représentation textuelle de nom simple>
      {, <représentation textuelle de nom simple> }* <spec de paramètre>

<quasi-énoncé de définition de processus> ::=
      <définition> : PROCESS ( [ <quasi-liste de paramètres formels> ] )
      END [ <représentation textuelle de nom simple> ];

<quasi-énoncé de définition de signal> ::=
      SIGNAL <quasi-définition de signal> {, <quasi-définition de signal> }*;

<quasi-définition de signal> ::=
      <définition> [= ( <mode> {, <mode> }* ) ] [ TO ]
```

11 EXÉCUTION CONCURRENTE

```
<énoncé de définition de signal> ::=
      SIGNAL <définition de signal> {, <définition de signal> }*;

<définition de signal> ::=
      <définition> [= ( <mode> {, <mode> }* ) ] [ TO <nom de processus> ]
```

12 PROPRIÉTÉS SÉMANTIQUES GÉNÉRALES

```
<énoncé de visibilité> ::=
      <énoncé d'octroi>
    | <énoncé de saisie>

<clause renommer préfixe> ::=
      ( <ancien préfixe> -> <nouveau préfixe> ) ! <postfixe>

<ancien préfixe> ::=
      <préfixe>
    | <vide>

<nouveau préfixe> ::=
      <préfixe>
    | <vide>
```

```
<postfixe> ::=
      <postfixe de saisie> {, <postfixe de saisie> }*
    | <postfixe d'octroi> {, <postfixe d'octroi> }*

<énoncé d'octroi> ::=
      GRANT <clause renommer préfixe> {, <clause renommer préfixe> }*;
    | GRANT <fenêtre d'octroi> [ <clause préfixe> ];

<fenêtre d'octroi> ::=
      <postfixe d'octroi> {, <postfixe d'octroi> }*

<postfixe d'octroi> ::=
      <représentation textuelle de nom>
    | <représentation textuelle de nom de neumode> <clause d'interdiction>
    | [ <préfixe> ! ] ALL

<clause préfixe> ::=
      PREFIXED [ <préfixe> ]

<clause d'interdiction> ::=
      FORBID { <liste de noms d'interdiction> | ALL }

<liste de noms d'interdiction> ::=
      ( <nom de champ> {, <nom de champ> }* )

<énoncé de saisie> ::=
      SEIZE <clause renommer préfixe> {, <clause renommer préfixe> }*;
    | SEIZE <fenêtre de saisie> [ <clause préfixe> ];

<fenêtre de saisie> ::=
      <postfixe de saisie> {, <postfixe de saisie> }*

<postfixe de saisie> ::=
      <représentation textuelle de nom>
    | [ <préfixe> ! ] ALL

<spécification d'étiquettes de cas> ::=
      <liste d'étiquettes de cas> {, <liste d'étiquettes de cas> }*

<liste d'étiquettes de cas> ::=
      ( <étiquette de cas> {, <étiquette de cas> }* )
    | <indifférent>

<étiquette de cas> ::=
      <expression littérale discrète>
    | <intervalle littéral>
    | <nom de mode discret>
    | ELSE

<indifférent> ::=
      (*)
```

APPENDICE G

Index des règles de production

non-terminal	défini dans la section	page	employé page(s)
<action affirmer>	6.10	87	75
<action aller>	6.9	87	75
<action appeler>	6.7	84	75
<action arrêter>	6.14	88	75
<action causer>	6.12	88	75
<action conditionnelle>	6.3	77	75
<action continuer>	6.15	88	75
<action d'affectation multiple>	6.2	75	75
<action d'affectation simple>	6.2	75	75
<action d'affectation>	6.2	75	75
<action de cas>	6.4	78	75
<action de temporisation absolue>	9.3.2	123	122
<action de temporisation cyclique>	9.3.3	123	122
<action de temporisation relative>	9.3.1	122	122
<action de temporisation>	9.3	122	75
<action démarrer>	6.13	88	75
<action envoyer signal>	6.18.2	91	91
<action envoyer tampon>	6.18.3	92	91
<action envoyer>	6.18.1	91	75
<action faire>	6.5.1	79	75
<action mettre en attente et choisir>	6.17	90	75
<action mettre en attente>	6.16	89	75
<action parenthésée>	6.1	75	75
<action recevoir et choisir>	6.19.1	92	75
<action recevoir signal et choisir>	6.19.2	93	92
<action recevoir tampon et choisir>	6.19.3	94	92
<action résulter>	6.8	86	75
<action revenir>	6.8	86	75
<action sortir>	6.6	83	75
<action vide>	6.11	87	75
<action>	6.1	75	75
<ancien préfixe>	12.2.3.3	158	158
<appel d'opération prédéfinie affecter>	6.20.4	98	96
<appel d'opération prédéfinie associer>	7.4.2	103	102
<appel d'opération prédéfinie attribut d'accès>	7.4.8	107	102
<appel d'opération prédéfinie attribut d'association>	7.4.4	104	102
<appel d'opération prédéfinie CHILL>	6.20	95	
<appel d'opération prédéfinie connecter>	7.4.6	105	102
<appel d'opération prédéfinie de durée>	9.4.1	124	124
<appel d'opération prédéfinie de temps absolu>	9.4.2	124	124
<appel d'opération prédéfinie de texte>	7.5.3	111	102
<appel d'opération prédéfinie de valeur temps>	9.4	124	96
<appel d'opération prédéfinie déconnecter>	7.4.7	107	102
<appel d'opération prédéfinie dissocier>	7.4.3	103	102
<appel d'opération prédéfinie écrire article>	7.4.9	108	102
<appel d'opération prédéfinie est associé>	7.4.2	103	102
<appel d'opération prédéfinie fixer texte>	7.5.8	118	102

non-terminal	défini dans la section	page	employé page(s)
<appel d'opération prédéfinie lire article>	7.4.9	108	102
<appel d'opération prédéfinie modification>	7.4.5	104	102
<appel d'opération prédéfinie obtenir texte>	7.5.8	118	102
<appel d'opération prédéfinie rendant locus CHILL>	6.20.2	95	95
<appel d'opération prédéfinie rendant locus d'e/s>	7.4.1	102	95
<appel d'opération prédéfinie rendant locus>	4.2.12	48	41
<appel d'opération prédéfinie rendant valeur CHILL>	6.20.3	95	95
<appel d'opération prédéfinie rendant valeur d'e/s>	7.4.1	102	96
<appel d'opération prédéfinie rendant valeur>	5.2.13	64	51
<appel d'opération prédéfinie simple CHILL>	6.20.1	96	95
<appel d'opération prédéfinie simple d'e/s>	7.4.1	102	95
<appel d'opération prédéfinie simple de temporisation>	9.4.3	125	95
<appel d'opération prédéfinie terminer>	6.20.4	98	95
<appel d'opération prédéfinie>	6.7	84	48,64
<appel de procédure rendant locus>	4.2.11	48	41
<appel de procédure rendant valeur>	5.2.12	64	51
<appel de procédure>	6.7	84	48,64,84
<argument de format>	7.5.3	111	111
<argument de locus>	7.5.3	111	111
<argument de mode>	6.20.3	96	96,98
<argument de texte>	7.5.3	111	111
<argument de valeur>	7.5.3	111	111
<argument longueur>	6.20.3	96	96
<argument pour upper lower>	6.20.3	96	96
<attribut de paramètre>	3.7	22	22
<attribut de résultat>	3.7	22	22
<bit final>	3.12.5	34	34
<bit initial>	3.12.5	34	34
<bloc début-fin>	10.3	130	75
<borne inférieure>	3.4.6	19	19
<borne supérieure>	3.4.6	19	19
<caractère>	–	0	9,54,55,113,114
<cas à choisir>	6.4	78	78
<chaîne de caractères>	2.4	9	9
<chaîne de commande de format>	7.5.4	113	113
<champ à choisir>	3.12.4	31	31
<champ de structure>	4.2.10	47	41
<champ fixe>	3.12.4	31	31
<champ récurrent>	3.12.4	31	31
<champ>	3.12.4	31	31
<chiffre hexadécimal>	5.2.4.2	53	53,56
<chiffre octal>	5.2.4.2	53	56,56
<chiffre>	2.2	8	8,53,113,114
<choix d'exceptions>	8.2	120	120
<choix de champs>	3.12.4	31	31
<choix de champs>	3.12.4	27	27
<citation>	5.2.4.7	55	55
<clause alors>	6.3	77	77
<clause d'e/s>	7.5.7	117	113
<clause d'édition>	7.5.6	116	113
<clause d'interdiction>	12.2.3.4	160	160
<clause de conversion>	7.5.5	114	113

non-terminal	défini dans la section	page	employé page(s)
<clause de directive>	2.6	10	
<clause de format>	7.5.4	113	113
<clause parenthésée>	7.5.4	113	113
<clause préfixe>	12.2.3.4	160	159,161
<clause renommer préfixe>	12.2.3.3	158	159,161
<clause sinon>	6.3	77	77
<code d'e/s>	7.5.7	117	117
<code d'édition>	7.5.6	116	116
<code de commande>	7.5.4	113	113
<code de conversion>	7.5.5	114	114
<commande avec>	6.5.4	83	83
<commande pour>	6.5.2	80	79
<commande tandis>	6.5.3	82	79
<commentaire fin de ligne>	2.4	9	9
<commentaire parenthésé>	2.4	9	9
<commentaire>	2.4	9	
<compteur de boucle>	6.5.2	80	80
<contenu de locus>	5.2.2	51	50
<contexte distant>	10.10.1	137	138
<contexte>	10.10.2	138	138
<conversion d'expression>	5.2.11	63	50
<conversion de locus>	4.2.13	49	41
<corps de contexte>	10.2	128	137,138
<corps de module de spec>	10.2	128	138
<corps de module>	10.2	128	134
<corps de procédure>	10.2	128	131
<corps de processus>	10.2	128	133
<corps de région de spec>	10.2	128	138
<corps de région>	10.2	128	135
<corps début-fin>	10.2	128	130
<déclaration de loc-identité>	4.1.3	40	39
<déclaration de locus>	4.1.2	39	39
<déclaration>	4.1.1	39	39
<définition de mode>	3.2.1	13	14
<définition de nom de champ>	2.7	10	10
<définition de procédure>	10.4	131	131
<définition de processus>	10.5	133	133
<définition de signal>	11.5	145	145
<définition de synonyme>	5.1	50	50
<définition>	2.7	10	10,18,75,80,94,131,133,134,135, 140,145
<descripteur dérepéré>	4.2.5	43	41
<directive d'implémentation>	–	0	10
<directive>	2.6	10	10
<élément d'ensemble avec numéros>	3.4.5	18	18
<élément d'ensemble>	3.4.5	18	18
<élément de chaîne>	4.2.6	44	41
<élément de début>	4.2.6	44	44,45,60
<élément de droite>	4.2.7	45	45,60
<élément de format>	7.5.4	113	113
<élément de gauche>	4.2.7	45	45,60
<élément de liste d'e/s>	7.5.3	111	111

non-terminal	défini dans la section	page	employé page(s)
<élément de rangée>	4.2.8	46	41
<élément inférieur>	4.2.9	46	46,62
<élément supérieur>	4.2.9	46	46,62
<énoncé d'action>	6.1	75	128
<énoncé d'octroi>	12.2.3.4	159	158
<énoncé de définition de neumode>	3.2.3	14	128,139
<énoncé de définition de procédure>	10.4	131	128
<énoncé de définition de processus>	10.5	133	128
<énoncé de définition de signal>	11.5	145	128
<énoncé de définition de synmode>	3.2.2	14	128,139
<énoncé de définition de synonyme>	5.1	50	128,139
<énoncé de saisie>	12.2.3.5	161	158
<énoncé de visibilité>	12.2.3.2	158	128
<énoncé déclaratif>	4.1.1	39	128
<énoncé définissant>	10.2	128	128
<énoncé informatif>	10.2	128	128
<énumération de locus>	6.5.2	80	80
<énumération de valeur>	6.5.2	80	80
<énumération ensembliste>	6.5.2	80	80
<énumération par intervalle>	6.5.2	80	80
<énumération par pas>	6.5.2	80	80
<étiquette de cas>	12.3	164	164
<événement à choisir>	6.17	90	90
<expression année>	9.4.2	124	124
<expression conditionnelle>	5.3.2	67	67
<expression démarrer>	5.2.14	65	51,88
<expression écrire>	7.4.9	108	108
<expression heure>	9.4.2	125	124
<expression indice>	7.4.6	105	105,108,111
<expression jour>	9.4.2	124	124
<expression minute>	9.4.2	125	124
<expression mois>	9.4.2	124	124
<expression parenthésée>	5.2.16	65	61
<expression positionnement>	7.4.6	105	105
<expression recevoir>	5.3.9	74	74
<expression seconde>	9.4.2	125	124
<expression usage>	7.4.6	105	105
<expression>	5.3.2	67	18,19,24,26,28,29,31,34,44,45, 46,55,56,63,65,66,67,73,75,77, 78,80,82,87,89,96,105,108,111, 118,124,125,140,164
<extension d'ensemble avec numéros>	3.4.5	18	18
<extension d'ensemble sans numéros>	3.4.5	18	18
<extension d'ensemble>	3.4.5	18	18
<facteur de répétition>	7.5.4	113	113
<fenêtre d'octroi>	12.2.3.4	160	159
<fenêtre de saisie>	12.2.3.5	161	161
<filet de temporisation>	3.11.1	122	122,123
<filet>	8.2	120	39,40,75,131,133,134,135
<fin de ligne>	–	0	9
<généralité>	10.4	131	131
<implantation d'élément>	3.12.5	34	29

non-terminal	défini dans la section	page	employé page(s)
<implantation de champ>	3.12.5	34	31
<indicateur de fragment>	10.10.1	137	136,137
<indice supérieur>	3.12.3	29	29
<indifférent>	12.3	164	164
<initialisation domaniale>	4.1.2	39	39
<initialisation viagère>	4.1.2	39	39
<initialisation>	4.1.2	39	39
<intervalle littéral>	3.4.6	19	19,25,164
<intervalle>	5.2.5	56	56
<itération>	6.5.2	80	80
<largeur de clause>	7.5.5	114	144,116
<lettre>	2.2	8	8
<liste d'arguments d'e/s de texte>	7.5.3	111	111
<liste d'attributs de procédure>	10.4	131	131,140
<liste d'e/s>	7.5.3	111	111
<liste d'énoncés d'action>	10.2	128	77,78, 79,90,93,94,120,122,123,128
<liste d'énoncés informatifs>	10.2	128	128
<liste d'étiquettes de cas>	12.3	164	56,164
<liste d'événements>	6.17	90	90
<liste d'exceptions>	3.7	22	22,120,131,140
<liste d'expressions littérales>	3.12.4	31	31
<liste d'expressions>	4.2.8	46	46,61,96
<liste d'intervalles>	6.4	78	78
<liste de contextes>	10.10.2	138	134,135,138
<liste de définitions de noms de champ>	2.7	10	31
<liste de définitions>	2.7	10	13,39,40,50,93,131,139,140
<liste de marqueurs>	3.12.4	31	31
<liste de noms d'interdiction>	12.2.3.4	160	160
<liste de noms de champ>	5.2.5	57	57
<liste de paramètres d'opération prédéfinie>	6.7	84	84
<liste de paramètres effectifs>	6.7	84	65,84
<liste de paramètres formels>	10.4	131	131,133
<liste de paramètres pour associer>	7.4.2	103	103
<liste de paramètres pour modifier>	7.4.5	104	104
<liste de paramètres>	3.7	22	22
<liste de sélecteurs de cas>	6.4	78	67,78
<littéral binaire d'entier>	5.2.4.2	53	53
<littéral binaire de chaîne de bits>	5.2.4.8	56	56
<littéral d'ensemble>	5.2.4.5	54	52
<littéral d'entier>	5.2.4.2	53	52
<littéral de booléen>	5.2.4.3	53	52
<littéral de caractère>	5.2.4.4	54	52
<littéral de chaîne de bits>	5.2.4.8	56	52
<littéral de chaîne de caractères>	5.2.4.7	55	52,137

non-terminal	défini dans la section	page	employé page(s)
<littéral de vide>	5.2.4.6	54	52
<littéral décimal d'entier>	5.2.4.2	53	53
<littéral hexadécimal d'entier>	5.2.4.2	53	53
<littéral hexadécimal de chaîne de bits>	5.2.4.8	56	56
<littéral octal d'entier>	5.2.4.2	53	53
<littéral octal de chaîne de bits>	5.2.4.8	56	56
<littéral>	5.2.4.1	52	50
<locus année>	9.4.3	125	125
<locus de lecture>	7.4.9	108	108
<locus heure>	9.4.3	125	125
<locus jour>	9.4.3	125	125
<locus minute>	9.4.3	125	125
<locus mois>	9.4.3	125	125
<locus repéré>	5.3.9	74	74
<locus seconde>	9.4.3	125	125
<locus transfert>	7.4.6	105	105,107
<locus>	4.2.1	41	40,44,45,46,47,49,51,74,75,80, 83,84,86,88,89,90,92,93,94,96, 103,104,105,108,111,118,125
<longueur d'événement>	3.9.2	24	24
<longueur de chaîne>	3.12.2	28	28
<longueur de tampon>	3.9.3	24	24
<longueur de texte>	3.10.4	26	26
<longueur>	3.12.5	34	34
<mode accès>	3.10.3	25	25
<mode association>	3.10.2	25	25
<mode booléen>	3.4.3	17	16
<mode caractère>	3.4.4	17	16
<mode chaîne paramétré>	3.12.2	28	28
<mode chaîne>	3.12.2	28	28
<mode composé>	3.12.1	28	15
<mode d'entrée-sortie>	3.10.1	25	15
<mode d'événement>	3.9.2	24	23
<mode d'indice>	3.10.3	25	25,26,29
<mode de synchronisation>	3.9.1	23	15
<mode définissant>	3.2.1	13	14
<mode des éléments de tampon>	3.9.3	24	24
<mode des éléments>	3.12.3	29	29
<mode descripteur>	3.6.4	21	20
<mode discret>	3.4.1	16	15
<mode durée>	3.11.2	27	27
<mode enregistrement>	3.10.3	25	25
<mode ensemble>	3.4.5	18	16
<mode ensembliste>	3.5	20	15
<mode entier>	3.4.2	16	16
<mode exemplaire>	3.8	23	15

non-terminal	défini dans la section	page	employé page(s)
<*mode intervalle*>	3.4.6	19	16
<*mode non composé*>	3.3	15	15
<*mode primitif*>	3.5	20	20
<*mode procédure*>	3.7	22	15
<*mode rangée paramétré*>	3.12.3	29	29
<*mode rangée*>	3.12.3	29	28
<*mode repère libre*>	3.6.3	21	20
<*mode repère lié*>	3.6.2	21	20
<*mode repéré*>	3.6.2	21	21
<*mode repère*>	3.6.1	20	15
<*mode structure paramétré*>	3.12.4	31	31
<*mode structure*>	3.12.4	31	28
<*mode tampon*>	3.9.3	24	23
<*mode temporisation*>	3.11.1	27	15
<*mode temps absolu*>	3.11.3	27	27
<*mode texte*>	3.10.4	26	25
<*mode*>	3.3	15	13,20,21,22,24,25,29,31,39,40, 50,139,140,145
<*module de contexte*>	10.10.1	137	75
<*module de spec simple*>	10.10.2	138	138
<*module de spec*>	10.10.2	138	75,128,135
<*module*>	10.6	134	75,135
<*modulion distant*>	10.10.1	136	134,135
<*mot*>	3.12.5	34	34
<*multiplet de rangée avec indices*>	5.2.5	56	56
<*multiplet de rangée sans indices*>	5.2.5	56	56
<*multiplet de rangée*>	5.2.5	56	56
<*multiplet de structure avec noms de champ*>	5.2.5	57	56
<*multiplet de structure sans noms de champ*>	5.2.5	56	56
<*multiplet de structure*>	5.2.5	56	56
<*multiplet ensembliste*>	5.2.5	56	56
<*multiplet*>	5.2.5	56	50
<*nom d'accès*>	4.2.2	42	41
<*nom d'exception*>	2.7	10	22,88
<*nom de champ*>	2.7	10	31,47,57,63,160
<*nom de mode chaîne originel*>	3.12.2	28	28
<*nom de mode rangée originel*>	3.12.3	29	29
<*nom de mode structure variable originel*>	3.12.4	31	31
<*nom de repère de texte*>	2.7	10	137
<*nom de valeur*>	5.2.3	51	50
<*nom*>	2.7	10	16,17,18,19,20,21,22,23,24,25, 27,28,29,31,42,43,49,51,53,54, 56,63,65,66,78,80,83,84,87,91, 93,96,145,164
<*nouveau préfixe*>	12.2.3.3	158	158
<*objet composé*>	6.5.2	80	80
<*opérande-0*>	5.3.3	68	67,68
<*opérande-1*>	5.3.4	69	68,69
<*opérande-2*>	5.3.5	69	69
<*opérande-3*>	5.3.6	71	69,71

non-terminal	défini dans la section	page	employé page(s)
<opérande-4>	5.3.7	72	71,72
<opérande-5>	5.3.8	73	72
<opérande-6>	5.3.9	74	73
<opérateur affectant>	6.2	76	75
<opérateur arithmétique additif>	5.3.6	71	71,76
<opérateur arithmétique multiplicatif>	5.3.7	72	72,76
<opérateur binaire fermé>	6.2	76	76
<opérateur d'appartenance>	5.3.5	70	69
<opérateur d'inclusion ensembliste>	5.3.5	70	69
<opérateur de concaténation de chaîne>	5.3.6	71	71,76
<opérateur de différence ensembliste>	5.3.6	71	71,76
<opérateur de répétition de chaîne>	5.3.8	73	73
<opérateur nullaire>	5.2.15	61	51
<opérateur relationnel>	5.3.5	70	69
<opérateur unaire>	5.3.8	73	73
<opérateur-3>	5.3.5	69	69
<opérateur-4>	5.3.6	71	71
<paramètre d'opération prédéfinie>	6.7	84	84
<paramètre effectif>	6.7	84	84
<paramètre formel>	10.4	131	131
<paramètre pour associer>	7.4.2	103	103
<paramètre pour modifier>	7.4.5	104	104
<partie avec>	6.5.4	83	79
<partie de commande>	6.5.1	79	79
<pas>	3.12.5	34	34
<pos>	3.12.5	34	34
<postfixe d'octroi>	12.2.3.4	160	158,160
<postfixe de saisie>	12.2.3.5	161	158,161
<postfixe>	12.2.3.3	158	158
<pourcent>	7.5.4	113	113
<préfixe simple>	2.7	10	10
<préfixe>	2.7	10	10,158,160,161
<premier élément>	4.2.9	46	46,62
<priorité>	6.16	89	89,90,91,92
<programme>	10.8	135	
<qualificatif de conversion>	7.5.5	114	114
<quasi-déclaration de loc-identité>	10.10.3	139	139
<quasi-déclaration de locus>	10.10.3	139	139
<quasi-déclaration>	10.10.3	139	139
<quasi-définition de signal>	10.10.3	140	140
<quasi-définition de synonyme>	10.10.3	140	140
<quasi-énoncé de définition de procédure>	10.10.3	140	139
<quasi-énoncé de définition de processus>	10.10.3	140	139
<quasi-énoncé de définition de signal>	10.10.3	140	139
<quasi-énoncé de définition de synonyme>	10.10.3	140	139
<quasi-énoncé déclaratif>	10.10.3	139	139
<quasi-énoncé définissant>	10.10.3	139	139
<quasi-énoncé informatif>	10.10.3	139	128
<quasi-liste de paramètres formels>	10.10.3	140	140
<quasi-paramètre formel>	10.10.3	140	140
<région de spec simple>	10.10.2	138	138
<région de spec>	10.10.2	138	128,135

non-terminal	défini dans la section	page	employé page(s)
<région>	10.7	135	128,135
<repère libre dérepéré>	4.2.4	43	41
<repère lié dérepéré>	4.2.3	42	41
<représentation textuelle de nom préfixe>	2.7	10	10
<représentation textuelle de nom simple>	2.2	8	10,75,131,133,134,135,136,138,140
<représentation textuelle de nom>	2.7	10	10,160,161
<résultat>	6.8	86	86
<séquence de contrôle>	5.2.4.7	55	54,55
<signal à choisir>	6.19.2	93	93
<solution alors>	5.3.2	67	67
<solution cas de valeur>	5.3.2	67	67
<solution sinon>	5.3.2	67	67
<sous-expression>	5.3.2	67	67
<sous-opérande-0>	5.3.3	68	68
<sous-opérande-1>	5.3.4	69	69
<sous-opérande-2>	5.3.5	69	69
<sous-opérande-3>	5.3.6	71	71
<sous-opérande-4>	5.3.7	72	72
<spec de module>	10.10.2	138	138
<spec de paramètre>	3.7	22	22,131,140
<spec de région>	10.10.2	138	138
<spec de résultat>	3.7	22	22,131,140
<spec distante>	10.10.1	136	138
<spécification d'étiquettes de cas>	12.3	164	31,67,78
<spécification de format>	7.5.4	113	113
<symbole d'affectation>	6.2	76	39,40,75,76,80
<taille de pas>	3.12.5	34	34
<taille de tranche>	4.2.7	45	45,46,60,62
<tampon à choisir>	6.19.3	94	94
<texte de format>	7.5.4	113	113
<tranche de chaîne>	4.2.7	45	41
<tranche de rangée>	4.2.9	46	41
<type de chaîne>	3.12.2	28	28
<valeur champ de structure>	5.2.10	63	50
<valeur de pas>	6.5.2	80	80
<valeur élément de chaîne>	5.2.6	60	50
<valeur élément de rangée>	5.2.8	61	50
<valeur finale>	6.5.2	80	80
<valeur indéfinie>	5.3.1	66	66
<valeur initiale>	6.5.2	80	80
<valeur primitive>	5.2.1	50	42,43,60,61,62,63,74,83,84,91, 98,122,123,125
<valeur tranche de chaîne>	5.2.7	60	50
<valeur tranche de rangée>	5.2.9	62	50
<valeur>	5.3.1	66	39,50,56,57,75,84,86,91,92,98, 103,104,140
<vide>	6.11	87	87,128,137,139,158

APPENDICE H

Index

Les numéros de page en caractères gras renvoient aux définitions d'un élément; ceux en caractères normaux renvoient aux occurrences d'utilisation des éléments de l'index.

ABS 72, **96**, 97-98, 174

ABSTIME **124**, 125, 174

accès 2, 5, 12, 31,34, 39-40, 42, 83, 101, 118, 135-136, 142

ACCESS **25**, 27, 163, 173

actif 5, **142**, 143-145

action 1, 3, 5-6, 9, 75, 80, 87, 90, 112, 115, 120-122, 128, 131, 133, 142, 144-145, 170

action **75**, 127

action affirmer 4, **87**

action affirmer 75, **87**

action aller 3, **87**, 130

action aller 75, **87**

action appeler **84**, 132

action appeler 75, **84**

action arrêter 5, **88**, 142

action arrêter 75, **88**

action causer 3-4, **88**, 120

action causer 75, **88**, 120

action conditionnelle 3, **77**

action conditionnelle 75, **77**, 127, 129

action continuer 5, 24, **89**, 90, 145

action continuer 75, **88**

action d'affectation **76**, 143

action d'affectation **75**

action d'affectation multiple **75**

action d'affectation simple 57, **75**

action de cas 3, 33, 68, **78**, 164-165

action de cas 75, **78**, 127, 129, 165

action de temporisation 122

action de temporisation 75, **122**, 129

action de temporisation absolue 122, **123**, 127

action de temporisation cyclique 122-123

action de temporisation cyclique 122, **123**, 127

action de temporisation relative **122**, 127

action démarrer **88**

action démarrer 75, **88**

action envoyer 5, 24, **91**, 92, 143

action envoyer 57, 75, **91**

action envoyer signal 91, 93, 145

action envoyer signal **91**

action envoyer tampon 92, 94, 144-145

action envoyer tampon 91, **92**

action faire 3, **79**, 80-83, 130, 144

action faire 42, 52, 75, **79**, 129, 143

action mettre en attente 24, **89**, 144

action mettre en attente 75, **89**

action mettre en attente et choisir 24, **90**, 144

action mettre en attente et choisir 75, **90**, 127, 129

action parenthésée 3, 83-84, 121

action parenthésée **75**

action recevoir et choisir 3, 5, 24, **92**, 145

action recevoir et choisir 52, 75, **92**, 127

action recevoir signal et choisir **93**, 144

action recevoir signal et choisir 92, **93**, 129

action recevoir tampon et choisir **94**, 144-145

action recevoir tampon et choisir 92, **94**, 129

action résulter 3, **86**, 132, 143

action résulter 57, 75, **86**, 87, 132

action revenir **86**, 131

action revenir 57, 75, **86**

action sortir 3, **83**, 84

action sortir 75, **83**, 84

action vide **87**

action vide 75, **87**

activation 86, 136, **142**

AFTER **122**, 173

ALL 137, **160-161**, 162, 173

ALLOCATE 2, 4, 57, **98**, 99, 136, 174

ALLOCATEFAIL 99, 175

allocation de mémoire **136**

ancien préfixe **158**, 159-162

AND **69**, **76**, 173

ANDIF **69**, 173

apparié par la nouveauté **153**

appel d'opération prédéfinie 3-4, 48, 57, 84, 97, 99, 103, 107-114, 119, 125, 136, 169

appel d'opération prédéfinie **84**, 85, 95-96, 169

appel d'opération prédéfinie affecter 96, **98**

appel d'opération prédéfinie associer 102, **103**

appel d'opération prédéfinie attribut d'accès 102, **107**

appel d'opération prédéfinie attribut d'association 102, **104**

appel d'opération prédéfinie CHILL **84**, **95**

appel d'opération prédéfinie connecter 102, **105**

appel d'opération prédéfinie de durée 124

appel d'opération prédéfinie de durée **124**

appel d'opération prédéfinie de temps absolu 125

appel d'opération prédéfinie de temps absolu **124**
appel d'opération prédéfinie de texte 102, **111**
appel d'opération prédéfinie de valeur temps 96, **124**
appel d'opération prédéfinie déconnecter 102, **107**
appel d'opération prédéfinie dissocier 102, **103**
appel d'opération prédéfinie écrire article 102, **108**
appel d'opération prédéfinie est associé 102, **103**
appel d'opération prédéfinie fixer texte 102, **118**
appel d'opération prédéfinie lire article 102, **108**
appel d'opération prédéfinie modification 102, **104**
appel d'opération prédéfinie obtenir texte 102, **118**
appel d'opération prédéfinie par l'implémentation 84
appel d'opération prédéfinie rendant locus **48**
appel d'opération prédéfinie rendant locus 41, **48**, 49
appel d'opération prédéfinie rendant **locus** 84
appel d'opération prédéfinie rendant locus 48-49, 143, 168
appel d'opération prédéfinie rendant locus CHILL **95**
appel d'opération prédéfinie rendant locus d'e/s 95, **102**
appel d'opération prédéfinie rendant valeur **64**
appel d'opération prédéfinie rendant valeur 51, **64**
appel d'opération prédéfinie **rendant valeur** 84
appel d'opération prédéfinie rendant valeur 64, 144, 168
appel d'opération prédéfinie rendant valeur CHILL 95, **96**
appel d'opération prédéfinie rendant valeur d'e/s 96, **102**
appel d'opération prédéfinie simple CHILL **95**
appel d'opération prédéfinie simple d'e/s 95, **102**
appel d'opération prédéfinie simple de temporisation 95, **125**
appel d'opération prédéfinie terminer 95, **98**
appel de procédure 3, 5, **84**, 86, 130-132, 143
appel de procédure 57, **84**, 85, 143-144
appel de procédure rendant locus **48**, 132
appel de procédure rendant locus 41, **48**, 143
appel de procédure rendant locus 48, 85, 143, 168
appel de procédure rendant valeur **64**, 132
appel de procédure rendant valeur 51, **64**, 144
appel de procédure rendant valeur 64, 85, 168
argument de format **111**, 112
argument de locus **111**, 115-116
argument de mode 57, **96**, 97-99
argument de texte **111**, 112
argument de valeur **111**, 115-116
argument longueur **96**
argument pour upper lower **96**, 97
ARRAY **29**, 30, 35, 163, 173
ASSERT **87**, 173
ASSERTFAIL 87, 175
ASSOCIATE 4, 25, 100, **103**, 174
ASSOCIATEFAIL 103, 170, 175
ASSOCIATION 25, 103, 107, 163, 174
association 2, 4, **25**, 39-40, 100-110, 170
AT **123**, 173
attribut d'accès **102**
attribut d'association **101**
attribut de paramètre **23**, 132-134, 149, 151
attribut de paramètre **22**
attribut de résultat **23**, 132
attribut de résultat **22**
BEGIN **130**, 173
BIN **19**, 20, 163, 173
bit final **34**, 36
bit initial **34**, 36, 151
bloc 1, 52, 82, 121, **127**, 128, 130, 134-136, 142, 156-157
bloc début-fin 3-4, **130**
bloc début-fin 75, 127, 129, **130**
BODY **134-135**, 173
BOOL 17, 44, 54, 60, 72, 103-104, 107, 154, 163, 174
BOOLS **28**, 29, 56, 71, 73, 163, 173
borne inférieure 30, 47
borne inférieure **17-19**, **29-30**, 37, 46-47, 61-62, 81, 97, 106, 108, 149, 151, 169
borne inférieure **19**, 20, 30, 37
borne supérieure **17-19**, 22, **29-30**, 37, 44, 46-47, 61-62, 81, 97, 109, 149, 151, 151, 169
borne supérieure **19**, 20, 30
BUFFER **24**, 163, 173
BY **80**, 173
caractère 2, 7-11, 17, 28, 54-55, 71, 110, 113-118
caractère 8-9, 54, 114, 168
caractère non réservé 55, 168
caractère non réservé 55, 168
caractère non-pourcent 113
caractère souligné **8**, 53, 56
CARD **96**, 97-98, 174
cas à choisir **78**
cas à choisir **78**, 127, 165
CASE **31**, **67**, 68, **78**, **90**, **93-93**, 173
catégorie sémantique **7**, 166
CAUSE **88**, 173
chaîne de bits 28, 68-69
chaîne de caractères 28, 71, 111, 114, 116
chaîne de caractères **9**
chaîne de commande de format 11-112, **113**
chaîne fixe 116
chaîne **variable** 109, 116

chaîne vide 26, 40, 45, 60, 73
champ 11, **31**, 32-36, 47-48, 57, 59, 63, 66, 83, 146, 160, 163
champ **31**, 149-150, 152
champ à choisir 32, 59
champ à choisir **31**, 32-33, 36, 59, 150, 152, 165
champ de structure 34-36, **47**, 79
champ de structure 41, **47**, 48, 63, 136, 143, 164
champ fixe 31-32
champ fixe **31**, 32-33, 150, 152
champ **marqueur** 16, **32**, 33, 39, 48, 58-59, 63, 76, 146, 165
champ récurrent **32**, 42, 52, 76, 164
champ récurrent **31**, 32-33, 150, 152
champ **récurrent** 33, 42-44, 52, 170
changer-le-signe **73**
CHAR 17-18, 44, 54, 60, 72, 153, 163, 174
CHARS 27, **28**, 29, 55, 71, 73, 163, 173
chemin **14**, 148
chiffre **8**, 8,53, 113-116
chiffre hexadécimal **53**, 56
chiffre octal **53**, 56
chiffres 115, 116
CHILL 1-10, 12-13, 17, 23, 25-26, 37, 49, 55, 63, 66, 75, 85, 95, 100-102, 108-110, 113-115, 122, 135-137, 140, 142-144, 167, 169
choix d'exceptions 130
choix d'exceptions **120**, 127, 129
choix de champs 164
choix de champs **31**, 32-33, 36, 59, 150, 152, 165
choix de champs **sans marqueurs** 33
choix de champs **sans marqueurs** **32**, 33
citation 55, 168
citation **55**
classe 2-3, 5, 7, **12**, 13, 19-20, 26, 30, 33-34, 40, 46-47, 50-74, 76, 78, 82-86, 91-94, 97-99, 103-104, 106-107, 109, 112, 115-116, 119, 124-125, 140, 143, 147-149, 152-153, 155-156, 165, 167-169
classe dynamique **12**, 51, 68-71, 76, 81, 132
classe **nulle** **12**, 55, 143, 155
classe par dérivation **12**, 53-56, 65, 70-71, 73, 97, 103-104, 106-107, 119, 124-125
classe par repère **12**, 107, 143
classe par valeur **12**, 33, 60, 71
classe résultante 12, 19, 58, 68-69, **71-73**, 82, 97, **147**, 165
classe statique 97
classe **toute** **12**, 33, **66**, 140, 143, 147, 155, 165
classes constantes **12**
clause alors **77**, 127
clause d'e/s 112-113, **117**
clause d'édition 112-113, **116**, 119
clause d'interdiction **160**, 161, 164
clause de conversion 112-114, **114**
clause de directive **10**
clause de directive **10**
clause de format 112, **113**
clause parenthésée 112, **113**
clause préfixe 159, **160**, 161-162
clause renommer préfixe **158**, 159-162
clause sinon **77**
clauses renommer préfixe 158
code d'e/s 112, **117**
code d'édition 112, **116**, 117, 119
code de commande 112, **113**
code de conversion 115
code de conversion 112, **114**, 115-116
cohérence 33, **36**, 68
cohérent **165**
combinaison de caractères spéciaux 8-9
commande avec **83**
commande pour 79, **80**, 82
commande pour 79, **80**
commande tandis 79
commande tandis 79, **82**, 127
commandes de mise en page **9**, 11, 113
commentaire 9, 11
commentaire **9**
commentaire fin de ligne **9**
commentaire parenthésé **9**
compactage **34**, 35
compatible 13, 20, 30, 34, 40, 46-47, 50, 58-59, 61-62, 67-70, 72-73, 76-78, 82, 85-86, 91-92, 98-99, 106, 109, 112, 147, 149, **155**, 165, 167-168
compatible en lecture 13, 41, 43, 85-86, 119, **153**, 154-155
compatible en lecture dynamique 13, 41, 85-86, **154**, 155
complémentaire **73**
complet 58, 78, **165**
compteur de boucle **80**, 81
compteur de boucle 42, 52, **80**, 81-82, 127
concaténation 9, 11, 28, 71
condition dynamique 4, 6-7, 64, 76, 113, 120, 169
condition statique 7, 64-65, 140, 144, 148
conditions d'accès au champ récurrent 42-44, **48**, 52, **63**
conditions d'affectation 40, 59, 65, 68, **76**, 85-87, 91-92, 99, 109
conditions de sélection de cas 33, 58, 68, 78
conditions dynamiques 7

conditions statiques **7**
conjonction 69
CONNECT 4, 100, **105**, 106-107, 174
connecté 4, 40, 100-103, 105-110, 117
CONNECTFAIL 106, 170, 175
constant 3, 50-59, 63, 66-74, 97, 115, 136, 140, 168, 170
contenu de locus **51**
contenu de locus 50, **51**, 144
CONTEXT **137-138**, 173
contexte 5, 85, 169
contexte 127, 129-130, **138**, 139-141, 161-162
contexte distant **137**, 138
CONTINUE **88**, 173
conversion d'expression **63**
conversion d'expression 50-51, **63**, 144, 170
conversion de locus **49**
conversion de locus 41, **49**, 63, 136, 143, 170
corps de contexte **128**, 137-138
corps de module 134, 138-139, 141
corps de module **128**, 134, 157
corps de module de spec **128**, 138
corps de procédure **128**, 131
corps de processus 142
corps de processus **128**, 133
corps de région 135, 138-139, 141
corps de région **128**, 135, 157
corps de région de spec **128**, 138
corps début-fin **128**, 130
correspondre 140-141
CREATE **104**, 174
CREATEFAIL 104, 170, 175
création d'un processus **142**
créé 2, 11, 23, 25, 27, 39-40, 65, 81, 98-101, 103, 108, 110-111, **127**, 128, 130, 132, 135-136, **142**, 155
critique 130, 132-133, 141, **142**, 143-144
CYCLE **123**, 173
DAYS **124**, 174
DCL **39**, 81, 132, **139**, 173
débordement 114-115
déclaration 1, 32, 39, 128, 130, 134, 136, 143, 161
déclaration **39**, 127
déclaration de loc-identité 2, **40**, 81, 128-129, 132, 136
déclaration de loc-identité 39, **40**, 41-42
déclaration de locus 2, 4, **39**, 132, 136
déclaration de locus **39**, 40, 42, 57
définition 5, 83
définition **10-11**, 13-16, 18, 39-40, 50, 75, 80-81, 84, 93-94, 127-128, 130-135, 137, 140-141, 145, 155-157, 161-164, 167
définition de mode 2, **13**, 14-15, 50
définition de mode **13**, 14-16, 127
définition de nom de champ 83
définition de nom de champ **10-11**, 31, 83, 164
définition de procédure 86, 121, **131**, 133, 136
définition de procédure 52, 127, 129, **121**, 132-133
définition de processus 5, 65, 84, 86-87, 121, **133**, 136, 141-142, 169
définition de processus 127, 129, **133**, 134
définition de signal 57, **145**
définition de signal 127, **145**
définition de synonyme 13, **50**
définition de synonyme 13, **50**, 57, 127
définition **impliquée** 157, **162**, 163
définition **réelle** 130, 141, 156-157
définitions de modes récursives 14
définitions récursives **13**, 14, 50
DELAY **89-90**, **122**, 123, 173
DELAYFAIL 89-90, 175
DELETE **104**, 105, 174
DELETEFAIL 105, 170, 175
dérepérant - dérepérage 2, 21
descripteur 2, 20, 22, 43
descripteur dérepéré 43
descripteur dérepéré 41, **43**, 44, 143
description sémantique 7-8
description syntaxique 7, 9, 166
différence 71
directement fortement visible 156, **157**
directement lié 156, **157**, 159
directive 10
directive **10**
directive d'implémentation **10**
directive d'implémentation 10, 169
DISCONNECT 100, **107**, 174
discret 51
disjonction exclusive **68**
disjonction inclusive **68**
DISSOCIATE 25, 100, **103**, 174
DO **79**, 86, 173
domaine 39-40, 79, 85, 89-90, 92-94, 121-123, **127**, 128-130, 135-136, 138, 141, 153, 156-164, 169
domaine **de destination** **158**, 159
domaine **originel** **158**, 159
domaine **réel** 130, 138-139, 141
DOWN **80**, 81, 173
DURATION 27, 124, 163, 174
durée de vie 1, 4, 39-40, 43-44, 48-49, 74, 81, 86, 89-92, 95, 98-99, 108, 127, **128**, 130, 132, 134-135, **136**

DYNAMIC **22**, 23, **25-26**, 27, **40**, 41, 48, 57, 85-86, 132, **139**, 173
écrivable 4, **101**, 104, 106
égalité **70**, 140
élément 2, 7, 28, 30, 34-36, 44, 46, 55-57, 60-61, 66, 68-69, 73, 81, 96-97, 101, 111-112, 116
élément d'ensemble 156
élément d'ensemble **18**
élément d'ensemble avec numéros **18**
élément de chaîne 28, **44**, 114
élément de chaîne 41, **44**, 60, 136, 143
élément de début **44**, 45, 60-61, 136
élément de droite **45**, 60-61, 136
élément de format 112, **113**
élément de gauche **45**, 60-61, 136
élément de liste d'e/s **111**, 112, 116
élément de rangée 34-35, **46**, 164
élément de rangée 41, **46**, 61, 136, 143
élément inférieur **46**, 47, 62, 136
élément lexical **8**, 9
élément supérieur **46**, 47, 62, 136
ELSE **31**, 32, 36, 57, 59, **67**, **77-79**, **93-94**, **120**, 121, 127, 129, 150, 152, **164**, 165, 173
ELSIF **67**, **77**, 173
EMPTY 43-44, 85, 91, 98-99, 107, 175
END **120**, **122-123**, **130-131**, **133-135**, 137, **138**, 139, **140**, 173
engendré 4, 131
englobant du plus près 83, 86, 136
englobé 5, 52, 86, 99, 127, **129**, 130, 134-136, 141-142
englober immédiatement 129
énoncé d'action 1, **75**, 87, 120, 134, 142
énoncé d'action **75**, 122-123, 128
énoncé d'octroi 138, **160**
énoncé d'octroi 158, **159**, 160-161, 164
énoncé de définition de neumode 6, 13, **15**
énoncé de définition de neumode **14**, 15-16, 128, 139
énoncé de définition de procédure 128, **131**, 132
énoncé de définition de processus 128, **133**, 134
énoncé de définition de signal 128, **145**
énoncé de définition de synmode **14**
énoncé de définition de synmode **14**, 128, 139
énoncé de définition de synonyme 3, **50**
énoncé de définition de synonyme **50**, 52, 128, 139-140
énoncé de saisie **161**
énoncé de saisie 158-159, **161**, 162
énoncé de visibilité 128, **158**, 159
énoncé déclaratif 2, **39**, 120
énoncé déclaratif **39**, 128
énoncé définissant **128**
énoncé informatif 1, 3, 120-121, 129
énoncé informatif **128**
énoncés de définition de procédure 22
énoncés de définition de signal 5
énoncés de visibilité 4-5, 139, 156, **158**
énoncés définissants 1
enregistrement de texte 26, 110-114, 116-119
ensemble de caractères 8-10, 17, 55, **171**
entamé 4, 39-40, 77-83, 90, 93-94, 120, 122-123, 128-129, **130**, 132, 142
entamer 142
énumération de locus 81
énumération de locus 42, **80**
énumération de valeur 52, **80**, 82
énumération ensembliste 80-81
énumération ensembliste **80**
énumération par intervalle 80-81
énumération par intervalle **80**
énumération par pas 80-81
énumération par pas **80**
EOLN 118-119, 174
équivalent 13, 76, 109, **148-149**, 150-155
équivalent dynamique 13, **154**, 155
ESAC **31**, **67**, **78**, **90**, **93-93**, 173
espace **9**
état libre 3, **100**
état traitement de fichiers 4, **100**, 101
état transfert de données 4, 100, **101**
étiquette 109
étiquette de cas 58, 165
étiquette de cas 78, **164**, 165
événement à choisir **90**, 127
EVENT **24**, 163, 173
EVER **80**, 173
exception 1, 3-6, 11, 41-48, 51-52, 59-66, 68-70, 72-79, 81-82, 85-93, 95, 98-99, 102-110, 112-113, 116-117, **120**, 121, 123-125, 130, 132, 148-150, 154-155, 169-170
EXCEPTIONS **22**, **131**, 133, **140**, 173
exécution concurrente 5, 133, 135, 142
existant 4, **101**, 104-106
EXISTING **104**, 174
EXIT **83**, 173
EXPIRED **125**, 126, 174
expression 23, 25, 32, 34, 38, 45-47, **51**, 57, 60, 62-63, 65-66, 73, 77-78, 81, 98, 101-102, 105, 110, 130, 144-145, 147, 164, 170
expression 7, 46-47, 56-59, 61-66, **67**, 75, 77, 80, 96, 98-99, 105, 108, 136, 144, 167-168

expression année **124**
expression booléenne 82
expression booléenne 7, 67, 77, 82, 87, 167
expression chaîne 97
expression *chaîne de caractères* 111, 167
expression conditionnelle 164-165
expression conditionnelle **67**, 68, 143-144, 165
expression démarrer 3, 5, **65**, **88**, 130, 142
expression démarrer 51, 57, **65**, **88**, 170
expression discrète 78, 97
expression écrire **108**, 109
expression **ensembliste** 81
expression *ensembliste* 80, 82, 96-97, 168
expression entière 37, 44-45, 61, 80, 96, 98-99, 118-119, 124-125, 168
expression heure 124, **125**
expression indice **105**, 106-109, 111-112, 118
expression jour **124**
expression littérale 140
expression littérale discrète 19, 29, 31, 58-59, 78, 164-165, 168
expression littérale entière 18-20, 24, 26, 28, 34-35, 55, 73, 89-92, 168
expression minute 124, **125**
expression mois **124**
expression parenthésée 46, 65
expression parenthésée 51, **65**, 66
expression positionnement **105**, 106
expression rangée 80, 82, 96-97, 167
expression recevoir 24, **74**, 144-145
expression recevoir **74**, 144
expression seconde 57
expression seconde 124, **125**
expression usage **105**, 106-107
expressions chaîne 80-82, 96-97, 111, 168
expressions discrètes 37, 78, 80, 96-98, 111, 168
extension d'ensemble **18**, 19
extension d'ensemble avec numéros **18**
extension d'ensemble sans numéros **18**
extrarégional 41, 59, 68, 99, **143**, 144
facteur de répétition 112, **113**
faible discordance 156-157
faiblement visible 156-157, 162
faisabilité **36**
FALSE 17, 53, 69-70, 87, 102-108, 115, 174, 203
fenêtre d'octroi 159, **160**
fenêtre de saisie **161-162**
FI **67**, **77**, 173
fichier 4, 26, **100**, 101-102, 104-110, 117, 170
filet 1, 4-6, 11, 75, **120**, 131, 129, 129, 131, 142, 169
filet 39-40, 75, 84, 86-88, **120**, 127, 129, 131, 133-135
filet de temporisation **122**, 123, 127, 129
filet défini par l'implémentation 121, **169**
fin de ligne **9**
FIRST 105-106, 174
FOR **80**, **137-138**, 173
FORBID **160**, 173
format fixe 114-115
format libre 114-115
forme Backus-Naur **7**
fort 3, **12**, 43-44, 60, 71, 78, 82-83, 97, 164
fortement visible **156**, 157, 159, 161-163
fragment 5, 9, 11, 136-137
fragment distant **136**, 137
GENERAL **131**, 132-133, 173
général - généralités 22, 32-33, 85, **132**, 133, 143, 167
généralité 85, 167
généralité 85, **132**, 141, 169
généralité **131**, 132
GETASSOCIATION **107**, 174
GETSTACK 2, 4, 57, **98**, 136, **174**
GETTEXTACCESS 118-119, 174
GETTEXTINDEX 118-119, 174
GETTEXTRECORD 118-119, 174
GETUSAGE **107**, 108, 174
GOTO **87**, 173
GRANT 158, **159**, 173
groupe 7, **127**, 129-130, 139, 141, 161, 164
hors du fichier **102**, 106-109
HOURS **124**, 174
identification 5, 10, 128, 155, **156**, 157
identification de filet **120**
IF 9, **67**, **77**, 173
immédiatement englobant 121, 127, **129**, 136, 139, 157-162
immédiatement englobé 121, **129**, 140, 141, 157, 159, 161, 164
implantation 30, 32, 34-35, 113
implantation d'élément 36, 82, 151
implantation d'élément **30**, 46-47, 149, 151-152, 169
implantation d'élément 29-30, **34**
implantation de champ 32-33, **36**, 83, 150
implantation de champ **32**, 48, 150, 152
implantation de champ 31-32, **34**, 35
implanté 30, **32**, 36
impliqué 156-157, 162-163
IN **22**, **70**, **80**, **85**, **93-93**, **122-123**, 127, 131-132, 173
in-situ **132**

INDEXABLE **104**, 174
indexable 4, **101**, 104-106
indexer 2
indicateur de fragment 136, **137**
indice **courant** 101, **106**, 108
indice **de base** 4, 101, **106**, 108
indice **de transfert** 101-102, 107, **108**, 109
indice effectif 110-112, 114, 116-118
indice supérieur **29**, 30, 47, 62
indifférent 150, 152, **164**, 165
indiqué explicitement 58, 66, 165
indiqué implicitement 165
indirectement fortement visible 156, **157**
inégalité **70**
inférieur à 36, 45, 60, 65, 70, 76, 112, 116-117, 119
inférieur ou égal à 20, 29-30, 36, 70
INIT **39**, 173
initialisation 39, 128
initialisation **39**, 40, 57
initialisation domaniale 128-129, 142-143
initialisation domaniale **39**, 40
initialisation viagère 128
initialisations viagères **39**
INLINE **131**, 132-133, 173
INOUT **22**, 85, 131-132, 134, 173
INSTANCE 23, 65, 163, 174
INT 13, 16, 19, 30, 53, 97-98, 110, 119, 131, 154, 163, 169, 174
intersection 69
intervalle 1-2, 17, 19-20, 30, 55, 57, 66, 78, 116, 124, 169
intervalle **56**
intervalle littéral **19**, 25-26, 78, 164-165
intrarégional 3, 41, 59, 68, 85, 91-92, 99, 133, **143**, 144, 161
INTTIME **125**, 174
invisible 58, **156**, **164**
ISASSOCIATED **103**, 174
itération 3
itération **80**
justification 114-115
l-équivalent 13, **148**, 149, **150**, 153
largeur 112, 114-117
largeur de clause 112, **114**, 115-116, 119
largeur de clause **variable** 111-112, 116
LAST 105-106, 174
LENGTH 28, **96**, 97, 174
lettre 8, 52, 115
lettre **8**
liaison **156**
liaison directe 156
libéré 121, **142**, 143-144
libre 142
lié 138, **156**, 157, 159
lié 11, 138, 141, 152-153, **157**, 159, 161-162, 164, 167
lié par la nouveauté 13, 15, 141, **148**, 152, **153**, 164
limitable 13, **154**, 155
lisible 4, **101**, 104, 106
liste d'arguments d'e/s de texte **111**
liste d'attributs de procédure **131**, 140
liste d'e/s **111**, 112, 116
liste d'énoncés d'action 77-82, 120-121, 123, 130, 164
liste d'énoncés d'action 77-79, 90, 93-94, 120, 122-123, 127, **128**, 129
liste d'énoncés informatifs **128**
liste d'étiquettes de cas 57, 78, 164-165
liste d'étiquettes de cas 56, 58, 78, 150, 152, **164**, 165
liste d'événements **90**
liste d'exceptions 121
liste d'exceptions **22**, 23, 120-121, 131-133, 140
liste d'expressions 37-38, **46**, 61, 96, 98-99
liste d'expressions littérales **31**, 33-34
liste d'intervalles 165
liste d'intervalles **78**
liste de classes **33**, 34, 98, 147, **165**
liste de contextes 127, 134-135, 137, **138**
liste de définitions **10**, 13, 39-40, 50, 93, 127, 131, **133**, 139-140
liste de définitions de noms de champ **10**, 31-32
liste de marqueurs **31**, 32-33, 150, 152
liste de noms d'interdiction 164
liste de noms d'interdiction **160**, 161, 164
liste de noms de champ 57
liste de noms de champ **57**, 59, 161
liste de paramètres 125
liste de paramètres **22**, 23
liste de paramètres d'opération prédéfinie **84**
liste de paramètres effectifs 84
liste de paramètres effectifs 65, **84**
liste de paramètres formels 65, 127, **131** 132-134, 141
liste de paramètres pour associer **103**
liste de paramètres pour modifier **104**, 105
liste de sélecteurs de cas **78**
liste de sélecteurs de cas 67, **78**
liste de valeurs 5, **33**, 38, 44, 48, 55-57, 59, 63, 93, 145, 154, 165
liste résultante des classes 33, 78, **165**
listes résultantes des classes 33
littéral 8, 17, 19, 32, **52**, 73
littéral 3, 34, 45, 47, 50, **51**, 52-54, 60, 62, 66-74, 97, 140, 168, 170

littéral 50-51, **52**
littéral binaire d'entier **53**
littéral binaire de chaîne de bits **56**
littéral d'ensemble **54**, 116
littéral d'ensemble 52, **54**
littéral d'entier **53**
littéral d'entier 52, **53**
littéral de booléen **54**
littéral de booléen 52, **53**, 54
littéral de caractère 18, 54
littéral de caractère 52, **54**
littéral de chaîne de bits **56**
littéral de chaîne de bits 52, **56**, 73
littéral de chaîne de caractères 9, **55**
littéral de chaîne de caractères 52, **55**, 73, 137
littéral de vide **55**
littéral de vide 52, **54**, 55
littéral décimal d'entier **53**
littéral discret **52**
littéral hexadécimal d'entier **53**
littéral hexadécimal de chaîne de bits 56
littéral octal d'entier **53**
littéral octal de chaîne de bits **56**
LOC **22**, 23, **40**, 42, 81, 85-87, 131-134, **139**, 173
locus 1-5, 12-13, 15, 20-22, 24-26, 31, 35-36, 39-40, **41**, 42-44, 46-49, 51, 55, 74, 76-77, 80-81, 83-86, 95, 97-106, 108-112, 116, 118-119, 125-128, 130-132, 134, 136, 142-143, 153-154, 160, 169-170
locus 40, **41**, 48, 51, 57, 74-77, 80, 83-86, 96-97, 103-104, 132, 136, 143-144, 161, 167
locus à mode dynamique 3, 76
locus accès 100-103, 105-108
locus **accès** 101
locus accès 105-106, 108-119, 118-119, 167
locus année **125**
locus association 100-103, 107
locus association 103-107, 167
locus chaîne 22, 44-45, 81
locus chaîne 41-44-45, 60, 80-82, 96-97, 111-112, 136, 143, 167
locus chaîne de caractères 111, 118-119, 167
locus de lecture **108**, 109
locus de mode statique 49, 63, 108, 136, 143, 167
locus discret 96-97, 111, 167
locus discrets 97
locus entier 125
locus événement 24, 89-90
locus événement 88-90, 167
locus exemplaire 90, 93-94, 167
locus heure **125**
locus **indéfini** **40**, 42, 48-49, 86, 132
locus jour **125**
locus minute **125**
locus mois **125**
locus rangée 22, 30, 46-47, 81
locus rangée 46-47, 61-62, 80-82, 96-97, 136, 143, 167
locus repéré 43-44, **74**, 99, 108
locus repéré **74**, 144
locus seconde **125**
locus structure 22, 31-32, 42, 44, 47, 83
locus structure 47-48, 63, 83, 136, 143, 164, 167
locus tampon 24, 74, 92, 94
locus tampon 57, 74, 92, 94-95, 167
locus texte 110
locus **texte** 110
locus texte 102, 105-107, 111-112, 117-119, 167
locus transfert **105**, 106-107
longueur **34**, 36, 97, 151
longueur d'événement **24**, 89-90, 149, 151
longueur d'événement **24**
longueur de chaîne 22, 28, **29**, 37, 44, **55-56**, 71, 73, 76, 98, 109, 111, 114, 116, 118, 150, 152, 154
longueur de chaîne **28**, 29
longueur de tampon **24**, 92, 149, 151
longueur de tampon **24**, 25
longueur de texte 26-27, 110-112, 117-119, 149, 151
longueur de texte **26**, 27
longueur effective 28, 44-45, 60-61, 68-69, 76, 81, 97, 114, 116-118
LONG-INT **17**
LOWER **96**, 97-98, 154, 174
majuscules **8**, 9
manière de placer le chariot 117
MAX 96, 97-98, 174
mémoire 32, 65, 77-79, 85, 90, 93, 95, 98-99, 120-121, 130, 148
métalangage 2, **7**
MILLISECS **124**, 174
MIN **96**, 97-98, 174
minuscules **8**, 9, 115
MINUTES **124**, 174
mis en attente 5, 92, 142
mise en attente 5, 24, 39, 74, 89-95, 122, **142**, 143-145
mise en attente d'un processus **144**
MOD **72**, 73, 173
mode 2-3, 5, 12-14, **15**, 16, 22-23, 26-27, 29-37, 39-49, 51-52, 57-65, 67-70, 72, 74, 76, 78, 81-87, 89-94, 97-99, 103-106, 108-110, 112, 115-116, 119, 126, 132-134, 140-141, 143, 145-151, 154-155, 160-165

mode 12-13, **15**, 16, 21-25, 29, 31-33, 39-41, 50, 57, 81, 132, 139-140, 145, 162, 168
mode accès 4, **26**, 102, 147, 149, 151, 153, 166-167
mode **accès** **25**
mode accès 27, 106, 110, 112, 119, 149, 151
mode association 4, **25**, 101, 147, 149, 151, 166-167
mode association **25**
mode booléen **17**, 148, 151, 166-167
mode booléen 16, **17**
mode caractère **17**, 18, 148, 151, 166
mode caractère 16, **17**
mode chaîne 28-29, 37, 44, 70, 82, 109, 146, 147, 149, 142, 154, 166-168
mode chaîne **28**, 169
mode chaîne 21-22, 168
mode chaîne **de bits** **29**, 44, 60, 149, 152
mode chaîne **de caractères** **29**, 44, 60, 149, 152, 167
mode chaîne dynamique **37**
mode chaîne fixe **28**, 29, 45, 60, 76, 81, 147, 150
mode chaîne **originel** 16, **29**
mode chaîne paramétré 37
mode chaîne **paramétré** **28**, 29
mode chaîne paramétré 15-16, **29**, 45, 60, 166
mode chaîne **variable** 14-15, 26, 28, **29**, 41, 44-45, 68-69, 76, 112, 147, 150, 152
mode **composant** 14-15, 29, 45, 60, 81
mode composé 2, 28
mode composé 15-16, **28**, 168
mode d'entrée-sortie 2, **25**
mode d'entrée-sortie 15, **25**
mode d'indice 26, 30, 106
mode d'indice **25-26**, 27, 29-30
mode **d'indice** 26, **30**, 46-47, 58, 61-62, 97-98, 105-109, 112, 149, 151-153, 165
mode de champ 16, **32**, 36, 59, 146-147, 150, 152-154
mode de synchronisation 2, **23**
mode de synchronisation 15, **23**
mode définissant 12-15, 29, 105, 162, 164
mode **définissant** **13**
mode définissant **13**, 14-15, 19, 163
mode des éléments 30, 81, 101
mode **des éléments** **29**, 30
mode des éléments 16, **30**, 36, 46, 57-59, 61, 82, 146-147, 149-150, 152-154
mode des **éléments de tampon** **24**, 25
mode des éléments de tampon **24**, 57, 74, 92, 94, 149, 151, 153
mode descripteur **22**, 149-151, 153, 155, 166, 168
mode descripteur 14, 20, **21**
mode discret 2, **16**, 26, 33, 36, 58-59, 63-64, 147, 165-168
mode discret 15, **16**, 168
mode discret 20, 25, 168
mode durée **27**, 149, 151, 166, 168-169
mode durée **27**
mode dynamique 2, 5, 7, 12, 20, 22, **37**, 44, 51, 76, 99, 154-155
mode enregistrement 26, 101, 109, 170
mode enregistrement **25-26**
mode **enregistrement** **26**, 108-109, 118, 149, 151, 153
mode **enregistrement de texte** 27, 110, 119, 149, 151
mode **enregistrement dynamique** **26**, 106, 109, 149, 151
mode **enregistrement statique** **26**, 107, 109, 149, 151
mode ensemble **18**, 54, 116, 148, 151, 156, 166
mode ensemble 16, **18**, 127
mode **ensemble** 18, 54, 140, 153
mode ensemble **avec numéros** **18**, 19, 26, 82, 148
mode ensemble **sans numéros** **18**, 97, 148
mode ensembliste 2, **20**, 58, 147-148, 151, 153, 166, 168
mode ensembliste 15, **20**
mode entier **17**, 135, 148, 151, 166, 168-169
mode entier **16**
mode entier défini par l'implémentation 5-6
mode événement **24**, 147, 149, 151, 166-167
mode événement 23, **24**
mode exemplaire 2, **23**, 149, 151, 155, 166-168
mode exemplaire 15, **23**
mode intervalle 14-16, **19**, 30, 76, 107, 109, 116, 147-149, 151, 166
mode intervalle 16, **19**
mode intervalle **avec numéros** **19**, 26
mode non composé **15**, 16, 168
mode **parent** 14-17, **19**, 147-148
mode primitif 20
mode primitif **20**
mode **primitif** **20**, 58-59, 70, 82, 97, 148, 151, 153
mode procédure 2, **22**, 23, 133, 141, 149, 151, 153, 155, 166, 168
mode procédure 14-15, **22**
mode **protégé** 2, **15**, 16, 30, 32, 146, 150-151, 153
mode protégé explicitement **15**
mode **protégé explicitement** 15-16
mode protégé implicitement **15**, 16, 30, 32, 146
mode **racine** 12, 19, 26, 68-74, 82, 97-98, 116, 140, **147**, 153, 165, 169
mode rangée 16, **30**, 34-37, 44, 58, 109, 146-147, 149-150, 152-154, 166-167
mode rangée 28, **29**, 30, 168
mode rangée 21-22, 168
mode rangée dynamique **37**, 59

mode rangée originel 16, **30**
mode rangée paramétré 37
mode rangée paramétré **29**, 30
mode rangée **paramétré** 15-16, **30**, 47, 62, 166
mode récursif 14, 148
mode repère 2, **20**, 146, 153, 155
mode repère 14-15, **20**
mode repéré 21
mode repéré **21**
mode **repéré** **21**, 43, 148, 150-151, 153-155
mode repère libre **21**, 149, 151, 155, 166-168
mode repère libre 20, **21**
mode repère lié **21**, 148, 150-151, 153-155, 166-168
mode repère lié 20, **21**
mode repéré **originel** **22**, 44, 149-151, 153-155
mode **résultant** 147
mode statique 2, **12**, 20-21, 155, 167
mode structure 2, 11, 16, 26, **31**, 32-36, 57-58, 83, 141, 146-147, 149-150, 152-154, 160-161, 166-168
mode structure 28, **31**, 32
mode structure fixe **32**
mode structure paramétré **31**, 32-33
mode structure **paramétré** 15-16, **32**, 33, 38, 58-59, 146, 149-150, 152, 154, 166
mode structure **paramétré avec marqueurs** **33**, 58-59, 146-147
mode structure **paramétré** dynamique 32, **38**, 48, 59, 70
mode structure paramétré sans marqueurs **33**
mode structure **paramétré sans marqueurs** 58-59
mode structure **variable** **32-33**, 44, 58-59, 154, 166
mode structure variable 21-22
mode structure **variable avec marqueurs** 33, 48, 58-59, 63, 165
mode structure **variable originel** 16, **33**, 38 149-150, 152, 154
mode structure **variable paramétrable** **33**, 146, 149, 152, 154
mode structure **variable sans marqueurs** **33**, 47, 58-59, 63, 165
mode tampon 2, **24**, 147, 149, 151, 153, 166-167
mode tampon 23, **24**
mode temporisation 2, **27**
mode temporisation 15, **27**
mode temps absolu **2**, **28**, 149, 151, 166-167, 169
mode temps absolu **27**
mode texte 2, 26-27, 110, 147, 149, 151, 153, 167
mode texte 25, **26**
MODIFY **104**, 105, 174
MODIFYFAIL 105, 170, 175
MODULE **134**, **137-138**, 173
module 3-5, 83-84, 120-121, 128-130, **134**, 135-136
module 75, 127, 129, **134**, 135, 137-139, 141, 160-162
module de contexte 75, **137**
module de spec 5
module de spec 75, 127-130, 135, 137, **138**, 139-141, 157, 160-162
module de spec simple 130, **138**, 140
modulion **127**, 128-129, 134-136, 138, 141, 158-162, 164
modulion distant 134-135, **136-137**, 138-139, 141
modulo **72**, 73
mot 7, 35-36, 170
mot **34**, 35-36, 151
multiplet **57**, 58, 67
multiplet 50-51, **56**, 57-59, 144
multiplet de rangée 57, 165
multiplet de rangée **56**, 57-59
multiplet de rangée avec indices 57, 164
multiplet de rangée avec indices **57**, 58, 165
multiplet de rangée sans indices 57
multiplet de rangée sans indices **56**, 58
multiplet de structure **56**, 57-59, 164
multiplet de structure avec noms de champ 57
multiplet de structure avec noms de champ 56, **57**, 59, 164
multiplet de structure sans noms de champ 57
multiplet de structure sans noms de champ **56**, 57-58
multiplet ensembliste 57-58
multiplet ensembliste **56**, 58-59
NEWMODE 13, **14**, 173
nom 2-6, 10, **11**, 13-14, 16-18, 21-23, 25, 27, 31-32, 39-40, 42, 48, 50, 52-55, 63, 80-82, 84, 86, 88, 91, 93, 127-128, 130-135, 138, 140, 155, 160, 166-167, 169
nom **10**, 11, 15, 71, 81, 127, 155, 157, 166-167
nom d'accès 2, 40, **42**, 83, 166
nom d'accès 41, **42**, 143
nom **d'accès** 162
nom **d'élément d'ensemble** **18**, 130, 140, 148, 153
nom d'élément d'ensemble 11, 54, 167
nom **d'énumération de locus** 42, **82**
nom d'énumération de locus 42, 143, 166
nom **d'énumération de valeur** 52, **81**
nom d'énumération de valeur 51-52, 166
nom **d'étiquette** **75**, 130, **134**, 140
nom d'étiquette 83-84, 87, 167
nom d'exception 4, 85, 120-121, **132**, 133, 169
nom d'exception **10-11**, 22, 88, 120
nom d'exception défini par l'implémentation 4-5, **169**

nom **d'opération prédéfinie** 95, 169
nom d'opération prédéfinie 84-85, 167
nom de champ 11, 57, 83, 164
nom de **champ** **10**, 11, 47, 57, 63, 160-161, 164
nom de champ 31, **32**, 33, 36, 38, 42, 48, 52, 58-59, 63, 83, 161
nom **de champ fixe** **32**, 33
nom **de champ marqueur** **32**, 33, 150, 152
nom de champ marqueur 31, 167
nom **de champ récurrent** **32**, 33, 36, 47, 63
nom de littéral de booléen 53, 166
nom **de littéral de vide** 55
nom de littéral de vide 54, 166
nom **de loc-identité** **40**, 42, **133**, 140, 153, 161
nom de loc-identité 42, 143, 166
nom **de locus** **40**, 42, **133**, 140, 161
nom de locus 42, 136, 143, 166-167
nom de locus faire-avec 42, **83**
nom de locus faire-avec 42, 143, 166, 170
nom de locus repère libre 167
nom de locus repère lié 167
nom **de mode** 6, 12, **13**, 14-16, 97, 162
nom de mode 16, 42-43, 49, 56-58, 63-64, 67, 96-99, 163, 166
nom de mode accès 25, 166
nom **de mode association** 25
nom de mode association 25, 166
nom **de mode booléen** 17
nom de mode booléen 17, 166
nom **de mode caractère** 17
nom de mode caractère 17, 166
nom *de mode chaîne* 28-29, 96-99, 166
nom de mode chaîne originel **28**, 29, 37
nom de mode chaîne **originel** 15
nom de mode chaîne paramétré 28, 166
nom de mode descripteur 21, 166
nom de mode discret 19-20, 78, 80-82, 96-97, 164-166
nom **de mode durée** 27
nom de mode durée 27, 166
nom de mode ensemble 18, 166
nom de mode ensembliste 20, 166
nom **de mode entier** 16
nom de mode entier 16, 166
nom de mode événement 24, 166
nom **de mode exemplaire** 23
nom de mode exemplaire 23, 166
nom de mode intervalle 19, 166
nom de mode procédure 22, 166
nom de mode rangée 29, 96-99, 166
nom de mode rangée originel **29**, 30, 37
nom de mode rangée **originel** 15
nom de mode rangée paramétré 29, 166
nom **de mode repère libre** 21
nom de mode repère libre 21, 166
nom de mode repère lié 21, 166
nom de mode structure 31, 166
nom de mode structure paramétré **31**, 166
nom de mode structure variable 31, 96, 98-99, 166
nom de mode structure variable originel **31**, 33-34, 37
nom de mode structure **variable originel** 15
nom de mode tampon 24, 166
nom **de mode temps absolu** 27
nom de mode temps absolu 27, 166
nom **de module** **134**
nom **de neumode** **15**, 19, 29, 140, 160, 164, 167, 169
nom de neumode 166
nom **de procédure** 52, 57, 86-87, **132-133**, 141-142, 153, 163
nom de procédure 84-85, 143-144, 167
nom **de procédure critique** 142
nom **de procédure générale** 22, 52, 133
nom de procédure générale 51-52, 167
nom **de processus** 6, 91, **133**, 141-142, **145**, 153, 163, 169
nom de processus 65, 145, 167
nom **de région** **135**
nom de repère de texte **10-11**, 137, 169
nom **de signal** 91, 93, 141, **145**, 153, 163
nom de signal 57, 91, 93, 167
nom **de synmode** **14**, 16, 19, 29-30, 44-45, 47, 60, 62, 71, 81-82, 105, 140
nom de synmode 166
nom **de synonyme** 13-14, **50**, 52, 140, 153, 161
nom de synonyme 51-52, 144, 166
nom de synonyme indéfini 66, 167
nom de valeur **52**, 83, 166
nom de valeur 50, **51**, 52, 144
nom **de valeur faire-avec** 52, **83**
nom de valeur faire-avec 51-52, 144, 166, 170
nom **de valeur reçue** 52, **93-94**
nom de valeur reçue 51-52, 144, 166
nom défini par l'implémentation 10, 85, 127, 167
nom non réservé 84, 167
nombre d'éléments **30**, 35, 37, 58, 149, 152, 154
nombre de valeurs **17-19**, 36, 148
noms **d'exception** 23, 149, 151
noms de champs **visibles** 141
noms **de littéral de booléen** 53, 166

noms **de mode** définis par l'implémentation 13, **169**
noms **de processus** définis par l'implémentation 5, **169**
noms **impliqués** 5, 157
noms réservés 167
non récursif **23**, 85, **132**
NONREF **22**, 48, 86, 132, **139**, 140, 173
NOPACK 30, 32, **34**, 35-36, 46-48, 82-83, 150-151, 173
NOT **73**, 173
NOTASSOCIATED 103-104, 106, 175
NOTCONNECTED 107-110, 175
nouveau préfixe **158**, 159-160, 162
nouveauté 12-13, **14**, 15, **16**, 148-149, 151-153, 164
nouveauté réelle 15, 141, 153
nul 16, 143-144, 151
NULL 21-23, 43-44, 55, 85, 91, 99, 107-108, 174
NUM 19, 30, 35, 37, 44-45, 47, 60-64, **96**, 97-98, 106, 108, 154, 174
objet composé **80**, 81
objet du monde extérieur 4, 25, **100**, 103-104
occurrence d'utilisation 5, 11, **128**, 155
octroyable **159**, 161
OD **79**, 86, 173
OF **31**, **67**, **78**, 173
ON **120**, 173
opérande-0 67, **68**
opérande-1 68, **69**
opérande-2 **69**, 70
opérande-3 69-70, **71**, 72
opérande-4 71, **72**, 73
opérande-5 72, **73**, 74
opérande-6 73, **74**, 143
opérateur affectant 76
opérateur affectant 75, **76**, 77
opérateur arithmétique additif **71**
opérateur arithmétique additif **71**, 72, 76
opérateur arithmétique multiplicatif **72**
opérateur arithmétique multiplicatif **72**, 73, 76
opérateur binaire fermé **76**
opérateur binaire fermé **76**, **77**
opérateur d'appartenance **70**
opérateur d'appartenance 69, **70**
opérateur d'inclusion ensembliste **70**
opérateur d'inclusion ensembliste 69, **70**
opérateur de concaténation de chaîne **71**
opérateur de concaténation de chaîne **71**, 72, 76
opérateur de différence ensembliste **71**
opérateur de différence ensembliste **71**, 72, 76
opérateur de préfixage 11
opérateur de répétition de chaîne **73**
opérateur de répétition de chaîne **73**
opérateur nullaire **65**
opérateur nullaire 51, **65**
opérateur relationnel 27, **70**
opérateur unaire **73**
opérateur unaire **73**
opérateur-3 **69**, 70
opérateur-4 **71**, 72
opérateurs rationnels 69, **70**
opération de connexion 26, **101**, 102, **105**, 108
opération de déconnexion **101**
opération de dissociation **100**
opération écrire 100, **101**, 105-107, **109**
opération lire **101**, 102, 105, 107, **108**, 109
opération prédéfinie par l'implémentation 5, 135, **169**
OR **68**, **76**, 173
ORIF **68**, 173
OUT **22**, 85, 131-132, 134, 173
OUTOFFILE **107**, 108, 174
OVERFLOW 64, 72-74, 81, 98, 175
PACK 30, 32, **34**, 35, 150-151, 173
paramétrable 12, 22-23, 26, 34, 41, 98, **146**, 154
paramètre d'opération prédéfinie **84**, 102
paramètre effectif 65, 84, 132, 142
paramètre effectif 57, 65, **84**, 85-86, 170
paramètre formel 65, 85, 132, 142
paramètre formel 42, 65, 127, **131**, 132-134, 143
paramètre pour associer **103**, 170
paramètre pour modifier **104**, 170
partie avec 42, 52, 79, **83**, 127
partie de commande **79**, 130
partie de commande **79**
pas 30, **34-35**, 150-151
passage de paramètres 6, 65, 85, 131-132, 169
passage par locus 131, **132**
passage par valeur 131, **132**
pile 98
portée 4-5, 127, **128**
POS **34**, 35, 150, 173
pos 151
pos 31-33, **34-35**, 36, 150-151
positionnnement du fichier **106**
postfixe 159
postfixe **158-159**, 160, 162
postfixe d'octroi 158-159, **160**, 161, 164
postfixe de saisie 158-159, **161**, 162
pourcent 113

pourcent **113**
POWERSET **20**, 163, 173
PRED 81, **96**, 97-98, 174
préfixe 158
préfixe **10**, 11, 158, 160-162
préfixe simple **10**, 160
PREFIXED **160**, 173
premier élément **46**, 47, 62, 136
priorité **89**, 90-94
priorité **89**, 90-92
PRIORITY **89**, 173
PROC **22**, **131**, 133, **140**, 163, 173
procédure 2-6, 22, 48, 55, 64-65, 84-87, 120, 128-132, 136, 142-144, 163
procédure **générale** 84, **131**
procédure rendant locus 5
procédure rendant valeur 5
procédures **in-situ** **131**
procédures **simples** **131**
PROCESS **133**, **140**, 173
processus 2, 4-6, 23-24, 27, 39, 55, 65, 74, 84, 86-95, 122-123, 125-126, 129-130, **142**, 143-145, 169
processus imaginaire le plus externe 85-86, 127, 134, **135**, 136, 142, 157, 169
produit 72
programmation par fragments **136**, 138, 140
programme 1-5, 8-12, 26, 37, 66, 75, 84, 100-101, 108-110, 120, 122, 128-129, 131, 133, **135**, 136-137, 142, 156
programme **135**
propriété de marquage et de paramétrage 12, 33, 39, **146**, 147
propriété de non-valeur 12, 23, 25-26, 33, 39-40, 51, 64, 76, 85, 133-134, 145, **147**
propriété de protection 2, 12, 16, 40, 76, 85, 90, 93-94, 99, 109, 116, 126, **146**
propriété de repérer 12, 143, **146**, 154-155
propriété héréditaire **12**, 13, 15, 17-24, 26-27, 29-30, 32-33, 148
propriété non héréditaire **12**, 16, 19, 29
propriétés dynamiques 102, 110
propriétés dynamiques **7**
propriétés statiques 5, 11, 38, 84, 138, 140, 169
propriétés statiques 7
protégé 2, 16, 32, 148, 153-154
PTR 21, 163, 174
qualificatif de conversion 112, **114**, 115
qualification de littéral **52**
quasi-déclaration 130, **139**
quasi-déclaration de loc-identité **139**, 140
quasi-déclaration de locus **139**
quasi-définition 11, 15, **130**, 138, 140-141, 152-153, 156-157
quasi-définition de signal **140**
quasi-définition de synonyme **140**, 170
quasi-domaine 130
quasi-énoncé de définition de procédure 130, 139, **140**
quasi-énoncé de définition de processus 130, 139, **140**
quasi-énoncé de définition de signal 139, **140**
quasi-énoncé de définition de synonyme 139, **140**
quasi-énoncé déclaratif **139**
quasi-énoncé définissant **139**, 140
quasi-énoncé informatif 128, **139**
quasi-énoncés **140**
quasi-liste de paramètres formels **140**, 141
quasi-nouveauté 15, 141, 153, 164
quasi-paramètre formel **140**
quotient **72**
RANGE **19**, 26, 30, 163, 173
rangée multidimensionnelle **30**
RANGEFAIL 41, 44-47, 51, 59-62, 68-70, 76, 78, 82, 98, 107, 109-110, 124-125, 148-150, 154, 175
réactivation 5, **142**
réactivation d'un processus **145**
READ **15**, 16, 30, 32, 153-154, 163, 173
READABLE **104**, 174
READFAIL 109, 175
READONLY 105-107, 110, 174
READRECORD 4, **108**, 109, 113, 118, 174
READTEXT **111**, 112, 114-118, 174
READWRITE 105-107, 174
RECEIVE **74**, **93-94**, 173
récursif **23**, 131, **132**, 143
RECURSIVE **22**, 23, **131**, 132-133, 173
récursivité **23**, 85, **132**, 149, 151, 169
REF 14, **21**, 110, 153-154, 163, 173
REGION **135**, **138**, 173
région 3-5, 99, 120-121, 128-130, 132, 134, **135**, 136, 142-145
région 127-129, **135**, 137-139, 141, 143-144, 160-162
région de spec 5
région de spec 127-130, 135, 137, **138**, 139-141, 143-144, 157, 160-162
région de spec simple 130, **138**, 140
régionalement sûr 40, 76, 85-86, 99, **144**
régionalité 65, 85-86, 103, 106-107, 109, 119, 140-141, **143**, 144, 169-170
règles d'identification 8, 11, 156
règles de vérification des modes 5, **146**
relations d'équivalence 5, **148**
relations de compatibilité **148**

REM 72, **73**, 173
REMOTE **136-137**, 173
remplissage 114-116
repérabilité 2, 36, 41
repérable 2, 20, 34, 36, 40, **41**, 42-49, 74, 82-83, 85-86, 97-98, 102, 109, 112, 126, 132-133, 140, 170
repère **accès** 107, 110, 118-119
repère **d'enregistrement de texte** 110, 118
repère libre 2, 20, 43
repère libre dérepéré 43
repère libre dérepéré 41, 43, 143
repère lié 2, 20, 42
repère lié dérepéré 42
repère lié dérepéré 41, **42**, 43, 143
représentation **34**, 35-36
représentation textuelle de nom 11, 75, 81, 83-84, 133-135, 137, 139, 148, 157, 160-161
représentation textuelle de nom **10**, 11, 138, 141, 152-153, 155-164, 167
représentation textuelle de nom **canonique** **11**, 155
représentation textuelle de nom de mode 160-161, 164, 167
représentation textuelle de nom définie par l'implémentation 157
représentation textuelle de nom impliquée 158, **162**, 163
représentation textuelle de nom prédéfinie 159
représentation textuelle de nom préfixe 155, 158
représentation textuelle de nom préfixe **10**, 11
représentation textuelle de nom simple **8**, 116
représentation textuelle de nom simple **8**, 9-10, **11**, 75, 115, 131, 133-141, 155, 161-163
représentation textuelle de nom simple réservée **9**
représentation textuelle de nom simple **réservée** 9, 84
représentations textuelles de nom simple **spéciales** 8, **9**, 115
reste de la division **72**
RESULT **86**, 173
résultat 2-5, 11, 32, 51, 64, 66-70, 73, 76, **86**, 92, 103, 108, 131, 142, 148-150, 154
résultat **86**
RETURN **86**, 173
RETURNS **22**, 173
ROW 9, **21**, 163, 173
saisissable **159**, 162
SAME 105-106, 174
SECS **124**, 174
SEIZE 137, **161**, 173
sélecteur 33, 78, 165
sélection 2-3, 78, 164
sélection de cas **164**, 165
sémantique 7-10, 32, 40, 42, 47, 49, 52, 63, 76, 81, 91-92, 102-104, 112, 118-119, 131, 136-137
sémantique **7**
semblable 13, 140-141, **148**, **151**, 152
SEND **91-92**, 173
SENDFAIL 91, 175
séquençable 4, **101**, 104-106
séquence de contrôle 54, **55**
SEQUENCIBLE **104**, 174
SET **18**, **90**, **93-94**, 105, 163, 173
SETEXTRECORD 118-119, 174
SETTEXTACCESS 119, 174
SETTEXTINDEX 119, 174
SHORT-INT **17**
SIGNAL **140**, **145**, 173
signal 5, 91-93, 130, 145, 163
signal à choisir 130
signal à choisir **93**, 127, 129
similaire 13, 147, **148**, 149, 151, 155-156, 169
SIMPLE **131**, 132, 173
simple 131, **132**
SIZE 16, 49, **96**, 97-98, 174
solution alors **67**
solution cas de valeur **67**
solution sinon **67**
somme 71
sous-expression **67**, 68, 144
sous-locus **accès** 26, 40, 105, 110, 118
sous-locus **enregistrement de texte** 40
sous-opérande-0 **68**
sous-opérande-1 **69**
sous-opérande-2 **69**, 70
sous-opérande-3 **71**
sous-opérande-4 **72**, 73
SPACEFAIL 65, 77-79, 85, 90, 93, 95, 99, 120, 130, 175
SPEC **136**, **138**, 139, 160, 173
spec de module 130, **138**, 139-141, 164
spec de paramètre 85-86
spec de paramètre **22**, 23, 57, 127, 131-134, 140
spec de région 130, **138**, 139-141, 164
spec de résultat 131-132
spec de résultat **23**, 48-49, 57, 64, 85-87, **132**, 149, 151, 153, 163
spec de résultat **22**, 23, 127, 131-133, 140
spec distante **136-137**, 138-139
spécification d'étiquettes de cas 32, 58, **78**, **164**, 165
spécification d'étiquettes de cas 31, 33, 67, 78, **164**, 165
spécification de format 112, **113**

specs de paramètre **23**, 85, **132**, 133, 149, 151, 153, 163
START **65**, 173
STATIC **39**, 40, 136, **139**, 142, 173
statique **41**, 74, **136**, 140
STEP **34**, 35-36, 150, 173
STOP **88**, 173
STRUCT 14, **31**, 35, 154, 163, 173
structure de programme 1, 5, **127**
structure **variable sans marqueurs** 170
SUCC 81, **96**, 97-98, 174
supérieur à 70, 72, 82, 98, 106, 109, 112, 117, 119, 154
supérieur ou égal à 70
sûr **14**
symbole d'affectation **76**
symbole d'affectation 39-40, 75, **76**, 80
symbole spécial **8**, **172**
SYN **50**, **140**, 173
SYNMODE **14**, 173
synonyme 13, **14**, 15-16, 29-30, 44-45, 47, 60, 62, 71, 81-82
syntaxe **7**, 8, 57, 78, 136
syntaxe dérivée **7**, 30-31, 57, 77, 114, 116, 137, 158
syntaxe stricte 7, 46, 149-150, 152
TAGFAIL 41-42, 48, 51-52, 59, 63, 70, 76, 109, 148-149, 175
taille **16**, 26, 32, 101
taille de pas **34**, 35-36, 151
taille de tranche **45**, 46-47, 60-62, 136
tampon 5, 22, 39, 91-92, 130
tampon à choisir **94**, 127, 129
temporisable 4, 89-90, 92-94, 122-123, 125-126, 170
terminaison d'un processus **142**
TERMINATE **98**, 99, 136, 170, 174
terminé 9-10, 79-82, 103, 113, 120, 129, 131, **142**
TEXT **26**, 163, 173
texte de format 112, **113**
TEXTFAIL 112, 116-117, 119, 175
THEN 9, **67**, **77**, 173
THIS **65**, 142, 173
TIME 27, 125, 163, 174
TIMEOUT **122**, 173
TIMERFAIL 123-124, 170, 175
TO **80**, **91**, **140**, 141, **145**, 173
traitement des exceptions **120**
tranche de chaîne **45**, 60, 112, 116
tranche de chaîne 41, **45**, 60, 136, 143
tranche de rangée 36, **47**
tranche de rangée 41, **46**, 47, 62, 136, 143
trancher 2
transmission du résultat 6
troncation 114-115
troncation du fichier **106**
TRUE 17, 53, 67-68, 70, 72, 77, 82, 103-105, 107-109, 115, 118, 174
type de chaîne **28**, 29
union 32-33, 68, 163
UP 28, **45-46**, **60**, **62**, 173
UPPER **96**, 97-98, 174
USAGE 105-107, 174
usage **102**, 106-110
v-équivalent 13, **148-149**, 155
valeur 1-5, 12-13, 15-32, 34-37, 39-43, 45, 47-48, 50-65, **66**, 67-68, 70-74, 76-78, 80-82, 84-86, 89-117, 119, 123-125, 130-132, 140, 142, 145, 148-152, 154, 160, 164-165, 169-170
valeur 39-40, 56-59, **66**, 75-76, 84-87, 91-92, 98-99, 103-104, 132, 143-144, 161, 165, 168
valeur absolue 96
valeur booléenne 28, 54, 68-70, 73, 101, 115
valeur chaîne 28, 60, 73, 109, 112, 118
valeur chaîne de bits 28, 56, 68-69, 71, 73, 116
valeur chaîne de caractères 28, 55, 71, 116
valeur champ de structure **63**
valeur champ de structure 50, **63**, 144, 164
valeur composée 28, 30-31, 66
valeur **constante** 3, 170
valeur constante 13, 39-40, 50, 57, 140, 144, 168
valeur d'association **101**, 170
valeur de mode ensembliste 20, 57, 68-71, 73, 80-81, 96
valeur de pas 80-81
valeur de pas **80**, 81-82
valeur de sélecteur 164, **165**
valeur de texte 110
valeur définie 3, 142
valeur élément de chaîne 60
valeur élément de chaîne 50, **60**
valeur élément de rangée **61**
valeur élément de rangée 50, **61**, 144
valeur ensembliste vide **57**, 97-98
valeur entière **4**, 17-18, 53, 71-73, 96, 115
valeur exemplaire **23**, 65, 88, 90, 93-94, **142**, 169
valeur exemplaire vide **55**
valeur finale 80-81
valeur finale **80**, 82
valeur indéfinie 3
valeur indéfinie **66**
valeur **indéfinie** 3, 39-40, 50, 58-59, 64, **66**, 76, 86, 98, 108, 132

valeur initiale 80-81

valeur initiale **80**, 82

valeur primitive **51**, 83, 147

valeur primitive **50**, 51, 74, 83, 96, 144, 167-168

valeur primitive chaîne 60-61, 168

valeur primitive descripteur 43-44, 143, 168

valeur primitive durée 122-123, 168

valeur primitive exemplaire 91, 168

valeur primitive procédure 84-86, 168

valeur primitive rangée 61-62, 144, 167

valeur primitive repère 98-99, 168

valeur primitive repère libre 43, 143, 168

valeur primitive repère lié 42-43, 143, 167

valeur primitive structure 63, 83, 144, 164, 168

valeur primitive temps absolu 123, 125, 167

valeur procédure vide **55**

valeur rangée 30, 57, 61-62, 109

valeur repère 2-3, **21**, 22, 98-99, 107-108, 110

valeur repère **affectée** **99**, 136

valeur repère vide **55**

valeur structure 31-32, 52, 57, 63, 83, 109, 170

valeur tranche de chaîne **60**

valeur tranche de chaîne 50, **60**, 61

valeur tranche de rangée **62**

valeur tranche de rangée 50, **62**, 144

valeurs anonymes **18**

valeurs d'accès **102**

valeurs de durée 170

valeurs nommées **18**

valeurs procédure **22**, 131

VARIABLE **104**, 174

variable 4, **101**, 104, 106-107, 112, 115-116

variable sans marqueurs

VARYING 27, **28**, 29, 173

vérification des modes 5, 13, 49, 63

verrouillé **142**, 143, 145

vide 11, 23, 28, 39-40, 57, 81, 85, 94, 101, 104, 110, 132, 137, 158, 160-163

vide **87**, 128, 137, 139, 158

visibilité 1, 4-5, 83, 128, 130, 134-135, 138, 155, **156**, 157-158, 160-162

visibilité de noms de champ **164**

visible 4, **128**, 138, **156**, 157, 163-164

WAIT **125**, 126, 174

WHERE 105-106, 174

WHILE **82**, 173

WITH **83**, 173

WRITEABLE **104**, 174

WRITEFAIL 110, 175

WRITEONLY 105-107, 109, 174

WRITERECORD 4, **108**, 109-110, 113, 118, 174

WRITETEXT **111**, 112, 114-119, 174

XOR **68**, **76**, 173

SÉRIES 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**

Imprimé en Suisse
Genève, 2010