UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

LIVRE JAUNE

TOME VI – FASCICULE VI.8

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

AVIS Z.200

VII^e ASSEMBLÉE PLÉNIÈRE
GENÈVE, 10–21 NOVEMBRE 1980

Genève 1981

UNION INTERNATIONALE DES TELECOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

LIVRE JAUNE

CORRIGENDUM AU FASCICULE VI.8

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

Première liste de fautes de copie dans l'Avis Z.200

VII^e ASSEMBLÉE PLÉNIÈRE
GENÈVE, 10–21 NOVEMBRE 1980

Genève 1982

CORRIGENDUM AU FASCICULE VI.8 DU LIVRE JAUNE

Première liste de fautes de copie dans l'Avis Z.200

Langage évolué du CCITT (CHILL)

1. Introduction

Ce papier contient une liste de corrections de "fautes de copie" dans la définition de CHILL, Z.200. Une "faute de copie" est définie comme une erreur dont la correction ne change pas l'interprétation que des personnes averties auraient donnée á la définition.

Le nombre (1) avant chaque correction indique qu' il s' agit de la première liste de fautes de copie. Les corrections futures seront ajoutées à la liste et auront les numéros (2), (3), etc.

2. Liste de corrections de fautes de copie

(1) page 17, lignes 11,12

- remplacer: .. si c'est une M-classe par valeur ou une M-classe par dérivation...

  par: .. si c'est une M-classe par valeur, une M-classe par dérivation ou une M-classe par repère..

(1) page 26

- ajouter une troisième condition statique (qui s' applique à la syntaxe dérivée des modes intervalle)

  L'expression littérale entière dans le cas de *BIN* doit rendre une valeur non négative

(1) page 29, section 3.6.4.

- ajouter une condition statique:
  Si le mode repéré originel est un mode structure, il doit être paramétrable

(1) page 45

- ligne 1,
  remplacer: Dans les déclarations et les spécifications des paramètres formels...

  par: Dans les déclarations, spécifications de paramètres et de résultat...

(1) page 54:

- remplacer la première condition statique

  par: La classe de la *valeur* ou *valeur constante* doit être compatible avec le *mode* et la valeur donnée doit être une des valeurs définies par le *mode*

(1) page 58, section 4.2.2.

- remplacer la première condition dynamique

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

- Dans la troisième condition dynamique, troisième ligne, souligner: récurrent

(1) page 78, sémantique, troisième paragraphe, dernière ligne:

- remplacer: .. voir section 9.1.4.)

par: .. voir section 9.1.3.)

(1) page 80, section 5.2.5, point 6, cinquième ligne:

- remplacer: .. pas (*ELSE*) doit être..

par: .. pas (*ELSE*) ni *<indifférent>* doit être ..

(1) page 92, section 5.2.16, sémantique de *GETSTACK*, deuxième ligne:

- remplacer: ... section 7.4 ...

par: ... section 7.9 ...

- deuxième propriété statique, ligne 2

- remplacer: ... la classe ...

par: ... la classe résultante ...

(1) page 93

- ajouter une nouvelle condition statique avant:

L'*expression rangée* comme argument de ...

Le *locus de mode statique* argument de *SIZE* doit être repérable

(1) page 94

- La première ligne doit commencer comme suit:

- Le mode structure variable doit être paramétrable et il doit y avoir autant d'expressions...

(1) page 111, ligne 8:

- remplacer: .. Chaque étiquette de cas...

par: .. Chaque liste d'étiquettes de cas...

(1) page 113, syntaxe, lignes (6.1) et (8.1)

- remplacer: *<expression>*

par: *<expression discrète>*

(1) page 114, sémantique, ligne 18

- remplacer: l'action faire.

par: l'action faire, ou si le filet de l'action faire est entamé et passe les bornes, ou si on quitte l'action faire par une action revenir ou une action arrêter.

(1) page 120, section 6.7, sémantique, ligne 1:

- remplacer: Une action appeler cause....

par: Une action appeler cause soit l'appel d'une procédure ou d'une opération prédéfinie. Une action appeler cause....

(1) page 121, conditions statiques, quatrième règle de compatibilité (attribut LOC):

- ajouter: Si l'*appel de procédure* n'est pas régional le *locus* (effectif) ne peut pas être régional (voir section 8.2.2.).

(1) page 130, section 6.19.2, ligne 10 à partir du bas:

-remplacer: ... noms de valeur introduits ...

par: ... noms de valeur reçue introduits ...

(1) page 134, section 7.1, lignes 1 et 2:

- remplacer: .. *région, action mettre en attente et choisir, action recevoir et choisir,...*

par: .. *région, action recevoir et choisir,...*

(1) page 143, section 7.4

- ajouter la condition statique suivante:
  Tous les noms dans la *liste d'exceptions* doivent être différents.

(1) page 149, section 8.2.1, ligne 9

- remplacer: .. si et seulement si ...

  par: .. si, soit ...

(1) page 151, section 8.2.2., point 2 (valeur), ligne 6:

- remplacer: C'est un *contenu de locus* qui est régional

  par: C'est un *contenu de locus* dont le locus contenant est régional ...

(1) page 154, section 8.4

- ajouter un nouveau paragraphe avant "Action recevoir tampon et choisir".

Expression recevoir (voir section 5.2.18)

Quand un processus évalue une expression recevoir, il réactive un autre processus, si et seulement si l'ensemble des processus envoyants en attente sur le locus tampon spécifié n'est pas vide. Dans ce cas, il reçoit une valeur de la plus haute priorité parmi les valeurs dans le locus tampon, ou bien des processus envoyants en attente. En recevant une valeur d'un tampon, le processus enlève la valeur du tampon et un processus envoyant en attente qui a la valeur de la plus haute priorité est sélectionné et devient actif suivant un algorithm de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de l'ensemble des processus envoyants en attente et sa valeur est mise dans le tampon avec la priorité spécifée. En recevant une valeur directement d'un processus envoyant en attente, le processus en attente qui porte la valeur de la plus haute priorité est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de l'ensemble des processus envoyants en attente et sa valeur est reçue.

(1) page 159, section 9.1.1.7, lignes 1 et 2:

- remplacer: ... n'est pas un mode composite...

  par: ... est un mode discret ou chaîne...

(1) page 168, section 9.1.2.6., règle 6, point 3, paragraph 3,
dernière ligne:

- remplacer: ... la liste de valeurs de N.

par: ... la liste de valeurs de M.

(1) page 203, ligne 34

- remplacer: *END stacks-1*

par: *END stacks_1*

(1) page 208

- insérer entre les lignes 140 et 141

```
140a  DCL c column;
```

(1) page 209

- remplacer les lignes 28-32

par:

```
27a  MANIPULATE:
27b MODULE;
27c     SEIZE NODE, REMOVE, INSERT;
28      DCL NODE_A NODE:= (:NULL, NULL, 536:);
29      REMOVE ();
30      REMOVE ();
31      INSERT (NODE_A);
31a END MANIPULATE;
32  END CIRCULAR_LIST;
```

(1) page 211, ligne 6

- remplacer: ... lets calla through ...

par: ... lets call through ....

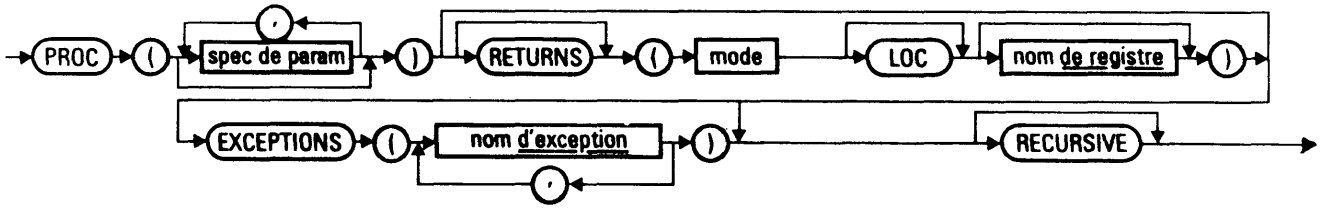
(1) page 212

- remplacer ligne 15:

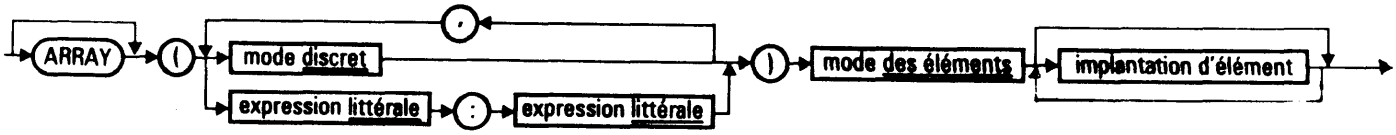
par: 15 *ACQUIRE, RELEASE, CONGESTED, STEP, READOUT, READY;*

(1) page 224

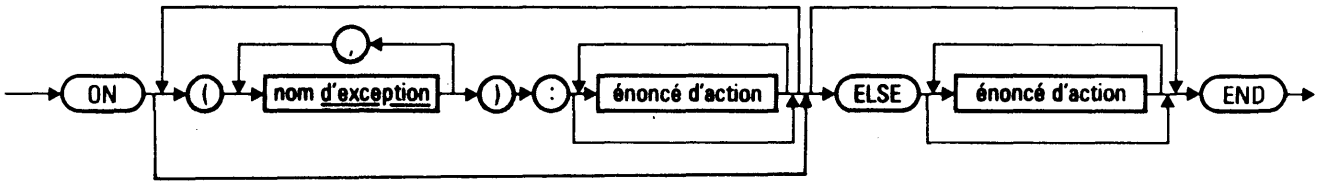
- remplacer le diagramme syntaxique PROC par:



- remplacer le diagramme syntaxique ARRAY par:



(1) page 229, remplacer le diagramme syntaxique de filet par:



Imprimé en Suisse — ISBN 92-61-01122-5

UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

LIVRE JAUNE

TOME VI – FASCICULE VI.8

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

AVIS Z.200

VII^e ASSEMBLÉE PLÉNIÈRE
GENÈVE, 10–21 NOVEMBRE 1980

Genève 1981

ISBN 92-61-01122-5

CONTENU DU LIVRE DU CCITT
EN VIGUEUR APRÈS LA SEPTIÈME ASSEMBLÉE PLÉNIÈRE (1980)

LIVRE JAUNE

Tome I – Procès-verbaux et rapports de l'Assemblée plénière.

Vœux et résolutions.

Avis sur:
- l'organisation du travail du CCITT (série A);
- les moyens d'expression (série B);
- les statistiques générales des télécommunications (série C).

Liste des Commissions d'études et les Questions mises à l'étude.

Tome II

FASCICULE II.1 – Principes généraux de tarification – Taxation et comptabilité dans les services internationaux de télécommunications. Avis de la série D (Commission III).

FASCICULE II.2 – Service téléphonique international – Exploitation. Avis E.100 à E.323 (Commission II).

FASCICULE II.3 – Service téléphonique international – Gestion du réseau – Ingénierie du trafic. Avis E.401 à E.543 (Commission II).

FASCICULE II.4 – Exploitation et tarification des services de télégraphie et de «télématique».[1] Avis de la série F (Commission I).

Tome III

FASCICULE III.1 – Caractéristiques générales des communications et des circuits téléphoniques internationaux. Avis G.101 à G.171 (Commissions XV, XVI, CMBD).

FASCICULE III.2 – Systèmes internationaux analogiques à courants porteurs – Caractéristiques des moyens de transmission. Avis G.211 à G.651 (Commissions XV, CMBD).

FASCICULE III.3 – Réseaux numériques – Systèmes de transmission et équipement de multiplexage. Avis G.701 à G.941 (Commission XVIII).

FASCICULE III.4 – Utilisation des lignes pour la transmission des signaux autres que téléphoniques – Transmissions radiophoniques et télévisuelles. Avis des séries H et J (Commission XV).

Tome IV

FASCICULE IV.1 – Maintenance; principes généraux, systèmes internationaux à courants porteurs, circuits téléphoniques internationaux. Avis M.10 à M.761 (Commission IV).

FASCICULE IV.2 – Maintenance des circuits internationaux pour la transmission de télégraphie harmonique ou de fac-similé – Maintenance des circuits internationaux loués. Avis M.800 à M.1235 (Commission IV).

FASCICULE IV.3 – Maintenance des circuits radiophoniques internationaux et transmissions télévisuelles internationales. Avis de la série N (Commission IV).

FASCICULE IV.4 – Spécifications des appareils de mesure. Avis de la série O (Commission IV).

---

[1] Le terme «service de télématique» est provisoire.

Tome V – Qualité de la transmission téléphonique. Avis de la série P (Commission XII).

Tome VI

FASCICULE VI.1 – Avis généraux sur la commutation et la signalisation téléphoniques – Interface avec le service maritime. Avis Q.1 à Q.118 *bis* (Commission XI).

FASCICULE VI.2 – Spécifications des systèmes de signalisation N^{os} 4 et 5. Avis Q.120 à Q.180 (Commission XI).

FASCICULE VI.3 – Spécifications du système de signalisation N° 6. Avis Q.251 à Q.300 (Commission XI).

FASCICULE VI.4 – Spécifications des systèmes de signalisation R1 et R2. Avis Q.310 à Q.490 (Commission XI).

FASCICULE VI.5 – Centraux numériques de transit pour applications nationales et internationales – Interfonctionnement des systèmes de signalisation. Avis Q.501 à Q.685 (Commission XI).

FASCICULE VI.6 – Spécifications du système de signalisation N° 7. Avis Q.701 à Q.741 (Commission XI).

FASCICULE VI.7 – Langage de spécification et de description fonctionnelles (LDS) – Langage homme-machine (LHM). Avis Z.101 à Z.104 et Z.311 à Z.341 (Commission XI).

FASCICULE VI.8 – Langage évolué du CCITT (CHILL). Avis Z.200 (Commission XI).

Tome VII

FASCICULE VII.1 – Transmission et commutation télégraphiques. Avis des séries R et U (Commission IX).

FASCICULE VII.2 – Equipements terminaux pour les services de télégraphie et de «télématique».[1] Avis des séries S et T (Commission VIII).

Tome VIII

FASCICULE VIII.1 – Communication de données sur le réseau téléphonique. Avis de la série V (Commission XVII).

FASCICULE VIII.2 – Réseaux de communications de données; services et facilités, équipements terminaux et interfaces. Avis X.1 à X.29 (Commission VII).

FASCICULE VIII.3 – Réseaux de communications de données; transmission, signalisation et commutation, réseau, maintenance, dispositions administratives. Avis X.40 à X.180 (Commission VII).

Tome IX – Protection contre les perturbations. Avis de la série K (Commission V). Protection des enveloppes de câble et des poteaux. Avis de la série L (Commission VI).

Tome X

FASCICULE X.1 – Termes et définitions.

FASCICULE X.2 – Index du Livre jaune.

---

[1] Le terme «service de télématique» est provisoire.

LANGAGE ÉVOLUÉ DU CCITT (CHILL)
(GENÈVE, 1980)

1.0 INTRODUCTION

Cet avis définit CHILL, le langage de programmation de haut niveau du CCITT. CHILL signifie "CCITT High Level Language".

Une autre définition de CHILL, de forme mathématique stricte, sera contenue dans un Manuel CCITT. Un autre Manuel CCITT, connu comme "Introduction to CHILL", sert d'introduction au langage.

1.1 GENERALITES

CHILL a été conçu en premier lieu pour la programmation des centraux téléphoniques à commande par programmes enregistrés (SPC). Cependant, il est considéré comme suffisamment général pour d'autres applications (par exemple la commutation de messages, la commutation par paquets, la modélisation, etc.).

CHILL a été conçu avec les exigences suivantes à l'esprit (voir la question 8/XI de la période d'études 1977-1980):

- améliorer la fiabilité en permettant un grand nombre de contrôles à la compilation;
- permettre la génération d'un code objet très efficace;
- être flexible et puissant afin de couvrir toute la gamme des applications et d'exploiter différentes espèces de matériel;
- encourager l'écriture de programmes modulaires et structurés;
- être facile à apprendre et utiliser.

CHILL implique l'existence d'un environnement pour le développement des programmes. Cet environnement peut comporter la mise en oeuvre, entre autres éléments, de la compilation séparée, d'entrées-sorties, et d'outils de mise au point. Ces éléments ne sont pas définis par cet avis.

Les programmes CHILL peuvent être écrits d'une façon indépendante du matériel pour la classe des machines connues pour être soit utilisées soit proposées pour les centraux téléphoniques SPC.

CHILL n'essaye pas de fournir des constructions spécifiques à chaque application mentionnée ci-dessus, mais a plutôt une base générale et un nombre de possibilités convenant à chaque application particulière.

CHILL comme langage, est indépendant des machines. Une implémentation particulière peut cependant contenir des objets du langage définis par l'implémentation. Des programmes utilisant de tels objets ne sont pas en général portables.

CHILL est conçu en supposant qu'il sera compilé depuis le texte source jusqu'au code objet. Il n'est pas expressément conçu pour rendre possible la compilation en une passe ou pour rendre minimale la taille des compilateurs.

Pour permettre la sécurité sans pertes inacceptables d'efficacité, un grand nombre de contrôles peuvent être réalisés statiquement. Un petit nombre de règles du langage ne peuvent être vérifiées que dynamiquement. Le non respect d'une telle règle produit une exception à l'exécution. Cependant, la génération de contrôles dynamiques pour ces exceptions est optionnelle, sauf si un filet d'exception est spécifié par le programmeur.

1.2 VUE GENERALE 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 pourvoit 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

Les objets informatifs manipulables de CHILL sont les valeurs et les locus où des valeurs peuvent être placées.

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 ensembliste	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.

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. Un mode explicitement noté dans un programme CHILL est toujours statique.

Ni les modes dynamiques ni les classes n'ont de notations en CHILL. Ils sont introduits uniquement dans le métalangage pour décrire des conditions de contexte statiques et dynamiques.

1.4 LOCUS ET LEURS ACCES

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;

3. les déclarations de locus avec base.

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*, 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: prendre une sous-chaîne, 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. (Il faut noter que le mot "dynamique" s'applique uniquement au mode; le locus lui-même n'est pas dynamique, c.-à-d. qu'il ne varie pas durant l'exécution; seules ses propriétés ne peuvent être complètement déterminées statiquement.)

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 OPERATIONS*

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.-à-d. qu'il n'est besoin de la calculer qu'une seule fois. Les valeurs constantes ou littérales sont supposées être évaluées avant l'exécution et ne peuvent générer d'exceptions. Une valeur peut être régionale auquel cas elle peut repérer d'une façon ou d'une autre des locus régionaux. Une valeur peut être composée, c.-à-d. contenir des sous-valeurs.

Les énoncés de définition de synonymes 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 séquentielle 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 démarrer, arrêter, mettre en attente, continuer, envoyer, mettre en attente et choisir et recevoir et choisir ainsi que l'évaluation d'une expression recevoir.

1.7 *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 à l'opération prédéfinie *GETSTACK*), 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.-à-d. 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 et 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.

1.8 EXECUTION CONCURRENTE

CHILL prévoit l'exécution concurrente d'unités de programme. Le processus est l'unité d'exécution concurrente. L'action 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 ou différentes définitions 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.

1.9 PROPRIETES SEMANTIQUES GENERALES

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.

1.10 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.

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.-à-d. 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.

La plupart des exceptions ont un nom. Ce nom 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.

1.11 OPTIONS POUR L'IMPLEMENTATION

CHILL permet des modes entier définis par l'implémentation, des opérations prédéfinies définies par l'implémentation, des définitions de processus définies par l'implémentation et des filets 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 a 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. 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 (imaginaire) le plus extérieur. Si ce filet reçoit le contrôle après occurrence d'une exception, l'implémentation peut décider des actions à accomplir.

2.0 PRELIMINAIRES

2.1 METALANGAGE

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 en utilisant 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 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 sous-catégorie sémantique (voir 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.

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).

2.1.2 DESCRIPTION SEMANTIQUE

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.-à-d. 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 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 partie soulignée est parfois utilisée dans le texte comme adjectif qualifiant le non terminal. Par exemple, la phrase "l'*expression* est constante" est identique à "l'*expression* est une *expression constante*".

La section conditions dynamiques décrit les conditions dépendant du contexte qui doivent être satisfaites pendant l'exécution. Dans certains cas, des conditions sont statiques si et seulement si aucun mode dynamique n'est impliqué. Dans ce cas, la condition est mentionnée dans les conditions statiques et référence y est faite dans les conditions dynamiques.

Dans la description sémantique, les non terminaux sont écrits en italique sans crochets angulaires pour indiquer les objets syntaxiques.

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 no. *6* ligne *20*.

2.1.4 REGLES D'IDENTIFICATION DANS LE METALANGAGE

Parfois, la description sémantique mentionne des noms spéciaux de CHILL (voir Appendice C). Les noms spéciaux sont toujours utilisés avec leur signification CHILL et ne sont donc pas influencés par les règles d'identification d'un programme CHILL existant.

2.2 VOCABULAIRE

Les programmes sont représentés au moyen de l'alphabet CCITT no. 5, avis V.3 (voir Appendice A1). Il est possible de représenter tout programme CHILL avec un ensemble minimum de caractères qui est un sous-ensemble du code de base de l'alphabet CCITT no. 5 (voir Appendice A2).

Les éléments lexicaux de CHILL sont:

- les symboles spéciaux
- les noms
- les littéraux

La liste des symboles spéciaux figure à l'Appendice B.

Les noms sont formés d'après la syntaxe suivante:

syntaxe:

<nom> ::= *(1)*

*<lettre> {<lettre> | <chiffre> |_}** *(1.1)*

Le caractère souligné (_) fait partie du nom, c.-à-d. que le nom *LIFE_TIME* est différent du nom *LIFETIME*. Dans le cas où un alphabet comprenant les lettres minuscules est disponible, celles-ci peuvent être utilisées dans les noms. Lettres majuscules et minuscules sont différentes, par exemple *Status* et *status* sont deux noms différents.

Le langage possède un certain nombre de noms spéciaux à signification prédéterminée, voir Appendice C. Certains d'entre eux sont réservés, c.-à-d. qu'ils ne peuvent être utilisés pour d'autres usages que s'ils sont explicitement libérés par la directive de libération.

Au cas où un alphabet avec lettres majuscules et minuscules est utilisé, les noms spéciaux peuvent être soit entièrement en représentation majuscule soit entièrement en représentation minuscule. Les noms réservés le sont uniquement dans la représentation choisie (par exemple, si les minuscules sont choisies, *row* est réservé, *ROW* non).

2.3 ESPACEMENTS

Les espaces peuvent être utilisés pour délimiter les éléments lexicaux d'un programme. Les éléments lexicaux sont terminés par le premier caractère qui ne peut pas en faire partie. Par exemple, *IFBTHEN* sera considéré comme *nom* 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 (*/**). Des espaces contigus ont le même effet de délimitation qu'un espace unique.

2.4 COMMENTAIRES

syntaxe:

<commentaire> ::= (1)
/ <chaîne de caractères> */* (1.1)

<chaîne de caractères> ::= (2)
*{<caractère>}** (2.1)

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

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

conditions statiques: La *chaîne de caractères* ne peut contenir la séquence spéciale: astérisque, barre oblique (**/*).

exemples:

4.1 / from collected algorithms from CACM nr.93 */* (1.1)

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'alphabet CCITT no. 5 (positions FE_0 à FE_5) ne sont pas mentionnées dans la description de la syntaxe

acontextuelle de CHILL. Cependant, une implémentation peut utiliser ces commandes de mise en page dans les programmes CHILL. Ils ont alors le même effet de délimitation qu'un espace. Ils ne peuvent pas être utilisés à l'intérieur d'éléments lexicaux.

2.6 DIRECTIVES AU COMPILATEUR

syntaxe:

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

<directive> ::=                                             (2)
        <directive CHILL>                                   (2.1)
      | <directive d'implémentation>                        (2.2)

<directive CHILL> ::=                                       (3)
        <directive de libération>                           (3.1)

<directive de libération> ::=                               (4)
        FREE(<liste de noms réservés>)                      (4.1)

<liste de noms> ::=                                         (5)
        <nom> {,<nom>}*                                     (5.1)
```

sémantique: Une clause de directive donne de l'information au compilateur. Sauf pour la directive de libération, cette information est spécifiée dans un format défini par l'implémentation.

Une directive d'implémentation ne peut influencer la sémantique d'un programme, c.-à-d. 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 directive de libération s'applique à une unité de compilation. Elle libérera les noms réservés spécifiés dans la *liste de noms réservés* de telle façon qu'ils puissent être redéfinis dans l'unité de compilation.

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 section 9.2.8).

conditions statiques: Le symbole de fin de directive facultatif (<>) ne peut être omis que s'il est placé juste avant un point-virgule (c.-à-d. que la *clause de directive* se termine par le premier <> ou point-virgule. Cependant, le point-virgule n'appartient pas à la *clause de directive*. En conséquence, la *directive* ne

peut contenir ni de symbole <> ni de point-virgule, sauf entre parenthèses, voir plus loin). Si des parenthèses apparaissent dans une *clause de directive*, elles doivent être convenablement équilibrées et si un point-virgule ou un symbole de fin de directive apparaît entre parenthèses, il ne termine pas la *directive*.

exemples:

15.1	*<> FREE (STEP)*	*(1.1)*
15.1	*FREE (STEP)*	*(4.1)*

3.0 MODES ET CLASSES

3.1 GENERALITES

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. Ce mode est toujours compatible avec la classe de la valeur et cette valeur est une des valeurs définies par le mode. Des valeurs fortes sont requises dans les contextes de valeur où 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*.

Les modes dynamiques n'ont pas de notations en CHILL. Cependant, pour la description, des notations virtuelles sont introduites dans ce document pour noter les modes dynamiques. Les notations virtuelles seront précédées du caractère perluète (*&*), par exemple *&VM(i)* désigne un mode dynamique paramétré au moyen du paramètre *i* déterminé à l'exécution.

De plus, à certains endroits on utilise des notations virtuelles pour les modes statiques. On le fait pour les modes qui ne sont pas ou ne peuvent pas être explicitement notés dans les textes de programme mais sont introduits virtuellement par certaines constructions du langage. Ces modes sont aussi notés au moyen de notations virtuelles précédées du caractère perluète.

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 tout mode M, il existe 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 tout mode M de nouveauté nil (voir section 9.1.1.1), il existe 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 ou une M-classe par dérivation où M est un mode dynamique.

3.1.3 PROPRIETES DES MODES, DES CLASSES ET LEURS RELATIONS

Toutes les propriétés fondamentales des modes et des classes et leurs relations sont définies au chapitre 9. On donne ci-après une liste de ces propriétés et relations:

1. Un mode M a une nouveauté.

2. Un mode M peut être protégé.

3. Un mode M peut avoir la propriété de protection.

4. Un mode M peut avoir la propriété de repérer.

5. Un mode M peut avoir la propriété de marquage et de paramétrage.

6. Un mode M peut avoir la propriété de synchronisation.

7. Un mode M peut être défini par un mode N.

8. Un mode M peut être compatible en lecture avec un mode N (asymétrique).

9. Un mode M peut être compatible avec une classe C (dans ce cas, on dit que C est compatible avec M).

10. Une classe C peut avoir un mode racine.

11. Une classe C peut être compatible avec une classe D (symétrique).

12. Etant donnée une liste de classes compatibles, il existe une classe résultante.

Des propriétés spécifiques sont définies pour chaque mode à la section appropriée. Une propriété est dite héréditaire si, lorsqu'elle est vraie pour un mode donné, elle est également vraie pour tous les noms de mode définis par ce mode. Par conséquent, les propriétés héréditaires ne seront pas explicitement définies pour les noms de mode. Chaque propriété valable pour un mode est également valable pour ce mode précédé du nom réservé *READ* (sauf dans certains cas où la propriété de protection est concernée: ces cas sont explicitement indiqués). Par conséquent, les propriétés ne seront pas explicitement définies pour les modes précédés de *READ*.

3.2 DEFINITIONS DE MODES

3.2.1 GENERALITES

syntaxe:

<définition de mode> ::= (1)
<liste de noms> = <mode définissant> (1.1)

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

syntaxe dérivée: Une *définition de mode* où la *liste de noms* comporte plus d'un *nom* est dérivée de plusieurs définitions de mode, une pour chaque nom, séparées par des virgules, avec le même *mode définissant*. Par exemple:
NEWMODE DOLLAR, POUND = INT;
est dérivé de:
NEWMODE DOLLAR = INT, POUND = INT;

sémantique: Les définitions de mode définissent un ou plusieurs noms comme étant des noms de mode, c.-à-d. des noms désignant des modes. Des définitions de mode apparaissent dans les énoncés de définition de neumodes et de synmodes. La différence entre un neumode et un synmode se trouve dans leur traitement par les algorithmes d'équivalence de mode (voir section 9.1). Toutes les propriétés héréditaires du mode définissant sont par définition transmises au nom de mode défini. Les définitions de modes peuvent être (mutuellement) récursives.

propriétés statiques: Un nom de mode est soit un des noms de mode définis dans le langage: *INT, BOOL, CHAR, PTR, INSTANCE, EVENT*, soit un nom défini dans une *définition de mode*.

Un nom de mode qui n'est pas un nom de mode prédéfini dans le langage a un unique mode définissant qui est le mode désigné par le *mode définissant* dans la *définition de mode* dans laquelle il est défini.

Un ensemble de définitions récursives est un ensemble de définitions de modes et/ou de définitions de synonymes (voir 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 ou un nom d'élément d'ensemble, 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 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 indexé 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 de façon unique 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* ou un *mode descripteur* ou un *mode procédure* à l'endoit marqué. Ce mode doit être, ou être contenu dans, le *mode définissant* du prédécesseur du *nom de mode*.

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:

```
1.12    operand_mode = INT                          (1.1)
3.3     complex = STRUCT(re,im INT)                 (1.1)
```

3.2.2 DEFINITIONS DE SYNMODES

syntaxe:

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

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

sémantique: Les énoncés de définition de synmodes définissent des noms dénotant des modes qui sont synonymes de leur mode définissant. Le traitement exact des noms définis dans une définition de synmode est expliqué a la section 9.1.

propriétés statiques: Un *nom* est dit nom de synmode si et seulement si il est défini dans une *définition de mode* dans un *énoncé de définition de synmodes*. Un nom de synmode est dit synonyme d'un mode donné (et réciproquement, le mode donné est dit synonyme du nom de synmode) si et seulement si:

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

exemples:

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

3.2.3 DEFINITIONS DE NEUMODES

syntaxe:

<énoncé de définition de neumodes> ::= (1)
NEWMODE <définition de mode> {, <définition de mode>}; (1.1)*

sémantique: Les énoncés de définition de neumodes définissent des noms signifiant des modes qui ne sont pas synonymes du mode définissant. Les valeurs définies par un neumode sont les valeurs définies par le mode définissant. Le traitement exact des noms définis dans un énoncé de définition de neumode est expliqué a la section 9.1.

propriétés statiques: Un *nom* est dit nom de neumode si et seulement si il est défini dans une *définition de mode* dans un *énoncé de définition de neumodes*.

Si le *mode définissant* est un mode intervalle, alors, avec le nom de neumode défini, un nom virtuel nouveau est introduit, noté *&parent_du_nom*, dénotant le mode parent du nom de neumode. Les valeurs définies par ce mode virtuel parent sont les valeurs du mode parent du mode intervalle définissant. Les bornes supérieure et inférieure du mode parent virtuel sont celles du mode parent du *mode intervalle* définissant.

Si le *mode définissant* est un mode chaîne, alors les nouveaux modes virtuels *&nom(i)* sont introduits pour chaque nombre *i* plus grand que la longueur de chaîne du nom de neumode. Ce

nombre *i* dénote la longueur de chaîne du mode virtuel. Les propriétés héréditaires bit ou caractère du nom de neumode sont transférées aux modes virtuels.

exemples:

```
11.4    NEWMODE line = INT(1:8);                              (1.1)
11.10   NEWMODE board = ARRAY(line) ARRAY(column) square;     (1.1)
```

3.3 CLASSIFICATION DES MODES

syntaxe:

```
<mode> ::=                                                    (1)
      <mode simple>                                           (1.1)
    | <mode composé>                                          (1.2)

<mode simple> ::=                                             (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)
```

sémantique: Les modes sont dénotés dans un programme CHILL par les productions terminales de la catégorie syntaxique *mode*. Dans la suite de ce chapitre, les propriétés spécifiques des différents modes sont définies. On définit les relations d'égalité (=) et d'inégalité (/=) sur l'ensemble des valeurs de chaque mode donné (voir section 5.3).

propriétés statiques: Un *mode* a une taille, qui est la valeur rendue par *SIZE(M)*, où *M* est un nom de synmode virtuel synonyme de *mode*.

3.4 MODES DISCRETS

3.4.1 GENERALITES

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 discrets définissent des ensembles et sous-ensembles de valeurs bien ordonnées. Tous les modes discrets qui ne sont pas des modes intervalle peuvent être des modes parents de modes intervalle (voir section 3.4.6). Tous les modes discrets définissent une borne supérieure et une borne inférieure dénotant respectivement leurs valeurs maximum et minimum.

3.4.2 *MODES ENTIER*

syntaxe:

```
<mode entier> ::=                                   (1)
    [READ] INT                                      (1.1)
  | [READ] BIN                                      (1.2)
  | [READ] <nom de mode entier>                     (1.3)
```

syntaxe dérivée: *BIN* est une syntaxe dérivée pour *INT*.

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 section 5.3.2). 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 section 11.2).

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

- La borne supérieure et la borne inférieure d'un mode entier sont les littéraux dénotant respectivement la plus grande et la plus petite valeur définies par le mode entier.
- Le nombre de valeurs d'un mode entier est défini par l'implémentation.

exemples:

```
1.4     INT                                         (1.1)
```

3.4.3 *MODES BOOLEEN*

syntaxe:

```
<mode booléen> ::=                                  (1)
    [READ] BOOL                                     (1.1)
  | [READ] <nom de mode booléen>                    (1.2)
```

sémantique: Un mode booléen définit les valeurs logiques de vérité (*TRUE* et *FALSE*) avec les opérations booléennes usuelles (voir section 5.3.2). *TRUE* est supérieur à *FALSE*.

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

- La borne supérieure d'un mode booléen est *TRUE*, la borne inférieure est *FALSE*.
- Le nombre de valeurs définies par un mode booléen est 2.

exemples:

5.4 BOOL *(1.1)*

3.4.4 *MODES CARACTERE*

syntaxe:

<mode caractère> ::= *(1)*
[READ] CHAR *(1.1)*
| *[READ] <nom de mode caractère>* *(1.2)*

sémantique: Un mode caractère définit les valeurs caractère telles qu'elles sont décrites par l'alphabet CCITT no. 5, version internationale de référence (Avis V.3, voir Appendice A1). Cet alphabet définit également l'ordre des caractères.

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

- La borne supérieure et la borne inférieure d'un mode caractère sont les littéraux de chaîne de caractères de longueur 1 dénotant respectivement la plus grande et la plus petite valeur définies par *CHAR*.
- Le nombre de valeurs définies par un mode caractère est 128.

exemples:

8.4 CHAR *(1.1)*

3.4.5 *MODES ENSEMBLE*

syntaxe:

<mode ensemble> ::= *(1)*
[READ] SET(<extension d'ensemble>) *(1.1)*
| *[READ] <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 numéro>
    {, <élément d'ensemble avec numéro>}*                     (3.1)

<élément d'ensemble avec numéro> ::=                          (4)
    <nom> = <expression littérale entière>                    (4.1)

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

<élément d'ensemble> ::=                                      (6)
    <nom>                                                     (6.1)
  | <valeur anonyme>                                          (6.2)

<valeur anonyme> ::=                                          (7)
    *                                                         (7.1)
```

sémantique: Un mode ensemble définit un ensemble de valeurs nommées ou anonymes. Les valeurs nommées sont dénotées par les noms 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 (voir ci-dessous). Cette représentation définit également l'ordre des valeurs.

propriétés statiques: Un mode ensemble a les propriétés héréditaires suivantes:

- Un mode ensemble a 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 entière (de représentation) qui est, dans le cas d'une *extension d'ensemble avec numéros*, la valeur rendue par l'*expression littérale entière* de l'*élément d'ensemble avec numéros* où apparaît le nom d'élément d'ensemble et, 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 1 et à *B* la valeur de représentation 3.

- Un mode ensemble a une borne inférieure et une borne supérieure qui sont ses noms d'élément d'ensemble dénotant respectivement les valeurs nommées la plus petite et la plus grande.

- Le nombre de valeurs d'un mode ensemble est, dans le cas d'une *extension d'ensemble avec numéros*, la plus grande des valeurs attachées aux noms d'élément d'ensemble augmentée de 1, sinon le nombre d'occurrences d'*élément d'ensemble* dans l'*extension d'ensemble sans numéros*.

- Un mode ensemble est un mode ensemble avec des trous si et seulement si le nombre d'occurrences de *nom* dans l'*extension d'ensemble* est plus petit que le nombre de valeurs du mode ensemble.

conditions statiques: Chaque *expression littérale entière* dans l'*extension d'ensemble* doit rendre une valeur entière non négative différente, en ce sens que pour une quelconque paire d'expressions *e1* et *e2*, *NUM(e1)* et *NUM(e2)* rendent des résultats différents.

Un mode ensemble doit définir au moins une valeur nommée.

exemples:

11.5	*SET(occupied, free)*	*(1.1)*
6.4	*month*	*(1.2)*

3.4.6 *MODES INTERVALLE*

syntaxe:

```
<mode intervalle> ::=                                              (1)
     [READ]<nom de mode discret>( intervalle littéral> )           (1.1)
   | [READ] RANGE( <intervalle littéral> )                         (1.2)
   | [READ] BIN( <expression littérale entière> )                  (1.3)
   | [READ] <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 : 2ⁿ-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 suivante (non-héréditaire): 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 synmode*, alors son mode parent est le mode parent du mode définissant du *nom de synmode*.

- Si le mode intervalle est un *nom de neumode*, alors son mode parent est le mode parent virtuellement introduit (voir section 3.2.3).

Un mode intervalle a les propriétés héréditaires suivantes:

- Un mode intervalle a 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(S) - NUM(I) + 1*, où *S* et *I* dénotent respectivement la borne supérieure et la borne inférieure du mode intervalle.

- Un mode intervalle est dit mode intervalle avec des trous si et seulement si son mode parent est un mode ensemble avec des trous et qu'au moins une valeur anonyme est dans l'intervalle spécifié par le mode intervalle.

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.

exemples:

```
9.4     INT(2:max)                                  (1.1)
11.11   line                                        (1.4)
```

```
9.4     2:max                                                    (2.1)
```

3.5 MODES ENSEMBLISTE

syntaxe:

```
<mode ensembliste> ::=                                           (1)
     [READ] POWERSET <mode primitif>                             (1.1)
   | [READ] <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 d'un mode 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 section 5.3).

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

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

exemples:

```
8.4     POWERSET CHAR                                            (1.1)
9.4     POWERSET INT(2:max)                                      (1.1)
9.6     number_list                                              (1.2)
```

3.6 MODES REPERE

3.6.1 GENERALITES

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 sections 4.2.3, 4.2.4 et 4.2.15), 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 REPERE LIE*

syntaxe:

<mode repère lié> ::= (1)
[READ] REF <mode repéré> (1.1)
| *[READ] <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 unique qui lui est attaché et qui est dénoté par *mode repéré*.

exemples:

10.38 REF cell (1.1)

3.6.3 *MODES REPERE LIBRE*

syntaxe:

<mode repère libre> ::= (1)
[READ] PTR (1.1)
| *[READ] <nom de mode repère libre>* (1.2)

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

exemples:

19.5 PTR (1.1)

3.6.4 MODES DESCRIPTEUR

syntaxe:

```
<mode descripteur> ::=                                              (1)
     [READ] ROW <mode chaîne>                                       (1.1)
   | [READ] ROW <mode rangée>                                       (1.2)
   | [READ] ROW <nom de mode de structure variable>                 (1.3)
   | [READ] <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 *nom de mode stucture variable*.

exemples:

```
8.6     ROW CHAR (max)                                              (1.1)
```

3.7 MODES PROCEDURE

syntaxe:

```
<mode procédure> ::=                                                (1)
     [READ] PROC( [<liste de paramètres>] )
     [<spec de résultat>]
     [EXCEPTIONS( <liste d'exceptions> )] [RECURSIVE]               (1.1)
   | [READ] <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>] [<nom de registre>]           (3.1)

<attribut de paramètre> ::=                                         (4)
     IN | OUT | INOUT | LOC                                         (4.1)
```

```
<spec de résultat> ::=                                          (5)
     [RETURNS] (<mode> [LOC] [<nom de registre>])               (5.1)

<liste d'exceptions> ::=                                        (6)
     <nom d'exception> {, <nom d'exception>}*                   (6.1)

<nom d'exception> ::=                                           (7)
     <nom>                                                      (7.1)
```

syntaxe dérivée: Une *spec de résultat* sans le nom réservé facultatif *RETURNS* est une syntaxe dérivée pour la *spec de résultat* avec *RETURNS*.

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 ou dans les énoncés de définition d'entrée. 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 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 paramètres, chaque spec de paramètre étant constituée d'un mode et, éventuellement, d'un attribut de paramètre et/ou d'un nom de registre. Les specs de paramètres sont définies par la *liste de paramètres*.

- Il a une spec de résultat facultative, constituée d'un mode, d'un attribut LOC facultatif et d'un nom de registre 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 une *spec de paramètre* ou dans la *spec de résultat* ne peut avoir la propriété de synchronisation que si *LOC* y est spécifié.

3.8 MODES EXEMPLAIRE

syntaxe:

```
<mode exemplaire> ::=                                   (1)
    [READ] INSTANCE                                     (1.1)
  | [READ] <nom de mode exemplaire>                     (1.2)
```

sémantique: Un mode exemplaire définit des valeurs qui identifient des processus de façon unique. La création d'un nouveau processus (voir sections 5.2.17 et 8.1) 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:

```
15.29  INSTANCE                                         (1.1)
```

3.9 MODES DE SYNCHRONISATION

3.9.1 GENERALITES

syntaxe:

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

sémantique: Les locus d'un mode de synchronisation donnent des moyens de synchronisation des processus et de communication entre eux (voir chapitre 8). 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 EVENEMENT*

syntaxe:

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

<longueur d'événement> ::= (2)
<expression littérale entière> (2.1)

sémantique: Les locus de mode événement donnent 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, qui sont décrites respectivement aux sections 6.15, 6.16 et 6.17.

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

Il a éventuellement une longueur d'événement qui lui est attachée et qui est la valeur rendue par *NUM(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)
[READ] BUFFER [(<longueur de tampon>)]
<mode des éléments de tampon> (1.1)
| *[READ]<nom de mode tampon>* (1.2)

<longueur de tampon> ::= (2)
<expression littérale entière> (2.1)

<mode des éléments de tampon> ::= (3)
<mode> (3.1)

Note: La syntaxe donnée ci-dessus est ambiguë à cause de la syntaxe des modes rangée. L'interprétation par défaut suivante est à appliquer: si le nom réservé *BUFFER* est immédiatement suivi d'une parenthèse ouvrante, le texte qui suit immédiatement est considéré comme le début de l'indication de la *longueur de tampon* facultative et non

comme appartenant au *mode des éléments de tampon*.

sémantique: Les locus de mode tampon donnent 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 5.2.18.

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

- Il a une longueur de tampon facultative, qui est la valeur rendue par *NUM (longueur de tampon)*.
- Il a un mode des éléments de tampon qui est le mode dénoté par 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 peut pas avoir la propriété de synchronisation.

exemples:

16.28 BUFFER(1) USER_MESSAGES (1.1)
16.32 USER_BUFFERS (1.2)

3.10 MODES COMPOSES

3.10.1 GENERALITES

syntaxe:

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

sémantique: Les locus et les valeurs composés ont respectivement des sous-locus et des sous-valeurs auxquels on peut avoir accès ou qu'on peut obtenir (voir sections 4.2.5-9, 4.2.13-14 et 5.2.6-12).

3.10.2 MODES CHAINE

syntaxe:

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

<mode chaîne paramétré> ::= (2)
[READ] <nom de mode chaîne originel>
(longueur de chaîne>) (2.1)
| *[READ] <nom de mode chaîne paramétré>* (2.2)

<nom de mode chaîne originel> ::= (3)
<nom de mode chaîne> (3.1)

<genre de chaîne> ::= (4)
CHAR (4.1)
| *BIT* (4.2)

<longueur de chaîne> ::= (5)
<expression littérale entière> (5.1)

sémantique: Un mode chaîne définit des valeurs chaîne de bits ou chaîne de caractères de longueur indiquée ou impliquée par le mode chaîne.

Les valeurs chaîne d'un mode chaîne donné sont bien ordonnées. Pour les valeurs chaîne de caractères, l'ordre est l'ordre lexicographique tel que défini par l'alphabet CCITT no. 5. Pour les valeurs chaîne de bits, l'ordre est l'ordre lexicographique tel que un bit qui est 1 est supérieur à un bit qui est 0.

L'opérateur de concaténation est défini sur les valeurs chaîne. Les opérateurs logiques usuels sont définis sur les valeurs chaîne de bits (voir section 5.3).

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

- c'est un mode chaîne de bits ou un mode chaîne de caractères selon que le *genre de chaîne* spécifie *BIT* ou *CHAR*, ou selon que le *nom de mode chaîne originel* est un mode chaîne de bits ou de caractères.
- il a une longueur de chaîne, qui est la valeur rendue par *NUM(longueur de chaîne)*.

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*.

exemples:

7.45 CHAR (20) *(1.1)*

3.10.3 *MODES RANGEE*

syntaxe:

<mode rangée> ::= *(1)*
[READ] [ARRAY] (<mode d'indice> {,<mode d'indice>})*
*<mode des éléments> {<implantation d'élément>}** *(1.1)*
| *<mode rangée paramétré>* *(1.2)*
| *[READ] <nom de mode rangée>* *(1.3)*

<mode rangée paramétré> ::= *(2)*
[READ] <nom de mode rangée originel>
(<indice supérieur>) *(2.1)*
| *[READ] <nom de mode rangée paramétré>* *(2.2)*

<nom de mode rangée originel> ::= *(3)*
<nom de mode rangée> *(3.1)*

<mode d'indice> ::= *(4)*
<mode discret> *(4.1)*
| *<intervalle littéral>* *(4.2)*

<indice supérieur> ::= *(5)*
<expression littérale> *(5.1)*

<mode des éléments> ::= *(6)*
<mode> *(6.1)*

syntaxe dérivée: Le nom réservé *ARRAY* est facultatif. Un *mode rangée* (qui n'est ni un *nom de mode rangée* ni un *mode rangée paramétré*) sans le nom réservé *ARRAY* est dérivé du mode rangée avec le nom réservé *ARRAY*.

La notation de mode d'indice *<intervalle littéral>* est dérivée du mode discret *RANGE(<intervalle littéral>)*. 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 section 3.10.6). Deux valeurs rangée sont égales si et seulement si toutes les valeurs élément se correspondant sont égales.

propriétés statiques: Un mode rangée a les propriétés héréditaires suivantes:

- Il a un mode d'indice qui est le mode discret dénoté par *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* 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.

- 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*.

- C'est un mode implanté si et seulement si une implantation d'élément est spécifiée et est un *pas*.

- Il a un nombre d'éléments qui est la valeur rendue par:
  NUM(borne supérieure) - NUM(borne inférieure) + 1.

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.

Le *mode d'indice* ne peut être un mode ensemble avec des trous ni un mode intervalle avec des trous.

exemples:

```
5.30    ARRAY(1:16) STRUCT(c4, c2, c1 BOOL)                    (1.1)
11.10   ARRAY(line) ARRAY(column) square                       (1.1)
11.15   board                                                  (1.3)
```

3.10.4 MODES STRUCTURE

syntaxe:

```
<mode structure> ::=                                           (1)
     <mode structure emboîtée>                                 (1.1)
   | <mode structure étagée>                                   (1.2)
   | <mode structure paramétré>                                (1.3)
   | [READ] <nom de mode structure>                            (1.4)

<mode structure emboîtée> ::=                                  (2)
     [READ] STRUCT (<champs> {,<champs>}*)                     (2.1)

<champs> ::=                                                   (3)
     <champs fixes>                                            (3.1)
   | <choix de champs>                                         (3.2)

<champs fixes> ::=                                             (4)
     <liste de noms> <mode> [<implantation de champ>]          (4.1)

<choix de champs> ::=                                          (5)
     CASE [<marqueurs>] OF
     <champs à choisir> {,<champs à choisir>}*
     [ELSE [<champs récurrents> {,<champs récurrents>}*]]
     ESAC                                                      (5.1)

<champs à choisir> ::=                                         (6)
     [<spécification d'étiquettes de cas>]
     : [<champs récurrents> {,<champs récurrents>}*]           (6.1)

<marqueurs> ::=                                                (7)
     <nom de champ marqueur> {,<nom de champ marqueur>}*       (7.1)

<champs récurrents> ::=                                        (8)
     <liste de noms> <mode> [<implantation de champ>]          (8.1)

<mode structure paramétré> ::=                                 (9)
     [READ] <nom de mode structure variable originel>
     (<liste d'expressions littérales>)                        (9.1)
   | [READ] <nom de mode structure paramétré>                  (9.2)

<nom de mode structure variable originel> ::=                  (10)
     <nom de mode de structure variable>                       (10.1)
```

<liste d'expressions littérales> ::= (11)
<expression littérale> {,*<expression littérale>*}* (11.1)

syntaxe dérivée: Un *mode structure étagée* est une syntaxe dérivée pour un *mode structure emboîtée*. Ceci est expliqué à la section 3.10.5.

Une occurrence de *champs fixes*, ou une occurrence de *champs récurrents*, où la *liste de noms* comporte plus d'un *nom*, est une syntaxe dérivée pour plusieurs occurrences de *champs fixes* ou de *champs récurrents*, selon le cas, chacune comportant un *nom*, le *mode* spécifié et l'*implantation de champ* facultative. Cette dernière ne peut 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 sélectionables 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 maximum de l'ensemble des champs à choisir (pire des cas).

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 (virtuels). Leur sémantique est définie à la section 3.11.4.

L'implantation d'un locus ou d'une valeur structure peut être contrôlée au moyen d'une spécification d'implantation de champs (voir section 3.10.6).

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

propriétés statiques:

généralités:

Un mode structure a les propriétés héréditaires suivantes:

- Un mode structure est un mode structure fixe si et seulement s'il est dénoté par un *mode structure emboîtée* (ou *étagée*) qui ne contient pas directement d'occurrence de *choix de champs*.

- Un mode structure est un mode structure variable si et seulement s'il est dénoté par un *mode structure emboîtée* (ou *étagée*) contenant au moins une occurrence de *choix de champs*.

- Un mode structure est un mode structure paramétré si et seulement s'il est dénoté par un *mode structure paramétré*.

- Un mode structure 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 nom de champ d'un mode structure donné a un mode de champ unique qui lui est attaché, et qui est soit *M* soit *READ M*, où *M* est le *mode* qui suit le nom de champ. Le mode de champ sera *READ M* si le *mode* qui suit le nom de champ n'est pas un mode protégé et que soit le nom de champ est un nom de champ marqueur d'un mode structure paramétré (voir ci-dessous), soit le *mode structure* est un mode protégé

  Un mode de champ d'un mode structure donné a une implantation de champ unique qui lui est attachée et qui est l'*implantation de champ* qui suit le nom de champ si elle est présente, sinon l'implantation de champ par défaut, qui est *PACK* ou *NOPACK*. Un nom de champ est repérable (par définition du langage) si et seulement si son implantation de champ est *NOPACK*.

- Un *mode structure* dénote un mode implanté si et seulement 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 la réunion de l'ensemble de noms définis par toute *liste de noms* dans les *champs fixes* avec l'ensemble des noms définis par toute *liste de noms* 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 un des *marqueurs* d'un *choix de champ*. Les *choix de champs* dans lesquels aucun *marqueur* n'est spécifié, sont des choix de champs sans marqueurs. Les noms de champ récurrent définis par toute *liste de noms* dans des *champs récurrents* d'un *choix de champs* sans marqueurs sont des noms de champ récurrent sans marqueurs. Les autres noms de champ récurrent sont des noms de champ récurrent avec marqueurs.

- Un mode structure variable est un mode structure variable sans marqueurs si et seulement 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 si et seulement 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 déterminée 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 section 9.1.3).

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*.

- C'est un mode structure paramétré avec marqueurs si et seulement si son mode structure variable originel est un mode structure variable avec marqueurs, sinon le mode structure paramétré est sans marqueurs.

- Il a un ensemble de noms de champ qui est la réunion de l'ensemble de noms de champ fixe de son mode structure variable originel avec 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 définie 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.

- A un mode structure paramétré est attachée une liste de valeurs définies par la *liste d'expressions littérales*.

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 champs 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 les *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.

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 section 9.1.3) ainsi que les mêmes exigences de complétude, cohérence et compatibilité que pour l'action de cas (voir section 6.4). Chacun des noms de champ marqueur des *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)                                        (2.1)
11.5    STRUCT(status SET(occupied, free),
              CASE status OF
               (occupied): p piece,
               (free):
               ESAC)                                             (2.1)
2.5     fraction                                                  (1.4)
11.5    status SET(occupied, free)                                (4.1)
11.6    status                                                    (7.1)
11.7    p piece                                                   (8.1)
```

3.10.5 *NOTATION ETAGEE DE STRUCTURES*

syntaxe dérivée:

```
<mode structure étagée> ::=                                       (1)
     1 [<spécification de rangée>]
     [READ] {,<champs de l'étage (2)>}*                           (1.1)

<champs de l'étage (n) ::=                                        (2)
     <champs fixes de l'étage (n)>                                (2.1)
   | <champs à choisir de l'étage (n)>                            (2.2)

<champs fixes de l'étage (n)> ::=                                 (3)
     n <liste de noms> <mode> [<implantation de champ>]           (3.1)
   | n <liste de noms> [<spécification de rangée>]
     [READ] [<implantation de champ>]
     {,champs de l'étage (n+1)>}*                                 (3.2)

<champs à choisir de l'étage (n)> ::=                             (4)
     CASE [<marqueurs>] OF
     <choix de champs de l'étage (n)>
     {,<choix de champs de l'étage(n)>}*
     [ELSE [<champs récurrents de l'étage (n)>
     {,<champs récurrents de l'étage (n)>}*]]
     ESAC                                                         (4.1)

<choix de champs de l'étage (n)> ::=                              (5)
     [<spécification d'étiquettes de cas>
     {,<spécification d'étiquettes de cas>}*]
     : [<champs récurrents de l'étage (n)>
     {,<champs récurrents de l'étage (n)>}*]                      (5.1)

<champs récurrents de l'étage (n)> ::=                            (6)
     n <liste de noms> <mode> [<implantation de champ>]           (6.1)
   | n <liste de noms> [<spécification de rangée>]
     [READ] [<implantation de champ>]
     {,<champs de l'étage (n+1)}*                                 (6.2)

<spécification de rangée> ::=                                     (7)
     [READ] [ARRAY] (<mode d'indice> {,<mode d'indice>}*)
```

*{<implantation d'élément>}** (7.1)

Note: Cette description d'une notation de numéros de niveau pour les structures comporte une extension de la méthode de description de la syntaxe expliquée au chapitre 2: la syntaxe est définie récursivement en utilisant comme paramètre le numéro de niveau *(n)*.

sémantique: Le *mode structure étagée* est une syntaxe dérivé pour un *mode structure emboîtée* unique.

La notation emboîtée est considérée comme la syntaxe stricte et toute la sémantique, les conditions et les propriétés sont expliquées en termes de cette syntaxe stricte (voir section 3.10.4).

Si une structure contient des champs qui sont eux-mêmes des structures ou des rangées de structures, une hiérarchie de sructures est formée et un numéro de niveau peut être associé à chaque champ.

Exemple:

```
SYNMODE M = STRUCT(B BOOL,
                   S ARRAY (1:10) STRUCT (T INT, U BOOL));
```

La structure tout entière est de niveau 1, *B* et *S* sont de niveau 2, *T* et *U* de niveau 3. Au lieu d'écrire des modes structure emboîtée, il est permis, dans le *mode structure étagée*, d'écrire le numéro de niveau en face du nom.

Exemple:

```
SYNMODE M = 1, 2 B BOOL,
               2 S ARRAY (1:10),
                 3 T INT,
                 3 U BOOL;
```

Dans les définitions de modes et les définitions de synonymes avec un mode, il n'y a pas de nom associé au premier niveau. L'association se produit à la déclaration ou au point de spécification de paramètres formels. A ces endroits, le nom du premier niveau sera placé après la position de niveau 1.

Exemple:

```
DCL 1 A,
      2 B BOOL,
      2 S ARRAY (1:10),
        3 T INT,
        3 U BOOL;
```

Dans les déclarations et les spécifications des paramètres formels, les attributs et les initialisations, quand il y en a, doivent être spécifiés à la fin de la position de niveau 1.

Exemple:

```
P : PROC(1 X INOUT,
         2 B BOOL,
         2 C INT);
```

Si dans un *mode structure étagée*, une rangée de structures est spécifiée, la spécification de rangée est donnée après l'indicateur de niveau.

conditions statiques: Les notations emboîtées et étagées ne peuvent être mélangées.

exemples:

```
19.9    DCL 1   BASED (P),
                2 I INFO POS(0,8:31),
                2 PREV PTR POS(1,0:15),
                2 NEXT PTR POS(1,16:31)                    (1.1)
```

3.10.6 DESCRIPTION D'IMPLANTATION POUR MODES RANGEE 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> [,<taille de patron>]])  (3.1)

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

<taille de patron> ::=                                     (5)
     <expression littérale entière>                        (5.1)

<mot> ::=                                                  (6)
     <expression littérale entière>                        (6.1)

<taille de pas> ::=                                        (7)
     <expression littérale entière>                        (7.1)

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

<bit final> ::= (9)
<expression littérale entière> (9.1)

<longueur> ::= (10)
<expression littérale entière> (10.1)

semantique: 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 de champs de 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 ausi la repérabilité (par définition du langage).

L'information *PACK*, *NOPACK* ne s'applique qu' à un niveau, c.-à-d. elle ne s'applique qu'aux éléments de la rangée 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 permis 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'un objet composé 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.

La position précise d'un composant C, c.-à-d. élément ou champ, d'un objet est donnée par les trois constantes suivantes: W_c, B_c et L_c où

W_c est la distance en mots entre le premier mot qui est (éventuellement partiellement) occupé par C et le premier mot qui est (éventuellement partiellement) occupé par l'objet dont C est un composant,

B_c est la distance en bits entre le premier bit occupé par C, et le bit le plus à gauche du premier mot qui est (éventuellement partiellement) occupé par C,

L_c est le nombre de bits que C occupe.

L'information de position, donnée pour les composants de l'objet détermine la position précise de ces composants si l'objet est entier (c.-à-d. s'il n'est pas un composant d'un autre objet). Pourtant si l'objet n'est pas entier, la position précise des composants dépendra de la position précise de l'objet lui-même.

Un *pas* spécifié pour les éléments d'une rangée est une abréviation pour une énumération explicite du *pos* de chaque élément individuel. Informellement, le *pos* et la *taille de pas* spécifient un "patron de représentation" pour les éléments qui entrent complètement dans les premiers *taille de patron* mots, en supposant que la rangée est entière. La position du premier élément est déterminée par *pos*; la position des élément suivants qui entrent complètement dans les premiers *taille de patron* mots, est telle que la distance en bits entre les premiers bits occupés par des éléments successifs est *taille de pas*. Le patron de représentation spécifié de cette manière est répété autant de fois qu'il faut pour les groupes suivants de *taille de patron* mots.

Pos

Etant donné un objet O d'un mode implanté dans lequel un *pos* de la forme:
POS(<numéro de mot>, <bit initial>, <longueur>)
est spécifié pour un composant C de cet objet, la position précise du composant C est déterminée de la façon suivante:

- Si l'objet O (dont C est un composant) est entier, alors

  W_c est *NUM(numéro de mot)*,

  B_c est *NUM(bit initial)*, et

  L_c est *NUM(longueur)*.

- Si l'objet O (dont C est un composant) n' est pas entier, alors

  W_c est dénoté par:
  NUM(numéro de mot) +
  (NUM(bit initial) + B_O*)/LARGEUR*,

B_c est dénoté par:
(NUM(bit initial) + B_0) MOD LARGEUR, et

L_c est dénoté par: *NUM(longueur)*

où *LARGEUR* est le nombre de bits dans un mot.

Pas

Soit élément$_i$ défini comme étant:

- l'élément d'indice le plus petit si i=0, sinon
- l'élément d'indice successeur de l'indice de l'élément$_n$ où n=i-1.

Soit i_0 le nombre d'éléments précédant élément$_0$ dans son patron de représentation. Etant donné un attribut *pas* de la forme:

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

le *pos* d'un élément individuel par rapport au début du patron de représentation d'élément$_0$ se détermine comme suit:

le *pos* d'élément$_i$ est:
*POS(NUM(taille de patron) * ((i + i_0)/DENS)*
+RBPOS$_i$ / LARGEUR, RBPOS$_i$ MOD LARGEUR, longueur)
pour 1 ≤ i ≤ 'nombre d'éléments'
où *RBPOS$_i$* est:
*NUM(numéro de mot) * LARGEUR + NUM(bit initial)*
*+ NUM(taille de pas) * ((i + i_0) MOD DENS)*
et *DENS* est:
*(NUM(taille de patron) * LARGEUR)/NUM(taille de pas)*
et *LARGEUR* est le nombre de bits dans un mot.

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, MTAILLE * LARGEUR)*

où *MTAILLE* est la taille du mode du composant pour lequel le *pos* est spécifié.

La notation:

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

est sémantiquement équivalente à:

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

où *PTAILLE* est le plus petit entier tel que:

*PTAILLE * LARGEUR ≥ NUM(taille de pas)*

La notation:

STEP(<pos>)

est sémantiquement équivalente à:

STEP(<pos>, LTAILLE)

où *LTAILLE* est la *<longueur>* spécifié dans le *pos*, ou déductible du *pos* par les régles ci-dessus.

propriétés statiques: Pour tout locus d'un mode rangée implanté, l'implantation d'élément du mode détermine la repérabilité (par définition du langage) de ses sous-locus (y comprises les sous-rangées et tranches de rangée) comme suit:

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

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 (par définition du langage) 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 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 et ne pas l'être sinon.

Toutes les occurrences d'*expression littérale entière* doivent donner une valeur non négative. De plus, *longueur*, *taille de pas* et *taille de patron* doivent donner une valeur non nulle, et *bit initial* et *bit final* doivent donner une valeur inférieur à *LARGEUR* où *LARGEUR* est le nombre de bits dans un mot (défini par l'implémentation). De plus, *bit initial* doit donner une valeur non supérieure à la valeur donnée par *bit final*.

Pour tout nom de champ d'un mode structure implanté, la *longueur* dans l'*implantation de champ* ne peut être inférieure au nombre minimum de bits nécessaires à représenter le champ.

Pout tout mode rangée implanté, la *longueur* dans le *pos* dans l'*implémentation d'élément* ne peut être inférieure au nombre minimum de bits nécessaires à représenter les éléments. De plus, pour toute *implantation d'élément*, les relations suivantes doivent se vérifier:

- *NUM(taille de pas) ≥ NUM(longueur)*
- *(NUM(taille de patron)*LARGEUR) MOD NUM(taille de pas) ≥ NUM(numéro de mot)*LARGEUR + NUM(bit initial)*

Cohérence et faisabilité

Cohérence:

Aucun composant d'une rangée ou 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 champs*; cependant, dans ce dernier cas, les noms de champ récurrent ne peuvent être tous deux définis dans le même *champs 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 l'implantation de composants qui ont eux-mêmes des composants repérables.

exemples:

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

3.11 MODES DYNAMIQUES

3.11.1 GENERALITES

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. Les modes dynamiques n'ont pas de notation en CHILL. 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.

3.11.2 MODES CHAINE DYNAMIQUES

dénotation virtuelle:

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

sémantique: Un mode chaîne dynamique est un mode chaîne paramétré de longueur inconnue statiquement. Le longueur de chaîne dynamique est la valeur rendue par l'*expression entière*.

propriétés statiques:

- Le mode chaîne dynamique est un mode chaîne de bits (de caractères) si et seulement si le *nom de mode chaîne originel* est un mode chaîne de bits (de caractères).

propriétés dynamiques:

- Un mode chaîne dynamique a une longueur dynamique, qui est la valeur rendue par:
  NUM(expression entière).

3.11.3 MODES RANGEE 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. La borne inférieure, le mode d'indice et le mode des éléments sont connus statiquement, la borne supérieure dynamique est la valeur rendue par l'*expression discrète*.

propriétés statiques:

- A un mode rangée dynamique sont attachés un mode d'indice, un mode des éléments, une implantation d'élément et une borne inférieure qui sont le mode d'indice, le mode des éléments, l'implantation d'élément et la borne inférieure du *nom de mode rangée originel*.

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(borne supérieure) + 1 - NUM(borne inférieure)

3.11.4 *MODES STRUCTURE PARAMETRES 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. La composition du mode structure ne peut être déterminée que dynamiquement d'après la liste de valeurs rendue par la *liste d'expressions*.

propriétés statiques:

- Un mode structure paramétré dynamique a un mode structure variable originel unique qui est le mode dénoté par *nom de mode structure variable originel*.

- Un mode structure paramétré dynamique est avec marqueurs si et seulement si son mode structure variable originel est un mode structure variable avec marqueurs, sinon il est sans marqueurs.

- L'ensemble des noms de champ (noms de champ fixe, noms de champ marqueur, noms de champ récurrent) d'un mode structure paramétré dynamique est l'ensemble des noms de champ (noms de champ fixe, noms de champ marqueur, noms de champ récurrent) 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*.

4.0 LOCUS ET LEURS ACCES

4.1 DECLARATIONS

4.1.1 GENERALITES

syntaxe:

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

<déclaration> ::=                                                (2)
     <déclaration de locus>                                      (2.1)
   | <déclaration de loc-identité>                               (2.2)
   | <déclaration de locus avec base>                            (2.3)
```

sémantique: **Un énoncé déclaratif déclare que un ou plusieurs noms sont un accès à un locus.**

exemples:

```
6.9      DCL j INT := julian_day_number,
             d, m, y INT;                                        (1.1)
6.10     d, m, y INT                                             (2.1)
11.34    starting_square LOC := b(m.lin_1)(m.col_1)              (2.2)
```

4.1.2 DECLARATIONS DE LOCUS

syntaxe:

```
<déclaration de locus> ::=                                       (1)
     <liste de noms> <mode> [STATIC] [<initialisation>]          (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 noms dans la *liste de noms*.**

Pour une *initialisation domaniale*, la *valeur* est évaluée chaque fois qu'on entre dans le domaine dans lequel la déclaration est placée (voir section 7.2.) et la valeur

obtenue est affectée au(x) locus. Avant que la *valeur* ne soit évaluée le(s) locus contient (contiennent) une valeur indéfinie (sauf si on spécifie un mode qui a la propriété de marquage et de paramètrage ou la propriété de synchronisation; voir plus loin).

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 sections 7.2 et 7.9).

Ne pas spécifier d'*initialisation* est équivalent, sémantiquement, à la spécification d'une *initialisation viagère* avec la *valeur indéfinie*. (voir section 5.3). Pourtant, si le *mode* a la propriété de marquage et de paramétrage les sous-locus champs marqueurs du locus sont initialisés avec les valeurs correspondantes des modes structure paramétrés associés.

La signification de la *valeur indéfinie* en tant qu'initialisation pour un locus de synchronisation est que les sous-locus événement et/ou tampon créés sont intialisés automatiquement à "vide" c.-à-d. aucun processus n'attend l'événement ou le tampon, et aucun message ne se trouve dans le tampon.

La sémantique de *STATIC* et de *filet* sera trouvée, respectivement à la section 7.9 et au chapitre 10.

propriétés statiques: Les noms déclarés dans une déclaration de locus sont des noms de locus. Le mode attaché au nom de locus est le *mode* spécifié dans la *déclaration de locus*. Un nom de locus est repérable (par définition du langage).

conditions statiques: La classe de la *valeur* ou *valeur constante* doit être compatible avec le *mode*.

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

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

exemples:

```
5.8     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 *DECLARATIONS DE LOC-IDENTITE*

syntaxe:

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

sémantique: Une déclaration de loc-identité crée autant d'accés au locus de mode statique spécifié qu'il y a de noms spécifiés dans la *liste de noms*.

Si le *locus de mode statique* 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 sections 7.2 et 7.9).

propriétés statiques: Les noms déclarés dans une déclaration de loc-identité sont des noms de loc-identité. Le mode qui s'attache à un nom de loc-identité est le *mode* spécifié dans la *déclaration de loc-identité*. Un nom de loc-identité est repérable (par définition du langage) si et seulement si le *locus de mode statique* spécifié est repérable (par définition du langage).

conditions statiques: Le *mode* spécifié doit être compatible en lecture avec le mode du *locus de mode statique*.

exemples:

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

4.1.4 *DECLARATIONS DE LOCUS AVEC BASE*

syntaxe:

<déclaration de locus avec base> ::= (1)
<liste de noms> <mode> BASED
[(<nom de locus repère lié ou libre>)] (1.1)

syntaxe dérivée: Une *déclaration de locus avec base* sans *nom de locus repère lié ou libre* est une syntaxe dérivée pour un énoncé de définition de synmode. Par exemple:
DCL I INT BASED;
est dérivé de:
SYNMODE I = INT;
Les noms déclarés sont des noms de synmode, synonyme du *mode* spécifié.

sémantique: Une déclaration de locus avec base (avec *nom de locus repère lié ou libre*) spécifie autant d'accès qu'il y a de noms dans la *liste de noms*. Les noms déclarés dans une déclaration de locus avec base servent comme autre manière d'accéder un locus en dérepérant une valeur repère. Cette valeur repère est contenue dans le locus spécifié par le *nom de locus repère lié ou libre*. Cette opération de dérepérage s'effectue chaque fois que et seulement quand un accés se fait via le nom basé.

propriétés statiques: Les noms déclarés dans une *déclaration de locus avec base* sont des noms basés. Le mode qui s'attache à un nom basé est le *mode* spécifié dans la *déclaration de locus avec base*. Un nom basé est repérable (par définition du langage).

conditions statiques: Si le mode du *nom de locus repère lié ou libre* est un mode repère lié, le *mode* spécifié doit être compatible en lecture avec le mode repéré du mode du *nom de locus repère lié ou libre*.

exemples:

```
19.9    1 X BASED (P),
          2 I INFO POS(0,8:31),
          2 PREV PTR POS(1,0:15),
          2 NEXT PTR POS(1,16:31)                                  (1,1)
```

4.2 LES LOCUS

4.2.1 GENERALITES

syntaxe:

```
<locus> ::=                                                        (1)
      <locus de mode statique>                                     (1.1)
    | <locus de mode dynamique>                                    (1.2)

<locus de mode statique> ::=                                       (2)
      <nom d'accès>                                                (2.1)
    | <repère lié dérepéré>                                        (2.2)
    | <repère libre dérepéré>                                      (2.3)
    | <élément de chaîne>                                          (2.4)
    | <sous-chaîne>                                                (2.5)
    | <élément de rangée>                                          (2.6)
    | <sous-rangée>                                                (2.7)
    | <champ de structure>                                         (2.8)
    | <appel de procédure rendant locus>                           (2.9)
    | <appel d'opération prédéfinie rendant locus>                 (2.10)
    | <conversion de locus>                                        (2.11)

<locus de mode dynamique> ::=                                      (3)
      <tranche de chaîne>                                          (3.1)
```

| *<tranche de rangée>* (3.2)
| *<descripteur dérepéré>* (3.3)

sémantique: Un locus est un objet qui peut contenir des valeurs. Un locus est dénoté soit par un *locus de mode statique*, c.-à-d. son mode est déterminable statiquement, soit par un *locus de mode dynamique*, c.-à-d. des parties de l'information de mode ne peuvent être obtenues que dynamiquement. Les locus doivent être accédés pour y placer ou en obtenir une valeur.

propriétés statiques: Un *locus de mode statique* a un mode statique. Seulement pour la description, un mode virtuel dynamique sera attaché à chaque *locus de mode dynamique* (voir section 3.1.). Dans le cas de locus de mode dynamique les vérifications de compatibilité exigées ne peuvent se faire complètement qu'à l'exécution. Une détection d'anomalie dans la partie dynamique de la vérification causera soit l'exception *RANGEFAIL*, soit l'exception *TAGFAIL*.

4.2.2 NOMS D'ACCES

syntaxe:

<nom d'accès> ::= (1)
<nom de locus> (1.1)
| *<nom de loc-identité>* (1.2)
| *<nom basé>* (1.3)
| *<nom d'énumération de locus>* (1.4)
| *<nom de locus faire-avec>* (1.5)

sémantique: Un nom d'accès est un accès à 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 basé, c.-à-d. un nom déclaré dans une *déclaration de locus avec base*;
- 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 marqueur, la sémantique est définie par l'implémentation.

propriétés statiques: Le mode attaché à un *nom d'accès* est respectivement le mode du *nom de locus*, du *nom de loc-identité*, du *nom basé*, du *nom d'énumération de locus*, du *nom de locus faire-avec*.

Un *nom d'accès* est repérable (par définition du langage) si et seulement si c'est un *nom de locus*, un *nom de loc-identité repérable*, un *nom basé*, un *nom d'énumération de locus repérable*, ou un *nom de locus faire-avec repérable*.

conditions dynamiques: Un *nom de loc-identité* ne peut pas dénoter un locus indéfini.

Quand on accède à un locus via un *nom basé*, les mêmes conditions dynamiques doivent être remplies qui si on dérepérait le *nom de locus repére lié ou libre* mentionné dans la *déclaration de locus avec base* associée (voir sections 4.2.3 et 4.2.4.).

Accéder à un locus via un *nom de locus faire-avec* cause l'exception *TAGFAIL* si le locus dénoté est un champ récurrent:

- d'un locus de mode structure variable avec marqueurs et que la (les) valeur(s) de(s) champ(s) marqueur(s) associé(s) indique(nt) que le champ n'existe pas;
- d'un locus de mode structure paramétré dynamique et que la liste de valeurs associée indique que le champ n'existe pas.

exemples:

```
4.11    a                                   (1.1)
11.38   starting                            (1.2)
19.14   X                                   (1.3)
15.25   EACH                                (1.4)
5.11    c1                                  (1.5)
```

4.2.3 *REPERES LIES DEREPERES*

syntaxe:

```
<repère lié dérepéré> ::=                                   (1)
    <expression repère lié> -> [<nom de mode>]              (1.1)
```

sémantique: Le locus obtenu en dérepérant une valeur repére lié est celui qui est repéré par la valeur repére 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 l'*expression repère lié*. Un *repère lié dérepéré* est repérable (par définition du langage).

conditions statiques: L'*expression 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 l'*expression repère lié*.

conditions dynamiques: La durée de vie du locus repéré ne peut pas être terminée.

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

exemples:

10.49 p-> *(1.1)*

4.2.4 REPERES LIBRES DEREPERES

syntaxe:

<repère libre dérepéré> ::= *(1)*
<expression repère libre> -> <nom de mode> *(1.1)*

sémantique: Le locus obtenu en dérepérant une valeur repère libre est celui qui est repéré par la valeur repère libre.

propriétés statiques: Le mode attaché au *repère libre dérepéré* est le *nom de mode*. Un *repère libre dérepéré* est repérable (par définition du langage).

conditions statiques: L'*expression repère libre* doit être forte.

conditions dynamiques: L'*expression repère libre* ne peut pas donner une valeur obtenue en repérant un locus non-repérable (voir section 5.2.13). La durée de vie du locus repéré ne peut pas être terminée.

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

L'exception *MODEFAIL* est causée si le *nom de mode* n'est pas compatible en lecture avec le mode du locus repéré.

4.2.5 *ELEMENTS DE CHAINE*

syntaxe:

<élément de chaîne> ::= (1)
<locus chaîne> (<position>) (1.1)

syntaxe dérivée: Un *élément de chaîne* **est une syntaxe dérivée pour une sous-chaîne de longueur 1 (voir section 4.2.6). Par exemple:**
<locus chaîne>(<position>)
est dérivé de:
<locus chaîne>(<position> UP 1)

exemples:

18.16 string->(i) (1.1)

4.2.6 *SOUS-CHAINES*

syntaxe:

<sous-chaîne> ::= (1)
<locus chaîne>
(<élément de gauche> : <élément de droite>) (1.1)
| <locus chaîne>(<position> UP <longueur de chaîne>) (1.2)

<élément de gauche> ::= (2)
<expression littérale entière> (2.1)

<élément de droite> ::= (3)
<expression littérale entière> (3.1)

<position> ::= (4)
<expression entière> (4.1)

sémantique: **Une sous-chaîne donne un locus chaîne qui est une sous-chaîne du locus chaîne spécifié.**

propriétés statiques: **Le mode attaché à une** *sous-chaîne* **est un mode chaîne paramétré, construit comme:**
&nom(longueur de chaîne)
où *&nom* **est un nom de synmode virtuel synonyme du mode (éventuellement dynamique) du** *locus chaîne*, **et où** *longueur de sous-chaîne* **est soit:** *longueur de chaîne* **soit:**
NUM(élément de droite) - NUM(élément de gauche) +1

conditions statiques: **L'***élément de gauche*, *élément de droite* **et** *longueur de chaîne* **doivent donner des valeurs entières non négatives telles que les relations suivantes se vérifient:**

1. *NUM(élément de gauche)* $\leq$ *NUM (élément de droite)*

2. *NUM(élément de droite)* $\leq$ *L-1*

3. $1 \leq$ *NUM(longueur de chaîne)* $\leq$ *L*

où *L* est la longueur de chaîne de *locus chaîne*. (Si le *locus chaîne* est un *locus de mode dynamique*, les relations 2. et 3. ne peuvent être vérifiées qu'à l'exécution; voir plus loin.)

conditions dynamiques: L'exception *RANGEFAIL* est causée si n'importe laquelle des relations 2. ou 3. ci-dessus n'est pas vérifiée dans le cas d'un *locus chaîne* de mode dynamique, ou si n'importe laquelle des relations suivantes se vérifie:

1. *NUM(position)* < *0*

2. *NUM(position)* + *NUM(longueur de chaîne)* > *L*

où *L* est la longueur de chaîne du mode du *locus chaîne*.

exemples:

18.23 string->(scanstart UP 10) *(1.2)*

4.2.7 ELEMENTS DE RANGEE

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

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 (par définition du langage) 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 n'importe laquelle des relations suivantes se vérifie:

1. *expression* < *I*

2. *expression* > *S*

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

exemples:

```
11.34    b(m.lin_1)(m.col_2)                                    (1.1)
```

4.2.8 SOUS-RANGEES

syntaxe:

```
<sous-rangée> ::=                                               (1)
     <locus rangée>
     ( <élément inférieur> : <élément supérieur> )              (1.1)
   | <locus rangée>
     ( <expression entière> UP <longueur de rangée> )           (1.2)

<élément inférieur> ::=                                         (2)
     <expression littérale>                                     (2.1)

<élément supérieur> ::=                                         (3(
     <expression littérale>                                     (3.1)

<longueur de rangée> ::=                                        (4)
     <expression littérale entière>                             (4.1)
```

sémantique: Une sous-rangée donne un locus (sous-)rangée qui est cette partie du locus rangée spécifié indiquée par *élément inférieur* et *élément supérieur*, ou par *expression entière* et *longueur de rangée*. La borne inférieure de la sous-rangée est égale à la borne inférieure de la rangée spécifiée; la borne supérieure est déterminée à partir des expressions spécifiées.

propriétés statiques: Le mode qui s'attache à une *sous-rangée* est un mode rangée paramétré défini comme suit:
&nom(indice supérieur)
où *&nom* est un nom de synmode virtuel synonyme du mode (peut être dynamique) du *locus rangée* et *indice supérieur* est soit *I + longueur de rangée - 1*, où *I* est la borne inférieure du mode rangée de *locus rangée*, soit *lit*, où *lit* est un littéral dont la classe est compatible avec celles d'*élément inférieur* et

d'*élément supérieur* et tel que:
NUM(lit)=NUM(I)+NUM(élément supérieur)-NUM(élément inférieur)
Une *sous-rangée* est repérable (par définition du langage) si l'implantation d'élément du mode du *locus rangée* est *NOPACK*.

conditions statiques: Les classes d'*élément inférieur* et d'*élément supérieur*, ou d'*expression entière* et de *longueur de rangée* doivent être compatibles avec le mode d'indice du mode du *locus rangée*.

L'*élément inférieur*, *élément supérieur* et *longueur de rangée* doivent donner des valeurs telles que les relations suivantes se vérifient:

1. $I \leq$ *élément inférieur* $\leq$ *élément supérieur*

2. $1 \leq$ *longueur de rangée*

3. *élément supérieur* $\leq S$

4. *longueur de rangée* $\leq S - I + 1$

où I et S sont respectivement la borne inférieure et la borne supérieure du mode du *locus rangée*. (Si le *locus rangée* est un *locus de mode dynamique*, les relations 3 et 4 ne peuvent se vérifier qu'à l'exécution; voir plus loin)

conditions dynamiques: L'exception *RANGEFAIL* est causée si n'importe laquelle des relations 3. ou 4. ci-dessus ne se vérifie pas pour un *locus rangée* de mode dynamique ou si n'importe laquelle des relations suivantes se vérifie:

1. $I >$ *expression entière*

2. *expression entière* + *longueur de rangée* $-1 > S$

où I et S sont respectivement la borne inférieure et la borne supérieure du mode rangée du *locus rangée*.

4.2.9 *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

Si le *locus structure* a un mode structure variable sans marqueur, 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 (par définition du langage) si le *nom de champ* est repérable (par définition du langage).

conditions statiques: Le *nom de champ* doit appartenir à l'ensemble des noms de champ du mode du *locus structure*.

conditions dynamiques: L'exception *TAGFAIL* est causée si le *locus structure* dénote:

- 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 dynamique paramétré et que la liste de valeurs associée indique que le champ n'existe pas.

exemples:

10.52 last->.info (1.1)

4.2.10 *APPELS DE PROCEDURE RENDANT LOCUS*

syntaxe:

<appel de procédure rendant locus> ::= (1)
<appel de procédure rendant locus> (1.1)

sémantique: Un locus est donné comme résultat d'un appel de procédure rendant locus.

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*.

conditions dynamiques: L'*appel de procédure rendant locus* ne peut pas donner un locus indéfini et la durée de vie du locus donné ne peut pas être terminée.

4.2.11 *APPELS D'OPERATION PREDEFINIE RENDANT LOCUS*

syntaxe:

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

sémantique: Un locus est donné comme résultat d'un appel d'opération prédéfinie par l'implémentation rendant locus.

propriétés statiques: Le mode qui s'attache à un *appel d'opération prédéfinie rendant locus* est le mode de résultat de l'*appel d'opération prédéfinie par l'implémentation rendant locus*.

conditions dynamiques: L'*appel d'opération prédéfinie par l'implémentation rendant locus* ne peut pas donner un locus indéfini et la durée de vie du locus donné ne peut pas être terminée.

4.2.12 CONVERSIONS DE LOCUS

syntaxe:

<conversion de locus> ::= (1)
<nom de mode>(*<locus de mode statique>*) (1.1)

sémantique: Une conversion de locus prend le pas sur les règles de vérification et de compatibilité de modes de CHILL. Elle attache explicitement un mode au locus de mode statique spéficié

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*.

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)

4.2.13 TRANCHES DE CHAINE

syntaxe:

<tranche de chaîne> ::= (1)
<locus chaîne>(*<début>* : *<fin>*) (1.1)

<début> ::= (2)
<expression entière> (2.1)

<fin> ::= (3)
<expression entière> (3.1)

Note: si à la fois *début* et *fin* sont une *expression littérale entière*, la construction syntaxique est ambiguë et sera interprétée comme une *sous-chaîne*.

sémantique: Une tranche de chaîne donne un locus chaîne de mode dynamique, c.-à-d. une chaîne dont la longueur est inconnue statiquement.

propriétés statiques: Le mode dynamique qui s'attache à une *tranche de chaîne* est un mode chaîne paramétré dynamique formé de la même manière que pour une *sous-chaîne* (voir section 4.2.6) mais avec un paramètre dynamique *longueur de chaîne* formé par:
NUM(fin) - NUM(début) + 1

conditions dynamiques: L'exception *RANGEFAIL* est causée si n'importe laquelle des relations suivantes se vérifie:

1. *NUM(début) > NUM(fin)*

2. *NUM(début) < 0*

3. *NUM(fin)* $\geq$ *L*

où *L* est la longueur (éventuellement dynamique) du mode chaîne du *locus chaîne*.

exemples:

18.26 blanks(count:9) *(1.1)*

4.2.14 TRANCHES DE RANGEE

syntaxe:

<tranche de rangée> ::= *(1)*
<locus rangée>(<premier> : <dernier>) *(1.1)*

<premier> ::= *(2)*
<expression> *(2.1)*

<dernier> ::= *(3)*
<expression> *(3.1)*

Note: si à la fois *premier* et *dernier* sont une *expression littérale* la construction syntaxique est ambiguë et sera interprétée comme une *sous-rangée*.

sémantique: Une tranche de rangée donne un locus rangée de mode dynamique, c.-à-d. une rangée dont la borne supérieure est inconnue statiquement.

propriétés statiques: Le mode dynamique qui s'attache à une *tranche de rangée* est un mode rangée paramétré dynamique formé de la même manière que pour une *sous-rangée* (voir section 4.2.8) mais avec un paramètre dynamique *indice supérieur*, donné par *exp*, où *exp* est une expression dont la classe est compatible avec celles de *début* et *fin* et telle que:
NUM(exp) = NUM(I) + NUM(fin) - NUM(début)
où *I* est la borne inférieure du mode du *locus rangée*.

Une *tranche de rangée* est repérable (par définition du langage) si l'implantation d'élément du mode de *locus rangée* est *NOPACK*.

conditions statiques: Les classes de *début* et de *fin* doivent être compatibles avec le mode d'indice du mode du *locus rangée*.

conditions dynamiques: L'exception *RANGEFAIL* est causée si n'importe laquelle des relations suivantes se vérifie:

1. *début > fin*

2. *début < I*

3. *fin > S*

où *I* et *S* dénotent respectivement la borne inférieure et la borne supérieure (éventuellement dynamique) du mode du *locus rangée*.

exemples:

```
17.27   res(0:count-1)                                   (1.1)
```

4.2.15 DESCRIPTEURS DEREPERES

syntaxe:

```
<descripteur dérepéré> ::=                               (1)
    <expression descripteur> ->                          (1.1)
```

sémantique: Un descripteur dérepéré donne le locus de mode dynamique qui est repéré par la valeur descripteur.

propriétés statiques: Le mode dynamique qui s'attache au *descripteur dérepéré* se construit de la manière suivante:
&nom de mode originel(<paramètre>{, <paramètre>})*
où *&nom de mode originel* est un nom de synmode virtuel synonyme du mode repéré originel du mode de l'*expression descripteur* (forte), et où les paramètres, dépendant du mode repéré originel sont:

- la longueur dynamique dans le cas d'un mode chaîne;

- la borne supérieure dynamique, dans le cas d'un mode rangée;

- la liste des valeurs associée au mode structure paramétré du locus, dans le cas d'un mode structure variable.

Un *descripteur dérepére* est repérable (par définition du langage).

conditions statiques: L'*expression descripteur* doit être forte.

conditions dynamiques: La durée de vie du locus repéré ne peut être terminée.

L'exception *EMPTY* est causée si l'*expression descripteur* donne *NULL*.

exemples:

8.10 input-> *(1.1)*

5.0 VALEURS ET LEURS OPERATIONS

5.1 DEFINITIONS DE SYNONYMES

Syntaxe:

<énoncé de définition de synonymes> ::= (1)
SYN <définition de synonyme>
{, <définition de synonyme>};* (1.1)

<définition de synonyme> ::= (2)
<liste de noms> [<mode>] = <valeur constante> (2.1)

syntaxe dérivée: Une *définition de synonyme* où la *liste de noms* comporte plusieurs noms est dérivée de plusieurs occurrences de *définition de synonyme*, une pour chaque nom, 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 comme étant une dénotation pour la valeur constante spécifiée.

propriétés statiques: Un nom défini 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 section 5.3.1).

Un nom de synonyme est littéral si et seulement si la *valeur constante* est une *expression 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 peuvent pas être récursives ni mutuellement récursives via d'autres définitions de synonyme ou définitions de mode, c.-à-d. aucun ensemble de définitions récursives ne peut contenir de définition de synonyme (voir section 3.2.1).

exemples

```
1.14    SYN neutral_for_add = 0,
            neutral_for_mult = 1;                    (1.1)
```

2.17 *neutral_for_add fraction = [0,1]* (2.1)

5.2 VALEUR PRIMITIVE

5.2.1 GENERALITES

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 sous-chaîne>* (1.6)
| *<valeur tranche de chaîne>* (1.7)
| *<valeur élément de rangée>* (1.8)
| *<valeur sous-rangée>* (1.9)
| *<valeur tranche de rangée>* (1.10)
| *<valeur champ de structure>* (1.11)
| *<locus repéré>* (1.12)
| *<conversion d'expression>* (1.13)
| *<appel de procédure rendant valeur>* (1.14)
| *<appel d'opération prédéfinie rendant valeur>* (1.15)
| *<expression démarrer>* (1.16)
| *<expression recevoir>* (1.17)
| *<opérateur nullaire>* (1.18)

sémantique: Une valeur primitive est le constituant de base d'une expression. Certaines valeurs primitives (contenus de locus de locus de mode dynamique, certains multiplets, valeurs tranche de rangée, valeurs tranche de chaîne) ont une classe dynamique, c.-à-d. une classe basée sur un mode dynamique. Pour ces valeurs primitives les vérifications de compatibilité ne peuvent s'achever qu'à l'exécution. Une détection d'anomalie résultera en 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 une *valeur primitive constante* si et seulement si c'est un *nom de valeur constant*, un *littéral*, un *multiplet constant*, un *locus repéré constant*, une *conversion d'expression constante* ou un *appel d'opération prédéfinie rendant valeur constant*.

Une *valeur primitive* est une *valeur primitive 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. Le locus est accédé pour obtenir la valeur contenue.

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 peut pas avoir la propriété de synchronisation.

conditions dynamiques: La valeur donnée ne peut pas être indéfinie (voir section 5.3.1).

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)
```

sémantique: Un nom de valeur donne une valeur.

Un nom de valeur entre dans une des catégories suivantes:

- un nom de synonyme, c.-à-d. un nom défini dans un *énoncé de définition de synonymes*;
- 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*.

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*.

Un *nom de valeur* est constant (littéral) si et seulement si c'est un *nom de synonyme* (*nom de synonyme littéral*).

conditions statiques: Le *nom de synonyme* ne peut pas être indéfini.

conditions dynamiques: Evaluer un *nom de valeur faire-avec* provoque l'exception *TAGFAIL* si la valeur dénotée est un champ récurrent:

- d'une valeur de mode structure variable avec marqueurs et que le (les) champ(s) marqueur(s) associé(s) indique(nt) que le champ dénoté n'existe pas;

- d'une valeur de mode structure paramétré dynamique et que la liste de valeurs associée indique que le champ dénoté n'existe pas.

5.2.4 *LITTERAUX*

5.2.4.1 Généralités

syntaxe:

```
<littéral> ::=                                              (1)
    <littéral d'entier>                                     (1.1)
  | <littéral de booléen>                                   (1.2)
  | <littéral d'ensemble>                                   (1.3)
  | <littéral de vide>                                      (1.4)
  | <littéral de procédure>                                 (1.5)
  | <littéral de chaîne de caractères>                      (1.6)
  | <littéral de chaîne de bits>                            (1.7)
```

sémantique: Un littéral donne une valeur constante qui est connue à la compilation.

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 d'ensemble*, un *littéral de chaîne de caractères* de longueur 1, ou un *littéral de chaîne de bits* de longueur 1.

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'] {<chifre> | _}+                                       (2.1)

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

<littéral octal d'entier> ::=                                    (4)
      O' {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _}+                   (4.1)

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

<chiffre> ::=                                                    (6)
      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9                    (6.1)

<chiffre hexadécimal> ::=                                        (7)
      <chiffre> | A | B | C | D | E | F                          (7.1)
```

sémantique: Un littéral d'entier donne une valeur entière. La notation décimale usuelle est offerte, de même que les notations binaire, octale, hexadécimale et décimale explicite. Le caractère souligné (_) n'est pas significatif, c.-à-d. il ne sert qu'à améliorer la lisibilité et 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.

conditions statiques: Ni la chaîne qui suit l'apostrophe (') ni le *littéral d'entier* tout entier ne peuvent consister seulement en caractères souligné.

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)
FALSE | TRUE (1.1)

sémantique: Un littéral de booléen donne une valeur booléenne.

propriétés statiques: La classe du *littéral de booléen* est la *BOOL*-classe par dérivation.

exemples:

5.46 FALSE (1.1)

5.2.4.4 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 (dans le contexte donné) qui a le *nom d'élément d'ensemble* spécifié comme un de ses noms d'éléments d'ensemble.

exemples:

6.51 dec (1.1)
11.89 king (1.1)

5.2.4.5 Littéral de vide

syntaxe:

<littéral de vide> ::= (1)
NULL (1.1)

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.

5.2.4.6 Littéraux de procédure

syntaxe:

<littéral de procédure> ::= (1)
<nom de procédure générale> (1.1)

sémantique: Un littéral de procédure donne une valeur procédure générale. Un littéral de procédure est un nom défini dans une définition de procédure ou dans une définition d'entrée (voir section 7.4).

propriétés statiques: La classe d'un *littéral de procédure* est la M-classe par dérivation, où M est le mode du *nom de procédure générale*.

5.2.4.7 Littéraux de chaîne de caractères

syntaxe:

<littéral de chaîne de caractères> ::= (1)
' {<caractère différent d'apostrophe>
| <apostrophe>}'* (1.1)
| C' {<chiffre hexadécimal> <chiffre hexadécimal>} '* (1.2)

<caractère> ::= (2)
<lettre> (2.1)
| <chiffre> (2.2)
| <symbole> (2.3)
| <espace> (2.4)

<lettre> ::= (3)
A | B | C | D | E | F | G | H | I | J | K | L | M (3.1)
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z (3.2)

<symbole> ::= (4)
*_ | ' | (|) | * | + | , | - | . | / | : | ; | < | = | >* (4.1)

<espace> ::= (5)
SP (5.1)

<apostrophe> ::= (6)
'' (6.1)

Note: *SP* dénote le caractère "espace"; voir Appendice A1.

sémantique: Un littéral de chaîne de caractères donne une valeur chaîne de caractères qui peut être de longueur 0. Un littéral de chaîne de caractères de longueur 1 peut servir de valeur caractère. Pour représenter le caractère apostrophe (') dans un littéral de chaîne de caractères, il doit s'écrire deux fois (''). Les caractères mentionnés ci-dessus constituent

l'ensemble minimal de caractères imprimables qui doit être fourni. Une implémentation peut permettre la présence de n'importe lequel des caractères imprimables de l'alphabet CCITT no. 5 dans un littéral de chaîne de caractères (voir Appendice A1). En plus de la représentation imprimable, on peut employer la représentation hexadécimale. Chaque paire de chiffres hexadécimaux dénote cette valeur caractère dont la représentation est le nombre hexadécimal donné (voir appendice A1).

propriétés statiques: La longueur d'un *littéral de chaîne de caractères* est soit le nombre d'occurrences de *caractère différent d'apostrophe* et d'*apostrophe*, soit le nombre de paires de *chiffres hexadécimaux*.

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

exemples:

```
8.18    'A-B<ZAA9K'''                    (1.1)
8.18    ''                               (6.1)
```

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' { 0 | 1 | _ }*'                                   (2.1)

<littéral octal de chaîne de bits> ::=                    (3)
     O' {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _ }*'           (3.1)

<littéral hexadécimal de chaîne de bits> ::=              (4)
     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. il ne sert qu'à améliorer la lisibilité et n'influence pas la valeur indiquée.

propriétés statiques: La longueur 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 *0*, *1*, *2*, *3*, *4*, *5*, *6* et *7* après *O'*, soit quatre fois le nombre d'occurrences de *chiffre*

hexadécimal aprés *H'*.

La classe d'un *littéral de chaîne de bits* est la *BIT(n)*-classe par dérivation, où *n* est la longueur du *littéral de chaîne de bits*.

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 ou sont les valeurs données par les expressions de l'intervalle. Si la deuxième expression donne une valeur inférieure à la valeur donnée par la première expression, l'intervalle est vide, c.-à-d. 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 (voir section 9.1.4).

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 le même ordre que ceux-ci sont spécifiés dans le mode structure attaché. Dans un multiplet de structure avec noms de champ, les valeurs sont données pour ces champs dont les noms 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 suivant la liste:

- si le *multiplet* est la *valeur* ou *valeur constante* dans 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 dans une *action d'affectation simple*, alors M est le *mode* (éventuellement dynamique) du *locus* partie gauche;

- si le *multiplet* est la *valeur constante* dans une *définition de synonyme* avec un *mode* spécifié, alors M est ce *mode*;

- si le *multiplet* est un *paramétre effectif* dans un *appel de procédure*, alors M est le mode de la spec de paramètre correspondante;

- si le *multiplet* est la *valeur* dans une *action revenir* ou une *action résulter*, alors M est le mode résultat de nom de procédure de l'*action résulter* ou de *l'action revenir* (voir 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.

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. Dépendant de ce que un *multiplet ensembliste*, *multiplet de rangée* ou *multiplet de structure* est spécifié, les règles de compatibilités suivantes doivent êtres 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. La valeur donnée par chaque *expression* doit être une des valeurs définies par ce mode primitif.

b. *multiplet de rangée*

1. Le mode de *multiplet* doit être un mode rangée.

2. La classe de chaque *valeur* doit être compatible avec le mode des éléments du mode du *multiplet*.

Dans le cas d'un *multiplet de rangée sans indices*:

3. Il faut autant d'occurrences de *valeur* que d'éléments dans le mode rangée du *multiplet*.

Dans le cas d'un *multiplet de rangée avec indices*:

4. Les conditions de sélection de cas doivent être remplies pour la liste d'occurrences de *liste d'étiquettes de cas* (voir section 9.1.3) La classe résultante doit être compatible avec le mode d'indice du mode du *multiplet*.

5. La valeur donnée par chaque *expression littérale* dans chaque *liste d'étiquettes de cas* et les valeurs définies par chaque *nom de mode* dans chaque *liste d'étiquettes de cas* doivent être comprises entre la borne inférieure et la borne supérieure (bornes comprises) du mode du *multiplet*.

6. Dans un *multiplet de rangée sans indice*, au moins une occurrence de *valeur* doit être une *expression*; dans un *multiplet de rangée avec indice* au moins une occurrence de *valeur* qui suit une *liste d'étiquettes de cas* qui n'est pas *(ELSE)* doit être une *expression* (voir section 5.3.1).

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 comprise entre les bornes du mode des éléments (bornes comprises), 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 peut pas être un mode structure qui a des noms de champ invisibles (voir section 9.2.7).

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 de 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*.

  6. Il doit y avoir autant d'occurrences de *valeur* qu'il y a de noms de champ indiqués existants par les valeurs données par les occurrences d'*expression littérale* 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 et seulement une 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 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 avec marqueurs, alors:

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

  12. Seuls les noms de champ correspondant à des champs indiqués comme existants par les valeurs données pour les occurrences d'*expression littérale* 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 des noms de champ*, et qui sont définis dans le même *choix de champs* doivent être tous définis dans le même *champs à choisir* ou aprés *ELSE*. Tous les noms de champ d'un choix de champs sélectionné ou définis aprés *ELSE*, doivent être mentionnés une et une seule 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* 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 résidant entre les bornes du mode du champ (bornes incluses), sauf si c'est une *valeur indéfinie*.

18. Au moins une occurrence de *valeur* doit être une *expression*.

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 section 6.2.3) s'appliquent (voir conditions a2, a3, b2, c4, c7, c10, c13 et c15).

Si le *multiplet* a un mode rangée dynamique, l'exception *RANGEFAIL* est causée si n'importe laquelle des conditions b3 ou b5 n'est pas respectée.

Si le *multiplet* a un mode structure paramétré dynamique, l'exception *TAGFAIL* est causée si la condition c16 n'est pas respectée.

exemples:

9.5	*number_list[]*	(1.1)
9.6	*[2:max]*	(2.1)
8.24	*[('A'):3,('B','K','Z'):1,(ELSE):0]*	(6.1)
17.5	*[(*):' ']*	(6.1)
12.28	*(:NULL,NULL,536:)*	(7.1)
11.16	*[.status:occupied,.p:[white,rook]]*	(9.1)

5.2.6 *VALEURS ELEMENT DE CHAINE*

syntaxe:

<valeur élément de chaîne> ::= (1)
<expression chaîne>(<position>) (1.1)

syntaxe dérivée: Une *valeur élément de chaîne* est une syntaxe dérivée pour une *valeur sous-chaîne* de longueur 1 (voir section 5.2.7). C.-à-d.
<expression chaîne>(<position>)
est dérivé de:
<expression chaîne> (position> UP 1)

5.2.7 *VALEURS SOUS-CHAINE*

syntaxe:

<valeur sous-chaîne> ::= (1)
<expression chaîne>
(<élément de gauche> : <élément de droite>) (1.1)
| *<expression chaîne>*
(<position> UP <longueur de chaîne>) (1.2)

Note: Si l'*expression chaîne* est un *locus chaîne*, la construction syntaxique est ambiguë et sera interprétée comme une *sous-chaîne* (voir section 4.2.6).

sémantique: Une valeur sous-chaîne donne une valeur chaîne qui est une sous-valeur de la valeur chaîne spécifiée.

propriétés statiques: La classe d'une *valeur sous-chaîne* est, si l'*expression chaîne* n'est pas **forte** alors la *CHAR(n)*-classe par dérivation ou *BIT(n)*-classe par dérivation (dépendant du fait que l'*expression chaîne* est une *expression chaîne* **de bits** ou **de caractères**) sinon la *&nom(n)*-classe par valeur, où *n* est soit: *<longueur de chaîne>* soit:
NUM(élément de droite)-NUM(élémentde gauche)+1
où *&nom* est un nom **de synmode** virtuel, synonyme du mode de l'*expression chaîne*.

conditions statiques: L'*élément de gauche*, *élément de droite* et *longueur de chaîne* doivent donner des valeurs entières non négatives telles que les relations suivantes se vérifient:

1. *NUM(élément de gauche)* $\leq$ *NUM(élément de droite)*

2. *NUM(élément de droite)* $\leq$ *L-1*

3. *1* $\leq$ *NUM(longueur de chaîne)* $\leq$ *L*

où *L* est la **longueur de chaîne** du mode racine de la classe de l'*expression chaîne*. (Si l'expression *chaîne* a une classe dynamique, les relations 2. et 3. ne peuvent se vérifier qu'à l'exécution; voir plus loin.)

conditions dynamiques: La valeur donnée par *valeur sous-chaîne* ne peut être **indéfinie**. L'exception *RANGEFAIL* est causée si n'importe laquelle des relations 2. ou 3. ci-dessus ne se vérifie pas dans le cas d'une *expression chaîne* de classe dynamique, ou si n'importe laquelle des relations suivantes se vérifie:

1. *NUM(position)* $<$ *0*

2. *NUM(position)* + *NUM(longueur de chaîne)* $>$ *L*

où *L* est la longueur de chaîne du mode racine de la classe de l'*expression chaîne*.

5.2.8 VALEURS TRANCHE DE CHAINE

syntaxe:

<valeur tranche de chaîne> ::= (1)
<expression chaîne>(<début> : <fin>) (1.1)

Note: Si l'*expression chaîne* est un *locus chaîne* la construction syntaxique est ambiguë et sera interprétée comme une *tranche de chaîne* (voir section 4.2.13). Si à la fois

début et *fin* sont une *expression entière littérale*, la construction syntaxique est ambiguë et sera interpétée comme une *valeur sous-chaîne* (voir section 5.2.7).

sémantique: Une valeur tranche de chaîne donne une valeur chaîne dynamique qui est une sous-valeur de la valeur chaîne spécifiée.

propriétés statiques: La classe d'une *valeur tranche de chaîne* est définie de la même manière que pour une valeur sous-chaîne (voir section 5.2.7.), mais avec un paramètre dynamique *n* qui est:
NUM(fin) - NUM(début) + 1

conditions dynamiques: La valeur donnée par une *valeur tranche de chaîne* ne peut pas être indéfinie.

L'exception *RANGEFAIL* est causée si n'importe laquelle des relations suivantes se vérifie:

1. *NUM(début) > NUM(fin)*

2. *NUM(début) < 0*

3. *NUM(fin) > L*

où *L* est la longueur (qui peut être dynamique) du mode racine de la classe de l'*expression chaîne*.

5.2.9 *VALEURS ELEMENT DE RANGEE*

syntaxe:

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

Note: Si l'*expression rangée* est un *locus rangée*, la construction syntaxique est ambiguë et sera interprétée comme un *élément de rangée* (voir section 4.2.7).

syntaxe dérivée: Voir section 4.2.7.

sémantique: Une valeur élément de rangée donne une valeur qui est un élément de la valeur rangée spécifiée.

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 l'*expression rangée*.

conditions statiques: L'*expression rangée* doit être forte. La classe de l'*expression* doit être compatible avec le mode d'indice du mode de l'*expression rangée*.

conditions dynamiques: La valeur donnée par une *valeur élément de rangée* ne peut pas être indéfinie.

L'exception *RANGEFAIL* est causée si n'importe laquelle des relations suivantes se vérifie:

1. *expression* < *I*

2. *expression* > *S*

où *I* et *S* sont respectivement la borne inférieure et la borne supérieure (éventuellement dynamique) du mode de l'*expression rangée*.

5.2.10 *VALEURS SOUS-RANGEE*

syntaxe:

<valeur sous-rangée> ::= (1)
<expression rangée>
(<élément inférieur> : <élément supérieur>) (1.1)
| *<expression rangée>*
(<expression entière> UP <longueur de rangée>) (1.2)

Note: Si l'*expression rangée* est un *locus rangée* la construction syntaxique est ambiguë et sera interprétée comme une *sous-rangée* (voir section 4.2.8).

sémantique: Une valeur sous-rangée donne une (sous)-valeur rangée qui est une partie de la valeur rangée spécifiée. La borne inférieure de la valeur sous-rangée est égale à la borne inférieure de la valeur rangée spécifiée; la borne supérieure est déterminée à partir des expressions (indices) spécifiées.

propriétés statiques: La classe d'une *valeur sous-rangée* est la M-classe par valeur, où M est un mode rangée paramétré défini comme:
&nom(indice supérieur)
où *&nom* est un nom de synmode virtuel synonyme du mode (éventuellement dynamique) de l'*expression rangée* et où *indice supérieur* est soit *I + longueur de rangée -1*, où *I* est la borne inférieure du mode de l'*expression rangée*, soit *lit*, où *lit* est le littéral dont la classe est compatible avec celles de *élément inférieur* et *élément supérieur* et tel que:
NUM(lit)=
NUM(I)+NUM(élément supérieur)-NUM(élément inférieur)

conditions statiques: L'*expression rangée* doit être forte. Les classes de *élément inférieur* et *élément supérieur* ou de *expression entière* et *longueur de rangée* doivent être compatibles avec le mode d'indice du mode de l'*expression rangée*. Les *élément inférieur*, *élément supérieur* et *longueur de rangée* doivent donner des valeurs telles que les relations suivantes se vérifient:

1. $I \leq$ *élément inférieur* $\leq$ *élément supérieur*

2. $1 \leq$ *longueur de rangée*

3. *élément supérieur* $< S$

4. *longueur de rangée* $\leq S - I + 1$

où I et S sont respectivement la borne inférieure et la borne supérieure de mode rangée de l'*expression rangée*. (Si l'*expression rangée* a une classe dynamique, les relations 3. et 4. ne peuvent être vérifiées qu'à l'exécution; voir plus loin.)

conditions dynamiques: La valeur donnée par une *valeur sous-rangée* ne peut pas être indéfinie.

L'exception *RANGEFAIL* est causée si n'importe laquelle des relations 3. ou 4. ci-dessus ne se vérifie pas dans le cas d'une classe dynamique, ou si n'importe laquelle des relations suivantes se vérifie:

1. $I >$ *expression entière*

2. *expression entière* $+$ *longueur de rangée* $- 1 > S$

où I et S sont respectivement la borne inférieure et la borne supérieure du mode de l'*expression rangée*.

5.2.11 *VALEURS TRANCHE DE RANGEE*

syntaxe:

⟨*valeur tranche de rangée*⟩ ::= (1)
⟨*expression rangée*⟩(⟨*premier*⟩ : ⟨*dernier*⟩) (1.1)

Note: Si l'*expression rangée* est un *locus rangée*, la construction syntaxique est ambiguë et sera interprétée comme une *tranche de rangée* (voir section 4.2.14). Si à la fois *premier* et *dernier* sont une *expression littérale*, la construction syntaxique est ambiguë et sera interprétée comme une *valeur sous-rangée* (voir section 5.2.10).

sémantique: Une valeur tranche de rangée donne une valeur rangée dynamique, qui est une sous-valeur de la valeur rangée spécifiée.

propriétés statiques: La classe d'une *valeur tranche de rangée* est la M-classe par valeur, où M est un mode rangée paramétré dynamique défini de la même manière que pour une *valeur sous-rangée* (voir section 5.2.10) mais avec un paramètre dynamique *borne supérieure* formé comme *exp*, où *exp* est une expression dont la classe est compatible avec celles de *premier* et *dernier* et telle que:
NUM(exp) = NUM(I) + NUM(dernier) - NUM(premier)

conditions statiques: L'*expression rangée* doit être forte. Les classes de *premier* et *dernier* doivent être compatibles avec le mode d'indice du mode de l'*expression rangée*.

conditions dynamiques: La valeur donnée par une valeur *tranche de rangée* ne peut pas être indéfinie.

L'exception *RANGEFAIL* est causée si n'importe laquelle des relations suivantes se vérifie:

1. *premier > dernier*

2. *premier < I*

3. *dernier > S*

où *I* et *S* dénotent respectivement la borne inférieure et la borne supérieure (peut être dynamique) du mode de l'*expression rangée*.

5.2.12 *VALEURS CHAMP DE STRUCTURE*

syntaxe:

<valeur champ de structure> ::= (1)
<expression structure> . <nom de champ> (1.1)

Note: Si l'*expression structure* est un *locus structure* la construction syntaxique est ambiguë et sera interprétée comme un *champ de structure* (voir section 4.2.9).

sémantique: Une valeur champ de structure donne une valeur qui est un champ de la valeur structure spécifiée. Si l'*expression 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: L'*expression structure* doit être forte. Le *nom de champ* doit appartenir à l'ensemble des noms de champ du mode de l'*expression structure*.

conditions dynamiques: La valeur donnée par une *valeur champ de structure* ne peut pas être indéfinie.

L'exception *TAGFAIL* est causée si l'*expression structure* a

- 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.

exemples:

16.49 (RECEIVE USER_BUFFER).ALLOCATOR *(1.1)*

5.2.13 *LOCUS REPERES*

syntaxe:

<locus repéré> ::= *(1)*
-> <locus> *(1.1)*

sémantique: Un locus repéré donne un repère vers le locus spécifié si le locus est repérable. Si le locus n'est pas repérable, il donne une valeur repère qui ne peut pas être dérepérée (voir section 4.2.4) et qui peut repérer un locus défini par l'implémentation.

propriétés statiques: Si le *locus* est repérable, la classe du *locus repéré* est la M-classe par repère, où M est le mode du *locus*. Sinon, la classe du *locus repéré* est la *PTR*-classe par dérivation. Un *locus repéré* est constant si et seulement si le *locus* est statique.

exemples:

8.23 ->c *(1.1)*

5.2.14 *CONVERSIONS D'EXPRESSION*

syntaxe:

<conversion d'expression> ::= *(1)*
<nom de mode>(<expression>) *(1.1)*

sémantique: Une conversion d'expression prend le pas sur les règles de vérification de mode et de compatibilité de CHILL. Un mode est attaché explicitement à l'expression. La sémantique dynamique précise d'une conversion d'expression est définie par l'implémentation et dépend des représentations internes des valeurs.

propriétés statiques: La classe d'une *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: L'*expression* ne peut avoir une classe dynamique. La classe de l'*expression* doit être compatible avec au moins un mode dont la taille est égale à la taille du *nom de mode*. Le *nom de mode* ne peut pas avoir la propriété de synchronisation.

5.2.15 *APPELS DE PROCEDURE 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 peut donner une valeur indéfinie (voir sections 5.3.1 et 6.8).

exemples:

6.51 *julian_day_number([10,dec,1979])* (1.1)
11.68 *ok_bishop(b,m)* (1.1)

5.2.16 *APPELS D'OPERATION PREDEFINIE RENDANT VALEUR*

syntaxe:

<appel d'opération prédéfinie rendant valeur> ::= (1)
<appel d'opération prédéfinie par l'implémentation rendant valeur> (1.1)
| *<appel d'opération prédéfinie par CHILL rendant valeur>* (1.2)

<appel d'opération prédéfinie par CHILL rendant valeur> ::= (2)
NUM(*<expression discrète>*) (2.1)

```
| PRED( <expression discrète> )                                  (2.2)
| PRED( <expression repère lié )                                 (2.3)
| SUCC( <expression discrète> )                                  (2.4)
| SUCC( <expression repère lié> )                                (2.5)
| ABS( <expression entière> )                                    (2.6)
| ADDR( <locus> )                                                (2.7)
| CARD( <expression ensembliste> )                               (2.8)
| MAX( <expression ensembliste> )                                (2.9)
| MIN( <expression ensembliste> )                                (2.10)
| SIZE( {<nom de mode> | <locus de mode statique>} )             (2.11)
| UPPER( {<expression rangée> | <expression chaîne>} )           (2.12)
| GETSTACK( <argument pour getstack> )                           (2.13)

<argument pour getstack> ::=                                     (3)
    <nom de mode>                                                (3.1)
  | <nom de mode rangée>( <expression> )                         (3.2)
  | <nom de mode chaîne>( <expression entière> )                 (3.3)
  | <nom de mode structure variable>
    ( <liste d'expressions> )                                    (3.4)
```

syntaxe dérivée: *ADDR(<locus>)* **est une syntaxe dérivée pour** *->(<locus>)*

sémantique: Un appel d'opération prédéfinie rendant valeur est soit un appel d'opération prédéfinie par l'implémentation rendant valeur, soit un appel d'opération prédéfinie par CHILL rendant valeur. Un appel d'opération prédéfinie par CHILL rendant valeur est une invocation à une des opérations prédéfinies par CHILL et donnant une valeur.

NUM donne une valeur entière qui a la même représentation interne que la valeur donnée par l'argument discret. *NUM* appliqué à des valeurs ensemble donne la valeur entière spécifiée par le mode ensemble. *NUM* appliqué à des valeurs caractère donne la valeur entière spécifiée par l'alphabet CCITT no. 5 (voir appendice A1). *NUM(TRUE)* donne *1* et *NUM(FALSE)* donne *0*. *NUM* appliqué à une valeur entière donne cette valeur entière.

PRED et *SUCC* appliqués à des valeurs discrètes donnent respectivement la valeur précédant ou suivant immédiatement, si elle existe. Sinon une exception est causée. Si la valeur discrète est une valeur ensemble d'un mode ensemble avec trous, les trous sont passés (c.-à-d. dans l'exemple de la section propriétés statiques de 3.4.5., *SUCC(A)* donne *B* et *PRED(B)* donne *A*.

PRED et *SUCC* appliqués à des valeurs repère lié ne sont définis que pour les valeurs repère lié qui repèrent des éléments de rangée. Ils donnent respectivement la valeur repère repérant l'élément de rangée qui a l'indice qui précède ou suit immédiatement, s'il existe.

ABS est défini pour les valeurs entières, donnant la valeur absolue de la valeur entière.

ADDR est une autre notation pour repérer un locus.

CARD, *MAX* et *MIN* sont définis pour des valeurs ensemblistes. *CARD* donne le nombre de valeurs primitives dans la valeur ensembliste. *MAX* et *MIN* donnent respectivement la plus grande et la plus petite des valeurs primitives dans la valeur ensembliste.

SIZE est défini pour les locus de mode statique repérables et pour les modes. 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 adressable qu'un locus repérable de ce mode occuperait. Dans le premier cas, le locus de mode statique ne sera pas évalué à l'exécution.

UPPER est défini pour des valeurs rangée (éventuellement dynamiques) et des valeurs chaîne, donnant l'indice supérieur de la valeur rangée ou l'indice de chaîne supérieur de la valeur chaîne. (c.-à-d. la longueur de chaîne moins 1).

GETSTACK crée sur la pile un locus de mode spécifié (voir section 7.4) et donne une valeur repère pour le locus créé. Si un nom de mode est spécifié, un locus de mode statique de ce mode est créé et une valeur repère lié est donnée. Sinon un locus de mode dynamique est créé, dont le mode est un mode paramétré avec des paramètres connus à l'exécution, comme spécifié par l'argument de *GETSTACK*, et une valeur descripteur repérant le locus est donnée.

propriétés statiques: La classe d'un appel à l'opération prédéfinie *NUM* est la *INT*-classe par dérivation. L'appel à l'opération prédéfinie est constant (littéral) si et seulement si l'argument est constant (littéral).

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

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

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

La classe d'un appel à l'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 à l'opération prédéfinie est constant si et seulement si l'argument est constant.

La classe d'un appel à l'opération prédéfinie *SIZE* est la INT-classe par dérivation. L'appel à L'opération prédéfinie est constant.

Si l'argument d'un appel à l'opération prédéfinie *UPPER* est une *expression rangée*, la classe d'un appel à l'opération prédéfinie *UPPER* est la M-classe par valeur, où M est le mode d'indice du mode rangée de l'*expression rangée* (forte). Si l'argument d'un appel à l'opération prédéfinie *UPPER* est une *expression chaîne*, la classe de l'appel à l'opération prédéfinie est la *INT*-classe par dérivation. Un appel à l'opération prédéfinie *UPPER* est constant et littéral si et seulement si la classe de l'*expression rangée* ou *expression chaîne* est une classe statique.

La classe de l'appel à l'opération prédéfinie *GETSTACK* est la M-classe par repère, où M est, suivant l'*argument de getstack*, soit le *nom de mode*, soit un mode paramétré dynamique formé par:
&<nom de mode rangée>(<expression>), ou
&<nom de mode chaîne>(<expression entière>), ou
&nom de mode structure variable >(<liste d'expressions>).

conditions statiques: Si l'argument d'un appel à l'opération prédéfinie *PRED* ou *SUCC* est constant, il ne peut donner respectivement la plus petite ou la plus grande valeur discrète définie par le mode racine de la classe de l'argument.

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

L'*expression ensembliste* comme argument de *CARD*, *MAX* ou *MIN* doit être forte.

L'*expression repère lié* comme argument de l'appel à l'opération prédéfinie *PRED* ou *SUCC* doit être forte.

L'*expression rangée* comme argument de l'appel à l'opération prédéfinie *UPPER* doit être forte.

Les conditions de compatibilité suivantes doivent être remplies pour un *argument de getstack* 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*.

- 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 du *nom de mode structure variable*.

conditions dynamiques: *PRED* et *SUCC* causent l'exception *OVERFLOW* s'ils sont appliqués à la plus petite ou plus grande valeur discrète définie par le mode racine de la classe de leur argument. *PRED* et *SUCC* causent l'exception *RANGEFAIL* s'ils sont appliqués à une valeur repère lié repérant un élément de rangée qui a le plus petit ou le plus grand indice. *PRED* et *SUCC* causent l'exception *EMPTY* si l'*expression repère lié* donne *NULL*.

MAX et *MIN* causent l'exception *EMPTY* s'ils sont appliqués à des valeurs ensembliste vides (c.-à-d. ne contenant aucune valeur primitive).

GETSTACK cause l'exception *SPACEFAIL* si les requêtes de mémoire ne peuvent être satisfaites.

GETSTACK cause l'exception *RANGEFAIL* si, dans l'*argument de getstack*:

- 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 égale ou supérieure à la longueur du *nom de mode chaîne*;
- une expression dans la *liste d'expressions* pour laquelle la classe correspondante dans 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.

ABS cause l'exception *OVERFLOW* si la valeur donnée est en dehors des bornes définies par le mode racine de la classe de l'argument.

exemples:

9.11	*MIN(sieve)*	*(2.10)*
11.91	*PRED(col_1)*	*(2.2)*
11.91	*SUCC(col_1)*	*(2.4)*

5.2.17 *EXPRESSIONS DEMARRER*

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 chapitre 8). 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. L'expression démarrer donne une valeur exemplaire univoque identifiant le processus créé.

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 peut 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 pour les n premiers paramètres effectifs sont les mêmes que pour le passage de paramètres à des procédures (voir section 6.7).

conditions dynamiques: L'*expression démarrer* peut causer n'importe laquelle des exceptions définies par l'implémentation et dont le nom est attaché au *nom de processus* (voir section 7.5).

Pour le passage de paramètres, les conditions d'affectation de n'importe lequel des paramètres effectifs en tenant compte du mode du paramètre formel associé s'appliquent (voir section 6.7).

L'*expression démarrer* cause l'exception *SPACEFAIL* si les requêtes de mémoire ne peuvent être satisfaites.

examples:

15.25 START COUNTER() (1.1)

5.2.18 *EXPRESSIONS RECEVOIR*

syntaxe:

<expression recevoir> ::= (1)
RECEIVE <locus tampon> (1.1)

sémantique: L'expression recevoir donne une valeur qui sort du tampon spécifié d'un des processus envoyants en attente. Si l'expression recevoir est exécutée pendant que le tampon ne contient pas de valeur ou qu'aucun processus envoyant n'est en attente pour le tampon, le processus exécutant est mis en attente jusqu'à ce qu'une valeur soit envoyée au tampon (voir le chapitre 8 pour tous les détails).

propriétés statiques: 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*.

conditions dynamiques: La durée de vie du locus tampon dénoté ne peut pas se terminer pendant que le processus exécutant est en attente pour ce locus tampon.

exemples:

16.49 RECEIVE USER_BUFFER *(1.1)*

5.2.19 *OPERATEUR NULLAIRE*

syntaxe:

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

sémantique: L'opérateur nullaire donne la valeur exemplaire unique qui identifie le processus qui l'exécute.

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

5.3 *VALEURS ET EXPRESSIONS*

5.3.1 *GENERALITES*

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 soit une valeur indéfinie ou une valeur (définie par CHILL) donnée comme le résultat de l'évaluation d'une expression.

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.

popriétés dynamiques: Une valeur est dite être 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.

(Note: Une *valeur* ne peut dénoter une valeur indéfinie que dans les contextes suivants:

- c'est une *valeur indéfinie*;
- c'est un *contenu de locus* qui contient une valeur indéfinie;
- c'est un *appel de procédure rendant valeur*, donnant une valeur indéfinie;
- c'est une *valeur sous-chaîne*, une *valeur tranche de chaîne*, une *valeur sous-rangée*, une *valeur tranche de rangée* ou une *valeur champ de structure* donnant une valeur indéfinie.)

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-1>                                      (1.1)
  | <sous-expression> { OR | XOR } <opérande-1>        (1.2)

<sous-expression> ::=                                 (2)
    <expression>                                      (2.1)
```

sémantique: L'ordre d'évaluation des constituants d'une expression, de ses sous-constituants, etc. est indéfini et ils peuvent être considérés comme étant évalués en ordre mélangé. Il n'est nécessaire de les évaluer que jusqu'au point où la valeur à donner est déterminée uniquement. Si l'expression est constante ou littérale, l'évaluation ne causera jamais d'exception.

Si *OR* ou *XOR* est spécifié la *sous-expression* et *l'opérande -1* donnent:

- des valeurs booléennes, auquel cas *OR* et *XOR* dénotent les opérateurs logiques usuels donnant une valeur booléenne;
- des valeurs chaîne de bits, auquel cas *OR* et *XOR* dénotent les opérations logiques usuelles sur les chaînes de bits, donnant une valeur chaîne de bits;
- des valeurs ensembliste, auquel cas *OR* dénote l'union des deux valeurs ensembliste et *XOR* dénote la valeur ensembliste consistant en les valeurs primitives qui ne sont que dans une des valeurs ensembliste spécifiées (c.-à-d. *A XOR B = A-B OR B-A*).

propriétés statiques: Si l'*expression* est un *opérande-1*, la classe de l'*expression* est la classe de l'*opérande-1*. Si *OR* ou *XOR* est spécifié, la classe de l'*expression* est la classe résultante de la classe de *sous-expression* et de *opérande-1*.

Une *expression* est constante (littérale) si et seulement si elle est soit un *opérande-1* qui est constant (littéral), soit construite d'une *expression* et un *opérande-1* qui sont tous les deux constants (littéraux).

conditions statiques: Si *OR* ou *XOR* est spécifié, la classe de la *sous-expression* doit être compatible avec la classe de l'*opérande-1*. Les deux classes doivent avoir un mode racine booléen, chaîne de bit ou ensembliste.

conditions dynamiques: Dans le cas de *OR* ou *XOR* une exception *RANGEFAIL* est causée si l'un ou les deux opérandes ont une classe dynamique et que la partie dynamique de la vérification de compatibilité mentionnée ci-dessus échoue.

exemples:

```
10.27   i<min                          (1.1)
10.27   i<min OR i>max                 (1.2)
```

5.3.3 *OPERANDE-1*

syntaxe:

```
<opérande-1> ::=                                              (1)
     <opérande-2>                                             (1.1)
   | <sous-opérande-1> AND <opérande-2>                       (1.2)

<sous-opérande-1> ::=                                         (2)
     <opérande-1>                                             (2.1)
```

sémantique: Si *AND* est spécifié, *sous-opérande-1* et *opérande-2* donnent:

- des valeurs booléennes, auquel cas *AND* dénote l'opération "et" logique usuelle, donnant une valeur booléenne;
- des valeurs chaîne de bits, auquel cas *AND* dénote l'opération "et" logique usuelle sur les chaînes de bits, donnant une valeur chaîne de bits;
- des valeurs ensembliste, auquel cas *AND* dénote l'opération d'intersection de valeurs ensembliste, donnant une valeur ensembliste comme résultat.

propriétés statiques: Si un *opérande-1* est un *opérande-2*, la classe de l'*opérande-1* est la classe de l'*opérande-2*.

Si *AND* est spécifié, la classe de l'*opérande-1* est la classe résultante des classes de l'*opérande-2* et *sous-opérande-1*.

Un *opérande-1* est constant (littéral) si et seulement s'il est soit un *opérande-2* qui est constant (littéral), soit construit d'un *opérande-1* et un *opérande-2* qui sont tous les deux constants (littéraux).

conditions statiques: Si *AND* est spécifié, la classe du *sous-opérande-1* doit être compatible avec la classe de l'*opérande-2*. Ces classes doivent toutes deux avoir un mode racine booléen, ensembliste ou chaîne de bits.

conditions dynamiques: Dans le cas de *AND* l'exception *RANGEFAIL* est causée si l'un ou les deux opérandes ont une classe dynamique et que la partie dynamique de la vérification de compatibilité mentionnée ci-dessus échoue.

exemples:

```
5.11    (a1 OR b1)                    (1.1)
5.11    NOT k2 AND (a1 OR b1)         (1.2)
```

5.3.4 *OPERANDE-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: Les opérateurs d'égalité (=) et 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 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 c'est le cas ou non qu'une valeur ensembliste est contenue dans: <=, contient: >=, est strictement contenue dans: <, ou contient strictement: > l'autre valeur ensembliste. L'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 construit d'un *opérande-2* et 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 *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 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 que la partie dynamique des vérifications de compatibilité mentionnées ci-dessus échoue. L'exception *TAGFAIL* est causée si et seulement si une classe dynamique est basée sur un mode structure paramétré dynamique.

exemples:

```
10.46   NULL                                    (1.1)
10.46   last=NULL                               (1.2)
```

5.3.5 *OPERANDE-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.

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*, est, selon les classes de l'*opérande-4* et *sous-opérande-3*:

  - si aucune des deux n'est forte, la classe est la *BIT(n)*-classe par dérivation ou *CHAR(n)*-classe par dérivation selon que les deux opérandes sont des chaînes de bits ou de caractères, où *n* est la somme des longueurs 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 de l'un des opérandes forts et où *n* dénote la somme des longueurs 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 *opérande-4* et de *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 d'un *opérande-3* et un *opérande-4* qui sont constants (littéraux).

conditions statiques: Si un *opérateur-4* est spécifié, les conditions de compatibilité suivantes doivent être remplies:

- si l'*opérateur-4* est un *opérateur arithmétique d'addition*, les classes des deux opérandes doivent être compatibles et avoir toutes les deux un mode racine entier;

- si l'*opérateur-4* et un *opérateur de concaténation de chaîne*, les modes racine des classes des deux opérandes doivent tous deux être des modes chaîne de bits ou des modes chaîne de charactères et, si les deux classes sont des classes par valeur, les modes racine doivent avoir la même nouveauté;

- si *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 comprise entre les bornes spécifiées par le mode racine de la classe de l'*opérande-3*.

exemples:

```
1.5     j                                                    (1.2)
1.5     i+j                                                  (1.2)
```

5.3.6 *OPERANDE-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*, $0 \leq K < J$ tel qu'il existe un entier *N* tel que *I=N*J+K*. *J* doit être supérieur à *0*.

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 de *sous-opérande-4* et de *opérande-5*.

Un *opérande-4* est constant (littéral) si et seulement s'il est soit un *opérande-5* qui est constant (littéral), soit construit d'un *opérande-4* et un *opérande-5* qui sont tous deux constants (littéraux).

conditions statiques: Si un *opérateur arithmétique multiplicatif* est spécifié, les classes de *opérande-5* et *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*) donnent une valeur qui n'est pas dans l'ensemble des valeurs défini par le mode racine de la classe de *opérande-4* ou s'effectuent 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)
```

5.3.7 *OPERANDE-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.

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 *opérande-6*.
- si l'*opérateur unaire* est l'*opérateur de répétition de chaîne* alors c'est la *CHAR(n)*- ou *BIT(n)*-classe par dérivation (dépendant du fait que le littéral était un *littéral chaîne de caractères* ou *littéral chaîne de bits*) où $n = r * L$, où r est la valeur donnée par l'*expression entière littérale* et L est la longueur du littéral chaîne.

Un *opérande-5* est constant (littéral) si et seulement si l'*opérande-6* est constant (littéral).

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 chaîne de caractères* ou un *littéral 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'opérateur changer-de-signe (-) donne une valeur qui n'est pas dans l'ensemble de valeurs défini par le mode racine de la classe de l'*opérande-5*.

exemples:

```
5.11    NOT k2                                  (1.1)
7.50    (6)' '                                  (1.1)
7.50    (6)                                     (2.2)
```

5.3.8 OPERANDE-6

syntaxe:

```
<opérande-6> ::=                                (1)
    <valeur primitive>                          (1.1)
  | <expression parenthésée>                    (1.2)

<expression parenthésée> ::=                    (2)
    (<expression>)                              (2.1)
```

sémantique: Un opérande-6 est soit une valeur primitive (voir section 5.2) ou une expression parenthésée.

propriétés statiques: La classe de l'*opérande-6* est la classe de la *valeur primitive* ou de l'*expression parenthésée*, respectivement. La classe de l'*expression parenthésée* est la classe de l'*expression*.

Un *opérande-6* est constant (littéral) si et seulement si respectivement la *valeur primitive* ou *expression* est constante (littérale).

exemples:

```
1.5     i                                       (1.1)
5.11    (a1 OR b1)                              (1.2)
```

6.0 ACTIONS

6.1 GENERALITES

syntaxe:

```
<énoncé d'action> ::=                                              (1)
     [<nom> :] <action> [<filet>] [<nom d'étiquette>];              (1.1)

<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)
   | <module>                                                      (3.4)
   | <bloc début-fin>                                              (3.5)
   | <action mettre en attente et choisir>                        (3.6)
   | <action recevoir et choisir>                                  (3.7)
```

sémantique: Les énoncés d'action constituent la partie algorithmique d'un programme CHILL. Toute action peut être étiquetée et les actions qui pourraient causer une exception peuvent se terminer par un filet.

propriétés statiques: Un *nom* suivi par un deux points et placé devant une *action*, et seulement un tel nom, est défini comme étant un nom d'étiquette.

conditions statiques: Le *nom d'étiquette* devant le point-virgule ne peut être donné que si l'*action* est une *action parenthésée* ou si un *filet* est spécifié et seulement si un *nom* suivi d'un deux points est donné avant l'*action*. Le *nom d'étiquette* doit être égal à ce dernier *nom*.

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> | <opérateur affectant>}
<valeur> (2.1)

<action d'affectation multiple> ::= (3)
<locus> {,<locus>} <symbole d'affectation> <valeur>* (3.2)

<opérateur affectant> ::= (4)
<opérateur binaire fermé> <symbole d'affectation> (4.1)

<opérateur binaire fermé> ::= (5)
OR | XOR | AND (5.1)
| *<opérateur de différence ensembliste>* (5.2)
| *<opérateur arithmétique additif>* (5.3)
| *<opérateur arithmétique multiplicatif>* (5.4)

<symbole d'affectation> ::= (6)
:= | = (6.1)

syntaxe dérivée: Le symbole = est une syntaxe dérivée pour le symbole *:=*.

sémantique: L'action d'affectation place une valeur dans un ou plusieurs locus.

Si le 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, les champs récurrents qui en dépendent, reçoivent une valeur indéfinie.

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 synchronisation. Chaque mode doit être compatible avec la classe de la *valeur*. Les vérifications sont dynamiques dans le cas de locus de mode dynamique et/ou de valeur dynamique.

Si la *valeur* est une *expression régionale* (voir section 8.2.2) chaque *locus* doit être régional.

Si, dans une *action d'affectation simple*, on spécifie un *opérateur affectant*, la *valeur* spécifiée doit être une *expression*.

conditions dynamiques: L'exception *TAGFAIL* est causée si, dans le cas d'un locus et/ou valeur de mode structure paramétré dynamique, la partie dynamique de la vérification de compatibilité mentionnée ci-dessus échoue.

L'exception *RANGEFAIL* est causée si un *locus* a un mode intervalle et que la valeur donnée par l'évaluation de *valeur* est hors des bornes spécifiées par ce mode intervalle.

L'exception *RANGEFAIL* est causée si, dans le cas d'un locus et/ou valeur de mode chaîne ou mode rangée paramétré dynamique, la partie dynamique de la vérification de compatibilité mentionnée ci-dessus échoue.

Les conditions mentionnées ci-dessus sont appelées les conditions d'affectation d'une valeur en tenant compte d'un mode (c.-à-d. le mode du locus).

Dans le cas d'un *opérateur affectant* les mêmes exceptions sont causées que 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:

4.11	*a:=b+c*	*(1.1)*
10.21	*stackindex-:=1*	*(2.1)*
19.16	*X.PREX, X.NEXT := NULL*	*(3.1)*
10.21	*-:=*	*(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: **L'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, la liste d'énoncés d'action qui suit *ELSE*, s'il y en a une.**

exemples:

```
7.24    IF n >= 10 THEN rn(r):='X';
              n-:=10;
              r+:=1;
        FI                                                  (1.1)
10.46   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> ::=                                   (4)
     <mode discret> {,<mode discret>}*                      (4.1)

<cas à choisir> ::=                                         (5)
     <spécification d'étiquettes de cas> :
     <liste d'énoncés d'action>                             (5.1)
```

sémantique: L'action de cas est un branchement multiple. Elle consiste en la spécification d'une ou 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). Chaque 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 étiquette 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 dans la liste des sélecteurs de cas.

Les expressions dans la liste de sélecteurs de cas sont évaluées dans un ordre indéfini et les évaluations peuvent éventuellement se mélanger. 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 section 9.1.3).

Le nombre d'occurrences d'*expression discrète* dans la liste de sélecteurs de cas doit être égal au nombre de classes dans la liste de classes résultante de la liste d'occurrences de *liste d'étiquettes de cas* et, si présente, au nombre d'occurrences 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 de classes résultante des occurrences de *liste d'étiquettes de cas* et, si présente, compatible avec le *mode discret* correspondant (par position) de la *liste d'intervalles*. Ce dernier mode doit aussi être compatible avec la classe correspondante de la liste de classes résultante.

Toute valeur donnée par une *expression littérale discrète* ou définie par un *intervalle littéral* ou un *mode discret* dans une *étiquette de cas* (voir section 9.1.3) doit résider dans l'intervalle du *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 *mode discret* correspondant dans la *liste d'intervalles*, si présente, doivent aussi résider dans cet intervalle.

Le nom réservé optionnel *ELSE*, suivi d'une *liste d'énoncés d'action*, ne peut s'omettre que si la liste d'occurrences de *liste d'étiquettes de cas* est complète (voir section 9.1.3).

conditions dynamiques: L'exception *RANGEFAIL* n'est causée que si une *liste d'intervalles* est spécifiée et que la valeur donnée par une *expression discrète* dans la *liste de sélecteurs de cas* ne réside pas entre les bornes spécifiées par le *mode discret* correspondant de la *liste d'intervalles*.

exemples:

```
4.10    CASE order OF
          (1): a:=b+c;
             RETURN;
          (2): d:=0;
          (ELSE): d:=1;
        ESAC                                                    (1.1)
11.44   starting.p.kind, starting.p.color                       (2.1)
11.62   (rook),(*):
            IF NOT ok_rook(b,m)
               THEN
                 CAUSE illegal;
            FI;                                                 (4.1)
```

6.5 *ACTION FAIRE*

6.5.1 *GENERALITES*

syntaxe:

```
<action faire> ::=                                              (1)
     DO [<commande>;] <liste d'énoncés d'action> OD             (1.1)

<commande> ::=                                                  (2)
     <commande pour> [<commande tandis>]                        (2.1)
   | <commande tandis>                                          (2.2)
   | <partie avec>                                              (2.3)
```

sémantique: L'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 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.

conditions dynamiques: L'exception *SPACEFAIL* est causée si les requêtes de mémoire ne peuvent pas être satisfaites.

exemples:

```
4.16    DO FOR i:=1 TO c;                                        (1.1)
           op(a,b,d,order-1);
           d:=a;
        OD
15.48   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)
     <nom>                                                       (5.1)

<valeur initiale> ::=                                            (6)
     <expression>                                                (6.1)

<valeur de pas> ::=                                              (7)
     BY <expression entière>                                     (7.1)

<valeur finale> ::=                                              (8)
     TO <expression>                                             (8.1)

<énumération par intervalle> ::=                                 (9)
     <compteur de boucle> [DOWN] IN <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 <locus rangée> (11.1)

sémantique: La liste d'énoncés d'action est entamée de façon répétée suivant la commande pour spécifiée.

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 décidable de terminer l'action faire. L'action faire est terminée si au moins un des compteurs de boucle indique la terminaison.

On fait une distinction entre terminaison normale et anormale. La terminaison normale arrive quand l'évaluation d'au moins un compteur de boucle indique la terminaison. La terminaison anormale arrive si l'évaluation d'une condition tandis donne *FALSE*, si une action sortir ou une action aller vers une étiquette (d'arrivée) définie en dehors de la liste d'énoncés d'action est exécutée, ou si une exception est causée pour laquelle le filet approprié est en dehors de, et ne termine pas, l'action faire.

1. *FOR EVER*:

   La liste d'énoncés d'action est répétée un nombre indéfini de fois; seule une terminaison anormale est possible.

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 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 est toujours défini implicitement à l'intérieur de la liste d'énoncés d'action. Cependant, si un nom d'accès qui est le même que le compteur de boucle est visible en dehors de l'action faire, la valeur du compteur de boucle sera placée dans le locus dénoté juste avant terminaison anormale. Dans le cas d'une terminaison normale, la valeur mise dans le locus dénoté par le nom d'accès externe est indéfinie.

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 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")*).
L'action faire est terminée (terminaison normale) si la liste d'énoncés d'action a été exécutée pour la plus grande (plus petite) valeur définie par le 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 entamée. Pour les exécutions suivantes de la liste d'énoncés d'action, la valeur suivante sera la valeur primitive suivante (précédente) dans la valeur ensembliste. L'action faire est terminée (terminaison normale) 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, valeur finale, et, éventuellement, 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")*).
L'action faire est terminée (terminaison normale) si l'évaluation donne une valeur qui est plus grande (plus petite) que la valeur finale, ou causerait 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*. La sémantique est comme si initialement la déclaration de loc-identité:
DCL <compteur de boucle> <mode> LOC := <premier locus>;
était rencontrée, où *<mode>* est le mode des éléments du mode du *locus rangée*, et *<premier locus>* l'élément d'indice le plus bas (le plus haut); pour les exécutions suivantes, comme si avant chaque exécution de la liste d'énoncés d'action la déclaration de loc-identité:
DCL <compteur de boucle> <mode> LOC := <locus suivant>;
était rencontrée, où *<locus suivant>* est l'élément de rangée d'indice:
"indice suivant" = SUCC("indice précédent")
(PRED("indice précédent")).
L'action faire est terminée (terminaison normale) si le compteur de boucle, juste avant l'évaluation suivante, indique l'élément de rangée qui a le plus grand (le plus petit) indice. Quand l'action faire est exécutée, le *locus rangée* n'est évalué qu'une seule fois.

propriétés statiques:

énumération de valeur:

Le *compteur de boucles* est un nom d'énumération de valeur. Si un nom visible dans le domaine dans lequel se trouve l'*action faire* est égal au *compteur de boucle*, le *compteur de boucle* est explicite, sinon il est implicite.

énumération par pas:

La classe d'un *compteur de boucle* explicite est la M-classe par valeur, ou M est le mode du nom d'accès externe (voir plus bas: conditions statiques).

La classe d'un *compteur de boucle* implicite est la classe résultante des classes de *valeur initiale*, *valeur de pas* si présente, et *valeur finale*.

énumération par intervalle:

La classe de compteur de boucle est la M-classe par valeur, où M est le *mode discret*.

énumération ensembliste:

La classe du *compteur de boucle* est la M-classe par valeur, où M est le mode primitif du mode de l'*expression ensembliste*.

énumération de locus:

Le *compteur de boucle* est un nom d'énumération de locus. Son mode est le mode des éléments du mode du *locus rangée*.

Un nom d'énumération de locus est repérable (selon le langage) si l'implantation d'élément du mode du *locus rangée* est *NOPACK*.

conditions statiques:

énumération par pas:

Les classes de *valeur initiale*, *valeur finale*, et *valeur de pas* si présente, doivent être deux à deux compatibles. Dans le cas d'un *compteur de boucle* qui est explicite, le nom visible à l'extérieur doit être un nom d'accès. Le mode du nom d'accès externe doit être compatible avec chacune de ces classes et ne peut pas être un mode protégé.

énumération ensembliste, énumération par intervalle:

Dans le cas d'un *compteur de boucle* explicite, le nom visible à l'extérieur doit être un nom d'accès. Le mode du nom d'accès externe doit être compatible avec la classe du *compteur de boucle*.

L'*expression ensembliste* doit être forte.

conditions dynamiques: Une exception *RANGEFAIL* est causée si la valeur donnée par *valeur de pas* n'est pas supérieure à 0 ou si, dans le cas d'un *compteur de boucle* explicite, la valeur à remettre dans le locus externe avant terminaison anormale ne réside pas entre les bornes spécifiées par le mode du locus externe. Cette exception est causée hors du bloc de l'action faire.

exemples:

```
4.16    FOR i:=1 TO c                                   (1.1)
15.27   FOR EVER                                        (1.1)
4.16    i:=1 TO c                                       (3.1)
9.11    j:=MIN(sieve) BY MIN(sieve) TO max              (3.1)
14.22   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 (terminaison anormale).

exemples:

7.28 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)*
| *<expression structure>* *(2.2)*

Note: Si l'*expression structure* est un *locus*, la construction syntaxique est ambiguë et sera interprétée comme un *locus structure*.

sémantique: Les noms de champ (visibles) des locus structure ou des valeurs structure spécifiés dans chaque *commande avec* sont rendus disponibles comme accès directs aux champs.

Si un *locus structure* est spécifié, des noms d'accès qui sont égaux aux noms de champ du mode du *locus structure* sont créés implicitement, dénotant les sous-locus du locus structure.

Si une *expressions structure* est spécifiée, des noms de valeur qui sont égaux aux noms de champ du mode de l'*expression structure* (forte) sont créés implicitement, dénotant les sous-valeurs de la valeur structure.

Quand on entame l'action faire, les locus structure et/ou les expressions structure ne sont évalués qu'une fois en entamant l'action faire, dans un ordre quelconque et les évaluations peuvent éventuellement se mélanger.

propriétés statiques:

expression structure: Tout nom rendu disponible dans l'action faire est un nom de valeur faire-avec. La classe est la M-classe par valeur, où M est le mode de ce nom de champ du mode structure de l'*expression structure* qui est rendu disponible comme nom de valeur faire-avec.

Locus structure: Tout nom rendu disponible dans l'action faire est 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 (par définition du langage) si l'implantation de champ du nom de champ associé est *NOPACK*

conditions statiques: L'*expression structure* doit être forte.

exemples:

15.48 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 une action parenthésée. L'exécution reprend immédiatement après l'action parenthésée englobant du plus près et étiquetée par le *nom d'étiquette*.

conditions statiques: L'*action sortir* doit résider à intérieur de l'énoncé d'action parenthésée étiqueté du *nom d'étiquette*. Si l'*action sortir* se trouve à l'intérieur d'une définition de procédure ou de processus, l'énoncé d'action parenthésé dont on sort doit aussi résider à l'intérieur de la même définition de procédure ou de processus (c.-à-d. l'action sortir ne peut s'employer pour quitter des procédures ou des processus).

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

exemples:

15.52 EXIT FIND_COUNTER (1.1)

6.7 *ACTION APPELER*

syntaxe:

<action appeler> ::= (1)
[CALL]
{<appel de procédure>
| <appel d'opération prédéfinie>} (1.1)

<appel de procédure> ::= (2)
{<nom de procédure> | <expression 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 de mode statique>* (4.2)

syntaxe dérivée: Le nom réservé *CALL* est facultatif. Une action appeler avec *CALL* est dérivée d'une action appeler sans *CALL*.

sémantique: Une action appeler cause un appel à la procédure générale indiquée par la valeur donnée par l'expression procédure, ou à la procédure indiquée par le nom de procédure. Les valeurs effectives et les locus effectifs spécifiés dans la liste de paramètres effectifs sont passés à la procédure.

propriétés statiques: Un *appel de procédure* a les propriétés suivantes: il a une liste de specs de paramètres, éventuellement une spec de résultat, un ensemble éventuellement vide de noms d'exception, une généralité, une récursivité, et il peut être régional (cette dernière propriété n'est possible que pour un *nom de procédure*, voir section 8.2.2). Ces propriétés sont héritées du *nom de procédure* ou d'un mode compatible avec la classe de l'*expression 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*.

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 paramètres. 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 peut pas avoir la propriété de synchronisation. Si l'*appel de procédure* n'est pas régional, la *valeur* (effective) ne peut pas être régionale (voir section 8.2.2).

- Si la spec de paramètre a l'attribut *INOUT* ou *OUT*, le *paramètre effectif* doit être un *locus de mode statique*, 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 de mode statique* (effectif) ne peut avoir la propriété de protection ni la propriété de synchronisation. Si l'*appel de procédure* n'est pas régional, le *locus* (effectif) ne peut être régional (voir section 8.2.2).

- 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 de mode statique*.

- Si la spec de paramètre a l'attribut *LOC*, le *paramètre effectif* doit être un *locus de mode statique* qui est à la fois repérable et tel que le mode dans la spec de paramètre est compatible en lecture avec le mode de ce *locus de mode statique* (effectif), ou être une *valeur* qui n'est pas un *locus* mais dont la classe est compatible avec le mode dans la spec de paramètre.

conditions dynamiques: Un appel de procédure peut causer toute exception de l'ensemble de noms d'exception qui lui est attaché. Il cause l'exception *EMPTY* si l'*expression procédure* donne *NULL*, il cause l'exception *SPACEFAIL* si on ne peut satisfaire les requêtes de mémoire et il cause l'exception *RECURSEFAIL* si la procédure s'appelle elle-même récursivement (c.-à-d. une activation précédente est toujours active) et que sa récursivité est non récursive.

Le passage des paramètres peut causer les exceptions suivantes:

- Si la spec de paramètre a l'attribut *IN*, *INOUT* ou *LOC*, les conditions d'affectation de la valeur (effective) (éventuellement contenue dans un locus effectif), en tenant compte du mode de la spec de paramètre doivent être respectées à l'endroit de l'appel (voir 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 section 6.2) et les exceptions possibles sont causées aprés 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 section 6.2).

L'*expression procédure* ne peut pas donner une procédure définie dans un processus dont l'activation n'est pas celle du processus qui exécute l'appel de procédure (voir section 8.1) et la durée de vie de la procédure dénotée ne peut pas être terminée.

exemples:

```
4.17    op(a,b,d,order-1)                                    (1.1)
```

6.8 *ACTION RESULTER ET ACTION REVENIR*

syntaxe:

```
<action revenir> ::=                                         (1)
     RETURN [<résultat>]                                     (1.1)

<action résulter> ::=                                        (2)
     RESULT <résultat>                                       (2.1)

<résultat> ::=                                               (3)
     <valeur>                                                (3.1)
   | <locus de mode statique>                                (3.2)
```

syntaxe dérivée: L'*action revenir* avec *résultat* est dérivé de *RESULT <résultat> ; RETURN*
Si un *filet* termine une telle *action revenir*, il est consideré comme terminant l'*action résulter* de laquelle il est dérivé.

sémantique: L'action résulter sert à établir le résultat devant être rendu par un appel de procédure. Ce résultat peut être un locus ou une valeur. L'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 l'action résulter exécutée en dernier lieu. Si aucune action résulter n'a été exécutée, l'appel de procédure donne un locus indéfini ou une valeur indéfinie.

propriétés statiques: A l'*action résulter* et à l'*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: L'*action revenir* et l'*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 peut terminer une *action revenir* (sans *résultat*).

Si *LOC* est spécifié dans la spec de résultat du nom de procédure de l'*action résulter*, le *résultat* doit être un *locus de mode statique*, tel que le mode dans la spec de résultat soit compatible en lecture avec le mode du *locus de mode statique*. Si le nom de procédure de l'*action résulter* n'est pas régional, le *locus de mode statique* dans le *résultat* ne peut pas être régional (voir section 8.2.2).

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. Si le nom de procédure d'une *action résulter* n'est pas régional, la *valeur* dans le *résultat* ne peut pas être régionale (voir section 8.2.2).

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:

4.20	*RETURN*	*(1.1)*
1.5	*RESULT i+j*	*(2.1)*
5.20	*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 du nom d'étiquette.

conditions statiques: Si l'*action aller* se trouve à l'intérieur d'une définition de procédure ou de processus, l'étiquette indiquée par le *nom d'étiquette* doit aussi être définie à l'intérieur de la définition (c.-à-d. il n'est pas possible de sauter hors d'une invocation de procédure ou de processus).

Un *filet* ne peut pas terminer une *action aller*.

6.10 *ACTION AFFIRMER*

syntaxe:

<action affirmer> ::=	*(1)*
ASSERT <expression booléenne>	*(1.1)*

sémantique: L'action affirmer fournit une manière de tester une condition.

conditions dynamiques: L'exception *ASSERTFAIL* est causée si l'*expression booléenne* donne *FALSE*.

exemples:

4.6 . *ASSERT b>0 AND c>0 AND order>0* (1.1)

6.11 ACTION VIDE

syntaxe:

<action vide> ::= (1)
<vide> (1.1)

<vide> ::= (2)

sémantique: L'action vide ne fait rien.

conditions statiques: Un *filet* ne peut pas terminer une *action vide*.

6.12 ACTION CAUSER

syntaxe:

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

sémantique: L'action causer cause une exception.

conditions statiques: Un *filet* ne peut pas terminer une *action causer*.

conditions dynamiques: L'*action causer* cause l'exception dont le nom est indiqué par *nom d'exception*.

exemples:

4.8 *CAUSE wrong_input* (1.1)

6.13 ACTION DEMARRER

syntaxe:

<action démarrer> ::= (1)
<expression démarrer> [SET <locus exemplaire>] (1.1)

syntaxe dérivée: L'*action démarrer* avec l'option *SET* est une syntaxe dérivée pour l'action d'affectation simple:
<locus exemplaire>:= <expression démarrer>

sémantique: L'action démarrer évalue l'expression démarrer (voir section 5.2.17), sans employer la valeur exemplaire donnée par cette expression.

exemples:

14.37 START CALL_DISTRIBUTOR() *(1.1)*

6.14 ACTION ARRETER

syntaxe:

<action arrêter> ::= *(1)*
STOP *(1.1)*

sémantique: L'action arrêter termine le processus qui exécute l'action arrêter.

conditions statiques: Un *filet* ne peut pas terminer une *action arrêter*.

6.15 ACTION CONTINUER

syntaxe:

<action continuer> ::= *(1)*
CONTINUE <locus événement> *(1.1)*

sémantique: L'action continuer permet au processus qui a la plus haute priorité, et qui est en attente sur le locus événement spécifié, d'être activé. S'il n'y a pas de processus unique de plus haute priorité, un des processus de plus haute priorité sera choisi suivant un algorithme défini par l'implémentation. S'il n'y a aucun processus en attente sur le locus événement spécifié, l'action continuer n'a aucun effet (voir chapitre 8 pour plus de détails).

exemples:

13.23 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: L'action mettre en attente cause la mise en attente du processus qui l'exécute. Il peut être activé par une action continuer sur le locus événement spécifié. La priorité indique la priorité du processus mis en attente dans

l'ensemble des processus qui sont en attente sur le locus événement indiqué. La priorité la plus basse, et par défaut, est 0 (voir chapitre 8 pour plus de détails).

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 un attribut de longueur et que le nombre de processus en attente sur le locus événement spécifié est égal à cette longueur juste avant l'évaluation du locus événement. Cette exception est causée avant la mise en attente du processus.

La durée de vie du locus événement spécifié ne peut pas se terminer pendant que le processus qui exécute une action mettre en attente est en attente sur lui.

exemples:

13.17 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>;]
*{<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>}**

sémantique: L'action mettre en attente et choisir cause la mise en attente du processus qui l'exécute. Il peut être activé par une action continuer sur l'un des locus événement spécifiés. Dans ce cas la liste d'énoncés d'action qui est précédée par le locus événement sur lequel l'action continuer qui a réactivé le processus est exécutée, sera entamée (voir chapitre 8 pour plus de détails). Avant que le processus ne soit mis en attente, chaque locus événement, et le locus exemplaire si spécifié, sera évalué. Ils seront évalués dans un ordre quelconque et peuvent éventuellement se mélanger. Si deux évaluations ou plus donnent le même locus événement, le choix d'une liste d'énoncés d'action est non-déterministe.

Si un locus exemplaire est spécifié, la valeur exemplaire qui identifie le processus qui exécuta l'action continuer activante, sera mise dans le locus exemplaire.

conditions statiques: Le mode du *locus exemplaire* ne peut pas avoir la propriété de protection. L'*expression littérale entière* dans *priorité* ne peut pas donner une valeur négative.

conditions dynamiques: L'exception *DELAYFAIL* est causée si le mode d'au moins un *locus événement* a un attribut de longueur tel que le nombre de processus en attente sur le locus événément spécifié est égal à la longueur après l'évaluation du *locus événement*. Cette exception est causée avant la mise en attente du processus.

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.

exemples:

```
14.20   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 GENERALITES

syntaxe:

```
<action envoyer> ::=                                            (1)
      <action envoyer signal>                                   (1.1)
    | <action envoyer tampon>                                   (1.2)
```

sémantique: L'action envoyer initie le transfert d'information de synchronisation à partir d'un processus envoyant. La sémantique détaillée dépend de ce que 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 <expression exemplaire>] [<priorité>]                 (1.1)
```

sémantique: Le signal spécifié est envoyé en même temps que la liste de valeurs et la priorité (si présente). La priorité par défaut et la plus basse est 0. Si le nom de signal a un nom de processus, cela signifie que seuls des processus de ce nom peuvent recevoir le signal. Si l'option *TO* est spécifiée, elle identifie le seul processus qui peut recevoir la liste des valeurs envoyées dans l'action envoyer signal. Cette identification de processus ne peut être en contradiction avec le nom de processus éventuellement attaché au nom de signal. A la fois le nom de processus attaché éventuellement au nom de signal et la valeur exemplaire possible s'attachent dynamiquement à la liste de valeurs envoyée (voir chapitre 8 pour plus de détails).

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 régionale (voir section 8.2.2). 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 l'*expression exemplaire* donne *NULL*.

L'exception *EXTINCT* est causée si et seulement si la durée de vie du processus indiqué par la valeur donnée par l'*expression exemplaire* est terminée au point d'exécution de l'action envoyer signal.

L'exception *MODEFAIL* 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 l'*expression exemplaire*.

exemples:

```
15.68   SEND READY TO RECEIVED_USER                    (1.1)
15.76   SEND READOUT(COUNT) TO USER
```

6.18.3 *ACTION ENVOYER TAMPON*

syntaxe:

```
<action envoyer tampon> ::=                            (1)
     SEND <locus tampon>(<valeur>) [<priorité>]        (1.1)
```

sémantique: La valeur spécifiée en même temps que la priorité est mise dans le locus tampon si sa capacité le permet. Ce n'est pas le cas si le mode du *locus tampon* a un attribut de longueur et que le nombre de valeurs qui sont dans le tampon

est égal à la longueur juste avant l'exécution de l'action envoyer tampon. Comme résultat, le processus envoyant sera mis en attente jusqu'à ce que la capacité soit suffisante dans le locus tampon ou jusqu'à ce que la valeur envoyée soit consommée. La priorité par défaut et la plus basse est 0. (voir chapitre 8 pour plus de détails).

conditions statiques: La classe de *valeur* doit être compatible avec le mode des éléments de tampon du mode du *locus tampon*. La *valeur* ne peut pas être régionale (voir section 8.2.2). L'*expression littérale entière* dans *priorité* ne doit pas donner une valeur négative.

conditions dynamiques: Pour l'*action envoyer tampon* 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 peut se terminer pendant que le processus qui exécute l'action envoyer tampon est en attente sur lui.

exemples:

16.115 SEND USER->([READY, ->COUNTER_BUFFER]) *(1.1)*

6.19 *ACTION RECEVOIR ET CHOISIR*

6.19.1 *GENERALITES*

syntaxe:

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

sémantique: L'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 résulte pas nécessairement en la mise en attente du processus exécutant (voir chapitre 8 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 choisir>}+
[ELSE <liste d'énoncés d'action>] ESAC (1.1)

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

sémantique: L'action recevoir signal et choisir reçoit un signal, éventuellement accompagné d'une liste de valeurs, dont le nom de signal est spécifié dans un signal à choisir.

Quand on entame une action recevoir signal et choisir, et si un signal de l'un des noms spécifiés et qui peut être reçu par un processus exécutant cette action est présent pour réception, le signal est reçu. Si un tel signal n'est pas présent et si *ELSE* n'est pas spécifié, le processus qui exécute l'action recevoir signal et choisir est mis en attente; si *ELSE* est spécifié, la liste d'énoncés d'action qui le suit sera entamée.

Un signal peut être reçu par un processus seulement si les conditions suivantes sont remplies:

- Si un nom de processus est attaché au signal, le nom du processus qui reçoit est ce nom de processus.
- Si une valeur exemplaire est attachée au signal elle identifie le processus qui reçoit.

Si un signal peut être reçu, la liste d'énoncés d'action précédée par le nom de signal du signal reçu sera entamée. Si plus d'un signal peut être reçu, on choisira un signal de la plus haute priorité suivant un algorithme de sélection défini par l'implémentation. Si le nom de signal a défini une liste de modes, c.-à-d. si une liste de valeurs est envoyée avec le signal, une liste de noms doit être spécifiée après *IN*. Ce sont des noms de valeur introduits pour dénoter les valeurs reçues. Si dans le domaine dans lequel l'action recevoir signal et choisir est placée un nom d'accès est visible qui est égal au nom introduit, la valeur reçue sera mise dans le locus dénoté immédiatement après réception du signal et donc avant l'exécution de la liste d'énoncés d'action.

Si l'option *SET* est spécifiée, la valeur exemplaire qui dénote le processus qui a envoyé le signal reçu sera mise dans le locus exemplaire spécifié, immédiatement après réception du signal.

propriétés statiques: Tout nom défini dans la *liste de nom* d'un *signal à choisir* est un nom de valeur reçue. Sa classe est la M-classe par valeur, où M est le mode correspondant du *nom de signal* qui précède. Si un nom est visible dans le domaine dans lequel l'*action recevoir signal et choisir* est placée et est égal à un des noms introduits après *IN*, le nom de valeur reçue est explicite, sinon implicite.

conditions statiques: Le mode du *locus exemplaire* ne peut avoir la propriété de protection.

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

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

Si le nom de valeur reçue est explicite, le nom visible à l'extérieur doit être un *nom d'accès* et son mode doit être compatible avec la classe du nom de valeur reçue, qui a le même nom. Le mode du *nom d'accès* ne peut pas avoir la propriété de protection.

conditions dynamiques: Si le nom de valeur reçue est explicite les conditions d'affectation de la valeur reçue en tenant compte du mode du *nom d'accès* externe doivent être respectées. Les exceptions possibles sont causées, après avoir reçu le signal et avant d'entamer la liste d'énoncés d'action.

L'exception *SPACEFAIL* est causée si, quand on entame la liste d'énoncés d'action, les requêtes de mémoire ne peuvent être satisfaites.

exemples:

```
15.73   RECEIVE CASE
           (STEP): 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 <nom>)

: <liste d'énoncés d'action> (2.1)

sémantique: L'action recevoir tampon et choisir reçoit une valeur d'un locus tampon ou d'un processus envoyant mis en attente sur un locus tampon, lequel est indiqué dans un tampon à choisir.

Quand on entame une action recevoir tampon et choisir et si une valeur réside dans, ou si un processus envoyant est en attente sur, un des locus tampon spécifiés, la valeur sera reçue et une liste d'énoncés d'action précédée par un *locus tampon* donnant le locus tampon d'où venait la valeur sera exécutée.

Quand l'action recevoir tampon et choisir est entamée, les locus tampon sont évalués dans un ordre non spécifié et ne doivent l'être que jusqu'à ce qu'un choix puisse être sélectionné. Si aucun des locus tampon spécifiés ne contient une valeur ni qu'aucun processus n'est en attente sur un locus tampon spécifié et si *ELSE* n'est pas spécifié, le processus exécutant est mis en attente. Si *ELSE* est spécifié la liste d'énoncés d'action qui le suit sera exécutée. Si plus d'une valeur peut être reçue, une valeur de la plus haute priorité sera sélectionnée suivant un algorithme de sélection défini par l'implémentation. Si deux occurrences de *locus tampon* ou plus donnent le même locus tampon d'où la valeur est reçue, la sélection de la liste d'énoncés d'action n'est pas déterministe.

La valeur est reçue immédiatement avant d'entamer la liste d'énoncés d'action qui suit le deux points. Le nom après *IN* est un nom de valeur reçue introduit et qui dénote la valeur reçue. Si, dans le domaine dans lequel se trouve l'*action recevoir tampon et choisir*, un nom d'accès est visible qui est égal au nom de valeur reçue introduit, la valeur reçue sera mise dans le locus dénoté, immédiatement avant d'entamer la liste d'énoncés d'action.

Si l'option *SET* est spécifiée, le locus exemplaire spécifié contient, immédiatement à la réception, la valeur exemplaire dénotant le processus qui a envoyé la valeur reçue.

propriétés statiques: Le nom après *IN* dans le *tampon à choisir* est 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* commençant le *tampon à choisir*.

Si un nom est visible dans le domaine dans lequel se trouve l'*action recevoir tampon et choisir* et qui est égal au *nom* introduit après *IN*, le nom de valeur reçue est appelé explicite, sinon implicite.

conditions statiques: Le mode de *locus exemplaire* ne peut pas avoir la propriété de protection. Si le nom de valeur reçue est explicite, le nom visible à l'extérieur doit être un *nom d'accès* et son mode doit être compatible avec la classe du nom de valeur reçue qui a le même nom. Ce mode ne peut avoir la propriété de protection.

conditions dynamiques: Si le nom de valeur reçue est explicite les conditions d'affectation de la valeur reçue en tenant compte du mode du *nom d'accès* externe doivent être remplies. Les exceptions possibles sont causées après avoir reçu la valeur et avant d'entamer la liste d'énoncés d'action.

L'exception *SPACEFAIL* est causée si, quand on entame une liste d'énoncés d'action, les requêtes de mémoire ne peuvent être satisfaites.

La durée de vie d'aucun des locus tampon donnés ne peut se terminer pendant que le processus qui exécute l'action recevoir et choisir est en attente sur lui.

7.0 STRUCTURE DE PROGRAMME

7.1 GENERALITES

Les *actions faire, bloc début-fin, module, région, action mettre en attente et choisir, action recevoir et choisir, définition de procédure* et *définition de processus,* toutes parenthésées, déterminent la structure du programme, c.-à-d. 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 l'*action faire,* y compris le *compteur de boucle* et la *commande tandis;*
  - le *bloc début-fin;*
  - la *définition de procédure,* en excluant la *spec de résultat;*
  - la *définition de processus;*
  - la *liste d'énoncés d'action* dans un *tampon à choisir* ou dans un *signal à choisir,* y compris le *nom* ou *liste de noms* après *IN;*
  - la *liste d'énoncés d'action* après *ELSE* dans une *action recevoir et choisir* ou un *filet;*
  - le *choix d'exceptions* dans un *filet.*
- Le mot modulion sera employé pour dénoter soit un *module* soit une *région.*
- 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 cette partie du groupe qui n'est pas englobée par dans un groupe interne au groupe (c.-à-d. la partie qui consiste en le niveau d'imbrication le plus externe du groupe).

Un groupe définit une portée pour les noms créés dans son domaine. Des noms peuvent être créés des manières suivantes:

- Un nom qui apparaît dans la *liste de noms* d'une *déclaration,* d'une *définition de mode,* ou d'une *définition de synonyme,* ou qui apparaît dans une *définition de signal* est créé dans le domaine dans lequel respectivement la *déclaration,* la *définition de mode,* la *définition de synonyme* ou la *définition du signal* apparaît.
- Un nom qui apparaît dans la *liste de noms* dans une *liste de paramètres formels* est créé dans le domaine de la *définition de procédure* ou de la *définition de processus* correspondante.

- Un nom, devant un deux points, suivi par une *action*, par une *région*, par une *définition de procédure*, par une *définition d'entrée* ou par une *définition de processus* est créé dans le domaine dans lequel respectivement l'*action*, la *région*, la *définition de procédure*, la *définition de procédure* qui contient la *définition d'entrée*, la *définition de processus* apparaît.

- Chaque nom d'énumération de valeur, nom d'énumération de locus, nom de valeur faire-avec et nom de locus faire-avec est créé dans le domaine du bloc de l'*action faire* associée.

- Chaque nom de valeur reçue est créé dans le domaine du bloc du *signal à choisir* ou du *tampon à choisir* associé.

- Un nom de champ ou un nom d'élément d'ensemble est créé dans le domaine où l'occurrence de définition du *mode structure* ou du *mode ensemble* associé est placée.

- Un nom d'exception est créé par une *action causer* ou une occurrence de *exceptions à choisir* (note: il n'y a pas de point spécifique de création d'un nom d'exception; voir chapitre 10).

- Un nom prédéfini par le langage est considéré comme étant créé dans le domaine du module prélude standard (voir section 7.8).

Les noms introduits (créés) par le programmeur, à l'exception des noms d'exception, ont un endroit unique où ils sont créés (déclarés ou définis). Cet endroit est appelé l'occurrence de définition du nom. Les endroits où le nom est employé sont appelés occurrences d'utilisation du nom. Les règles d'identification associent une occurrence de définition unique à chaque occurrence d'utilisation du nom (voir section 3.2.8). Il n'y a pas de distinction entre occurrence de définition et occurrence d'utilisation pour les noms d'exception (voir chapitre 10).

Un nom a une certaine portée, c.-à-d. cette 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 ont une certaine durée de vie, c.-à-d. cette 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.

7.2 DOMAINES ET IMBRICATION

syntaxe:

<corps de bloc) ::= *(1)*

```
        <liste d'énoncés informatifs>
        <liste d'énoncés d'action>                                  (1.1)

    <corps de procédure> ::=                                        (2)
        <liste d'énoncés informatifs>
        {<énoncé d'action> | <énoncé d'entrée>}*                     (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> }* <liste d'énoncés d'action>                     (4.1)

    <corps de région> ::=                                           (5)
        {<énoncé informatif> | <énoncé de visibilité>}*             (5.1)

    <liste d'énoncés d'action> ::=                                  (6)
        {<énoncé d'action>}*                                        (6.1)

    <liste d'énoncés informatifs> ::=                               (7)
        {<énoncé informatif>}*                                      (7.1)

    <énoncé informatif> ::=                                         (8)
        <énoncé déclaratif>                                         (8.1)
      | <énoncé définissant>                                        (8.2)

    <énoncé définissant> ::=                                        (9)
        <énoncé de définition de synmodes>                          (9.1)
      | <énoncé de définition de neumodes>                          (9.2)
      | <énoncé de définition de synonymes>                         (9.3)
      | <énoncé de définition de procédure>                         (9.4)
      | <énoncé de définition de processus>                         (9.5)
      | <énoncé de définition de signal>                            (9.6)
      | <vide>;                                                     (9.7)
```

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 et éventuellement les évaluations dynamiques des déclarations de loc-identité sont faites dans l'ordre dans lequel elles sont spécifiées textuellement.

Quand on entame le domaine d'un modulion, les initialisations domaniales et éventuellement les évaluations dynamiques des déclarations de loc-identité dans le domaine du modulion sont faites dans l'ordre dans lequel elles sont spécifiées textuellement.

propriétés statiques: Tout domaine possède un groupe englobant immédiatement unique, défini de la manière suivante:

- 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*, d'un *module* ou d'une *région*, alors son groupe englobant immédiatement est respectivement le groupe dans le domaine duquel l'*action faire*, le *bloc début-fin*, la *définition de procédure*, la *définition de processus*, le *module* ou la *région* se trouve.

- Si le groupe est la *liste d'énoncés d'action* incluant le cas échéant les noms introduits, d'un *tampon à choisir* ou d'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*, alors son groupe englobant immédiatement est respectivement le groupe dans le domaine duquel l'*action recevoir tampon et choisir* ou *action recevoir signal et choisir*, se trouve.

- Si le domaine est la *liste d'énoncés d'action* dans un *choix d'exceptions* ou la *liste d'énoncés d'action* qui suit *ELSE*, dans un *filet* qui ne termine pas un groupe, alors le groupe englobant immédiatement est le groupe dans le domaine duquel l'action terminée par le *filet*, se trouve.

- Si le domaine est un *choix d'exceptions* ou *liste d'énoncés d'action* après *ELSE*, d'un *filet* qui termine un groupe, alors son groupe englobant immédiatement est le groupe terminé par le *filet*.

Un domaine a un domaine englobant immédiatement unique, qui est le domaine du groupe englobant immédiatement. Un énoncé a un groupe englobant immédiatement unique, qui est le groupe dans le domaine duquel l'énoncé se trouve. Un domaine est dit englober immédiatement un groupe (domaine) si et seulement si le domaine est le domaine englobant immédiatement le groupe (domaine).

Un énoncé (domaine) est dit être englobé par un groupe, si et seulement si, soit le groupe est le groupe englobant immédiatement l'énoncé (domaine), soit le domaine englobant immédiatement 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 l'exécution 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 entrée principale (c.-à-d. pas via un point d'entrée défini additionnellement).

- Domaine processus: le processus est activé via une action démarrer.

- Domaine faire: l'action faire est exécutée comme une action après l'évaluation des expressions ou locus dans la partie 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'exception est exécuté à cause d'une exception.

Une liste d'énoncés d'action est dite être entamée quand et seulement quand sa premiere action, si présente, reçoit le contrôle depuis l'extérieur de la liste d'énoncés d'action.

7.3 BLOCS DEBUT-FIN

syntaxe:

<bloc début-fin> ::= *(1)*
BEGIN <corps de bloc> END *(1.1)*

sémantique: Un bloc début-fin est une action (action composite), 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 sections 7.9. et 9.2.5).

conditions dynamiques: Une exception *SPACEFAIL* est causée si le *bloc début-fin* requiert de la mémoire locale pour laquelle les requêtes de mémoire ne peuvent être satisfaites.

exemples:

voir 15.63 - 15.80

7.4 DEFINITIONS DE PROCEDURE

syntaxe:

```
<énoncé de définition de procédure> ::=                                  (1)
     <nom> : <définition de procédure>
     [<filet>] [nom de procédure>];                                      (1.1)

<définition de procédure> ::=                                            (2)
     PROC( [<liste de paramètres formels>] )
     [<spec de résultat>]
     [EXCEPTIONS( <liste d'exceptions> )]
     <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 noms> <spec de paramètre>                                 (4.1)

<attributs de procédure> ::=                                             (5)
     [<généralité>] [RECURSIVE]                                          (5.1)

<généralité> ::=                                                         (6)
     GENERAL                                                             (6.1)
   | SIMPLE                                                              (6.2)
   | INLINE                                                              (6.3)

<énoncé d'entrée> ::=                                                    (7)
     <nom> : <définition d'entrée>;                                      (7.1)

<définition d'entrée> ::=                                                (8)
     ENTRY                                                               (8.1)
```

syntaxe dérivée: Un paramètre formel où la *liste de noms* comporte plus d'un nom est dérivé de plusieurs occurrences de *paramètre formel* séparées par des virgules, une pour chaque nom et chacune avec la même *spec de paramètre*. Par exemple:
I, J INT LOC
est dérivé de:
I INT LOC, J INT LOC

sémantique: Une définition de procédure définit une séquence d'actions (éventuellement paramétrée) qui peut être appelée de différents endroits du programme. Le contrôle revient au point d'appel soit en exécutant une action revenir, ou en atteignant la fin du corps de procédure ou d'un choix d'exceptions du filet qui termine la définition de procédure (passant les bornes). Différents degrés de complexité de procédure peuvent se spécifier comme suit:

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,

elles ne peuvent pas être mises dans des locus procédure, ni ne peuvent être passées comme paramètres à, ou retournées comme résultat d'un, appel de procédure.

Les procédures générales (*GENERAL*) n'ont pas les restrictions des procédures simples et peuvent être traitées comme des valeurs procédures.

Les procédures in-situ (*INLINE*) ont les mêmes restrictions que les procédures simples et elles ne peuvent être récursives. Elles ont la même sémantique que les procédures normales, mais le compilateur insérera le code généré à l'endroit de l'invocation plutôt que de générer du 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).

Le nom devant la définition de procédure définit le nom de la procédure. Si le nom de procédure est général, c'est un littéral de procédure pour la valeur procédure définie. Sa classe est déterminée par les modes et attributs dans les paramètres formels et la spec de résultat.

Une procédure peut avoir des points d'entrée multiples par l'emploi d'énoncés d'entrée. Les énoncés sont considérés comme des définitions de procédure additionnelles. Le nom dans l'énoncé d'entrée définit le nom du point d'entrée de la procédure. Le point d'entrée est défini par la position textuelle de l'énoncé d'entrée.

passage de paramètre:

Il y a fondamentalement deux mécanismes de passage de paramètres: le passage par valeur et le passage par locus (attribut *LOC*). Les attributs *OUT* et *INOUT* indiquent des variations dans le mécanisme de passage par valeur.

passage par valeur

Dans le passage de paramètre 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 <nom de paramètre formel><mode> := <paramètre effectif>;
était recontrée. Cependant, cette initialisation ne peut pas causer d'exception à l'intérieur du corps de procédure. 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*, avec l'exception 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 paramètre par locus, un locus est passé comme paramètre au corps de procédure. Ni des locus non repérables, ni des locus dynamiques ne peuvent être passés de cette manière. Tout se passe comme si au point d'entrée de la procédure la déclaration de loc-identité:

DCL <nom de paramètre formel><mode> LOC
:= <paramètre effectif>;

était rencontrée. Cependant, une telle déclaration ne peut causer une exception à l'intérieur du corps de procédure.

Si une *valeur* est spécifiée, qui n'est pas un *locus de mode statique*, un locus contenant la valeur spécifiée sera créé implicitement et passé à l'endroit de l'appel. La durée de vie du locus créé est celle de l'appel de procédure.

transmission de résultat:

Une valeur ou un locus peuvent être retournés par la procédure. Dans le premier cas, une *valeur* est spécifiée dans toute *action résulter*; dans le dernier cas, un *locus de mode statique* (voir section 6.8). La valeur ou le locus retourné est déterminé par l'action résulter exécutée le plus récemment 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 un appel de procédure rendant locus (voir section 4.2.10) ni comme un appel de procédure rendant valeur (voir section 5.2.15) mais seulement comme une action appeler (section 6.7).

spécification de registre:

Une spécification de registre peut être donnée dans le paramètre formel de la procédure, et dans la spec de résultat. Dans le cas d'un passage par valeur, cela signifie que la valeur effective est contenue dans le registre spécifié; dans

le cas d'un passage par locus, cela signifie que le pointeur (caché) vers le locus effectif est contenu dans le registre spécifié. Si un registre est spécifié dans une spec de résultat, cela signifie que la valeur retournée ou que le pointeur (caché) vers le locus retourné est contenu dans le registre spécifié.

propriétés statiques: Un nom est un nom de procédure si et seulement si il est défini dans un *énoncé de définition de procédure* ou *un énoncé d'entrée* (c.-à-d. placé devant un deux points et une *définition de procédure* ou *définition d'entrée*).

Un nom de procédure possède une *définition de procédure* qui est définie comme:

- Si un nom de procédure est défini dans un *énoncé de définition de procédure*, alors c'est la *définition de procédure* dans cet énoncé.

- Si le nom de procédure est défini dans un *énoncé d'entrée*, alors c'est la *définition de procédure* dans le domaine de laquelle l'*énoncé d'entrée* est placé.

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 spec de paramètre consistant en un mode, éventuellement un attribut de paramètre et/ou un nom de registre.

- Il a éventuellement une spec de résultat, consistant en un mode, éventuellement l'attribut *LOC* et/ou un nom de registre.

- Il a un ensemble 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 alors soit général, soit simple, soit in-situ, dépendant de ce 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 section 8.2) la récursivité est non-récursive.

Un nom de procédure qui est général, est un littéral procédure. 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 paramètre formels*, séparées par des virgules.

Un nom défini dans une *liste de noms* 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 (par définition du langage).

conditions statiques: Si un nom de procédure est régional (voir section 3.2.2), sa définition de procédure ne peut pas spécifier *GENERAL*.

Si un nom de procédure est critique (voir section 8.2), 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, le *nom de procédure* optionnel devant le point-virgule doit être égal au *nom* devant le *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 synchronisation.

exemples:

```
1.3     add:
        PROC(i,j INT) (INT) EXCEPTIONS(OVERFLOW);
           RESULT i+j;
        END add;                                          (1.1)
```

7.5 DEFINITIONS DE PROCESSUS

syntaxe:

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

<définition de processus> ::= (2)
PROCESS ([<liste de paramètres formels>]);

<corps de processus> END *(2.1)*

sémantique: Une définition de processus définit une séquence d'actions éventuellement paramétrée, qui peut être démarrée pour exécution parallèle, de différents endroits du programme. (voir chapitre 8).

propriétés statiques: Un nom est un nom de processus si et seulement si il est défini dans un *énoncé de définition de processus*. (c.-à-d. placé devant un deux points et une *définition de processus*).

A un nom de processus peut être attaché un ensemble de noms d'exception défini par l'implémentation.

conditions statiques: Si présent, le *nom de processus* optionnel devant le point-virgule doit être égal au *nom* devant la *définition de processus*.

Un *énoncé de définition de processus* ne peut pas être englobé par une région, ni par un bloc autre que la définition de processus imaginaire la plus externe (voir section 7.8).

Les attributs de paramètres dans la *liste de paramètres formels* ne peuvent pas être *INOUT* ou *OUT*.

Seulement si *LOC* est spécifié dans la *spec de paramètre* dans un *paramètre formel* dans la *liste de paramètres formels*, le mode contenu peut avoir la propriété de synchronisation.

exemples:

```
14.12   PROCESS();
          DO FOR EVER:
            WAIT(10 /* seconds */);
            CONTINUE OPERATOR_IS_READY;
          OD;
        END                                              (2.1)
```

7.6 MODULES

syntaxe:

<module> ::= *(1)*
MODULE <corps de module> END *(1.1)*

sémantique: Un module est une action qui peut contenir des définitions et déclarations locales. Un module est un moyen de restreindre la visibilité des noms; 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 9.2.

propriétés statiques: Un nom est un nom de module si et seulement si il est défini en le plaçant devant un deux points avant *MODULE*.

exemples:

```
7.42    MODULE
          SEIZE convert;
          DCL n INT INIT := 1979;
          DCL rn CHAR(20) INIT :=(20)' ';
          GRANT n, rn;
          convert();
          ASSERT rn = 'MDCCCCLXXVIIII'//(6)' ';
        END                                                    (1.1)
```

7.7 REGIONS

syntaxe:

<région> ::= (1)
[<nom> :] REGION <corps de région> END
[<filet>] [<nom de région>]; (1.1)

sémantique: Une région est un moyen de réaliser l'exclusion mutuelle, pour accéder aux objets informatifs créés localement, lors de l'exécution parallèle des processus (voir chapitre 8). Elle détermine la visibilité des noms créés localement de la même manière qu'un module.

propriétés statiques: Un nom est un nom de région si et seulement si il est défini en le plaçant devant un deux points devant *REGION*.

conditions statiques: Le *nom de région* optionnel avant le point-virgule doit être égal au *nom* de la région.

Une *région* ne peut pas être englobée par un bloc autre que la définition de processus imaginaire la plus externe.

examples:

voir 13.1 - 13.25

7.8 PROGRAMMES

syntaxe:

<programme> ::= (1)
{<énoncé d'action module> | <région>}+ (1.1)

sémantique: Les programmes consistent en une liste de modules ou de régions, englobés par une définition de processus imaginaire la plus externe. Cette définition de processus est considérée contenir dans son domaine, un module prélude standard CHILL. Ce module contient les définitions des noms prédéfinis, des opérations prédéfinies, des modes prédéfinis et des noms de registre prédéfinis.

propriétés statiques: Les noms définis par le langage et par l'implémentation (voir Appendice C2) sont considérés être créés dans un module placé dans le domaine de la définition de processus imaginaire la plus externe et octroyés *PERVASIVE* par ce module (voir section 9.2.6.2).

7.9 ALLOCATION DE MEMOIRE ET DUREE 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 à l'opération prédéfinie *GETSTACK*.

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, 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 la 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 locus seulement existera pour toutes les invocations ou activations.

La durée de vie d'un locus créé en exécutant l'appel à l'opération prédéfinie *GETSTACK* est le temps entre cette exécution et le départ du bloc englobant du plus près. Si l'appel à l'opération prédéfinie *GETSTACK* est exécuté pendant qu'on évalue un paramètre effectif d'un appel de procédure ou d'une expression démarrer, la durée de vie du locus créé sera celle de l'appel de procédure ou la durée de vie du processus créé.

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 englobé par un bloc autre que la définition de processus imaginaire.

- Un *nom de loc-identité* tel que le *locus de mode statique* dans sa définition est statique.

- Un *élément de chaîne* ou *sous-chaîne* où le *locus chaîne* est statique et soit l'*élément de gauche* et l'*élément de droite*, soit la *position* qui s'y trouvent sont constants.

- Un *élément de rangée* ou *sous-rangée* où le *locus rangée* est statique et soit l'*expression*, ou l'*élément inférieur* et l'*élément supérieur*, soit l'*expression entière* qui s'y trouvent, sont constants.

- Un *champ de structure* où le *locus structure* est statique. Si le *locus structure* n'est pas un *locus structure paramétré*, le *nom de champ* ne doit pas être un *nom de champ récurrent*.

- Une *conversion de locus* où le *locus* qui s'y trouve, est statique.

8.0 EXECUTION CONCURRENTE

8.1 LES PROCESSUS ET LEURS DEFINITIONS

Un processus est l'exécution séquentielle d'une série d'énoncés, dont l'exécution séquentielle peut se faire en parallèle avec d'autres processus. Le comportement d'un processus est décrit par une définition de processus (voir section 7.5), 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 section 5.2.17). Il devient actif (c.-à-d. en exécution) et 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 paralléle. 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 section 7.8), 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 section 7.8) 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 dans l'expression démarrer.

Un processus est terminé par l'exécution d'une action arrêter ou en atteignant la fin du corps de processus ou la fin d'un choix d'exceptions du filet qui termine 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 ses processus subsidiaires (c.-à-d. créés par des expressions démarrer qu'il contient) seront terminés.

Un processus est, au niveau du programme CHILL, toujours dans l'un de 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 de en attente à actif est appelée la réactivation du processus.

8.2 EXCLUSION MUTUELLE ET REGIONS

8.2.1 GENERALITES

Les régions (voir section 7.7) 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 section 8.2.2) 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) si et seulement si il est défini à l'intérieur d'une région et octroyé par la région, ou si, un nom de procédure avec la même définition de procédure (voir section 7.4) est critique (ce qui n'a de sens que si des définitions d'entrée sont employées).

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 parallèle) 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 section 8.3). Dans le cas d'appels à des procédures critiques imbriquées dynamiquement, seule la région verrouillée le plus tard sera libérée.

Si, pendant qu'une région est verrouillée, un processus essaye d'appeler une de ses procédures critiques ou essaye d'entamer la région, le processus qui essaye 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'entamer la région ou d'appeler une de ses procédures critiques ou d'être réactivé dans une de ses procédures critiques, un processus seulement sera sélectionné pour entamer la région suivant un algorithme de sélection défini par l'implémentation.

8.2.2 REGIONALITE

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 section adéquates (action d'affectation, appel de procédure, action envoyer, action résulter);
- les procédures régionales ne sont pas générales (voir section 7.4).
- les procédures critiques ne sont ni générales, ni récursives (voir section 7.3.).

Un *locus*, *valeur* ou *nom de procédure* peut être régional. Cette propriété se définit comme suit:

1. Locus

   Un *locus* est régional 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* et qui n'est pas défini dans un *paramètre formel* d'une procédure critique,
     - soit un *nom de loc-identité*, dans la définition duquel le *locus de mode statique est* régional ou qui est défini dans le *paramètre formel* d'une procédure régionale,
     - soit un *nom basé* dans la définition duquel le *nom de locus repère lié ou libre* est régional,
     - soit un *nom d'énumération de locus*, dont le *locus rangée* dans l'*action faire* associée est régional,
     - soit un *nom de locus faire-avec* dont le *locus structure* dans l'*action faire* associée est régional.
   - C'est un *repère lié dérepéré*, contenant une *expression repère lié* qui est régionale.
   - C'est un *repère libre dérepéré*, contenant une *expression repère libre* qui est régionale.
   - C'est un *descripteur dérepéré*, contenant une *expression descripteur* qui est régionale.

- C'est un *élément de rangée*, une *sous-rangée* ou une *tranche de rangée* contenant un *locus rangée* qui est régional.

- C'est un *élément de chaîne*, une *sous-chaîne* ou *une tranche de chaîne* contenant un *locus chaîne* qui est régional.

- C'est un *champ de structure* contenant un *locus structure* qui est régional.

- 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 régional.

- C'est un *appel d'opération prédéfinie rendant locus*, que l'implémentation spécifie être régional.

- C'est une *conversion de locus* contenant un *locus de mode statique* qui est régional.

2. Valeur

Une *valeur* ou *expression* est régionale si et seulement si elle est soit une *valeur primitive* qui est régionale soit une *expression parenthésée* contenant une *expression* qui est régionale.

Une *valeur primitive* est régionale si et seulement si une des conditions suivantes est satisfaite:

- C'est un *contenu de locus* qui est régional et dont le mode a la propriété de repérer.

- C'est un *nom de valeur* qui est:

  - soit un *nom de synonyme* dans la définition duquel la *valeur constante* est régionale;

  - soit un *nom de valeur faire-avec* dont l'*expression structure* dans l'*action faire* associée est régionale et dont le mode a la propriété de repérer.

- C'est un *multiplet* contenant un *multiplet de rangée* ou *multiplet de structure* dans lequel au moins une occurrence de *valeur* est régionale.

- C'est une *valeur élément de rangée*, une *valeur sous-rangée*, ou une *valeur tranche de rangée* contenant une *expression rangée* qui est régionale et dont le mode des éléments du mode de l'*expression rangée* a la propriété de repérer.

- C'est une *valeur champ de structure* contenant une *expression structure* qui est régionale et dont le mode du champ a la propriété de repérer.

- C'est un *locus repéré* contenant un *locus* qui est régional.

- C'est une *conversion d'expression* contenant une *expression* qui est régionale.

- C'est un *appel de procédure rendant valeur* où dans l'*appel de procédure rendant valeur* un *nom de procédure* est spécifié qui est régional et dont le mode de la spec de résultat a la propriété de repérer.

- C'est un *appel d'opération prédéfinie rendant valeur*, qui est soit un *appel d'opération prédéfinie par l'implémentation rendant valeur* qui rend une valeur dont la classe est compatible avec un mode qui a la propriété de repérer et que l'implémentation spécifie être régional, soit *ADDR(<locus>)*, où *<locus>* est régional.

3. Nom de procédure

Un *nom de procédure* est régional si et seulement si il est défini à l'intérieur d'une région et il n'est pas critique (c.-à-d. pas octroyé par la région).

8.3 MISE EN ATTENTE D'UN PROCESSUS

Quand un processus est actif, il peut être mis en attente en exécutant ou en évaluant l'une des actions ou l'une des expressions suivantes:

Action mettre en attente (voir section 6.16). Quand un processus exécute une action mettre en attente, il est mis en attente. Il devient un des membres, avec la priorité spécifiée, de l'ensemble des processus mis en attente qui est attaché au locus événement spécifié.

Action mettre en attente et choisir (voir section 6.17). Quand un processus exécute une action mettre en attente et choisir, il est mis en attente. Il devient un des membres, avec la priorité spécifiée, de chaque ensemble de processus mis en attente qui est attaché à un locus événement spécifié dans un événement à choisir de l'action mettre en attente et choisir.

Expression recevoir (voir section 5.2.18). Quand un processus évalue une expression recevoir, il est mis en attente si et seulement si il n'y a pas de valeurs dans, ni de processus envoyants en attente sur, le locus tampon spécifié. Il devient un des membres d'un ensemble de processus recevants mis en attente attaché au locus tampon (vide) spécifié.

Action recevoir tampon et choisir (voir section 6.19.3). Quand un processus exécute une action recevoir tampon et choisir, il est mis en attente si et seulement si il n'y a de valeurs dans aucun des locus tampon spécifiés, ni de processus envoyants en attente sur aucun des locus tampons spécifiés, et si *ELSE* n'est pas spécifié. Il devient un membre de

chaque ensemble de processus recevants mis en attente attaché à un locus tampon spécifié dans un tampon à choisir de l'action recevoir tampon et choisir.

Action recevoir signal et choisir (voir section 6.19.2). Quand un processus exécute une action recevoir signal et choisir, il est mis en attente si et seulement si aucun signal qui peut être reçu par le processus qui exécute l'action recevoir signal et choisir n'est suspendu et seulement si *ELSE* n'est pas spécifié. Le processus devient un membre de chaque ensemble de processus mis en attente attaché à un nom de signal spécifié dans un signal à choisir.

Action envoyer tampon (voir section 6.18.3). Quand un processus exécute une action envoyer tampon, il est mis en attente si et seulement si le mode du locus tampon a une longueur et que le nombre de valeurs dans le locus tampon est égal à la longueur juste avant l'opération d'envoi. Le processus devient un membre, avec la priorité spécifiée, de l'ensemble des processus envoyants en attente attaché au locus tampon.

Quand un processus exécute une action qui provoque sa mise en attente pendant que le contrôle réside dans une procédure critique, la région associée sera libérée. Le contexte dynamique de la procédure sera retenu jusqu'à ce que le processus soit réactivé à l'endroit de sa mise en attente dans la région. La région sera alors à nouveau verrouillée.

8.4 REACTIVATION D'UN PROCESSUS

Quand un processus est en attente, il est réactivé si et seulement si un autre processus exécute une des actions suivantes:

Action continuer (voir section 6.15). Quand un processus exécute une action continuer, il réactive un autre processus si et seulement si l'ensemble des processus en attente du locus événement spécifié n'est pas vide. Un processus avec la priorité la plus haute est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de tous les ensembles de processus en attente.

Action envoyer tampon (voir section 6.18.3). Si un processus exécute une action envoyer tampon, il réactive un autre processus si et seulement si l'ensemble des processus recevants en attente du locus tampon spécifié n'est pas vide. Un processus est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de tous les ensembles de processus en attente. Si l'ensemble des processus recevants en attente du locus tampon est vide, la valeur envoyée est mise dans le tampon accompagnée de sa priorité spécifiée si la capacité du tampon le permet (voir section 8.3).

Action envoyer signal (voir section 6.18.2). Quand un processus exécute une action envoyer signal, il réactive un autre processus si et seulement si l'ensemble des processus en attente du nom de signal spécifié contient

un processus qui peut recevoir le signal. Un processus est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de tous les ensembles de processus en attente. Si aucun processus en attente n'est prêt à recevoir le signal, le signal est suspendu avec sa priorité spécifiée, sa liste de valeurs, son nom de processus et/ou valeur exemplaire.

Action recevoir tampon et choisir (voir section 6.19.3). Quand un processus exécute une action recevoir tampon et choisir, il réactive un autre processus si et seulement si l'ensemble des processus envoyants en attente sur n'importe lequel des locus tampon spécifiés n'est pas vide. Dans ce cas il reçoit une valeur de la plus haute priorité parmi les valeurs dans le locus tampon, ou bien des processus envoyants en attente. En recevant une valeur d'un tampon, le processus enlève la valeur du tampon et un processus envoyant en attente qui a la valeur de la plus haute priorité est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de tous les ensembles de processus envoyants en attente et sa valeur est mise dans le tampon avec la priorité spécifiée. En recevant une valeur directement d'un processus envoyant en attente, le processus en attente qui porte la valeur de la plus haute priorité est sélectionné et devient actif suivant un algorithme de sélection défini par l'implémentation. Ce processus réactivé est alors enlevé de tous les ensembles de processus envoyants en attente et sa valeur est reçue.

Quand un processus exécute une action qui provoque la réactivation d'un autre processus, tandis que le processus qui réactive est dans une procédure critique, le processus qui réactive reste actif, c.-à-d. ne libèrera pas la région à cet endroit.

8.5 ENONCE DE DEFINITION DE SIGNAL

syntaxe:

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

<définition de signal> ::= (2)
<nom> [= (<mode> {,<mode>})]*
[TO <nom de processus>]

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 signal et choisir, les valeurs sont gardées jusqu'a ce qu'un processus les reçoive.

propriétés statiques: Un nom est un nom de signal si et seulement si il est défini dans une *définition 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 synchronisation.

exemples:

```
15.16   SIGNAL INITIATE = (INSTANCE),
                TERMINATE;                                    (1.1)
```

9.0 PROPRIETES SEMANTIQUES GENERALES

9.1 VERIFICATION DE MODES

9.1.1 PROPRIETES DES MODES ET DES CLASSES

9.1.1.1 Nouveauté

Informel

La nouveauté d'un mode indique si le mode est défini via un énoncé de définition de neumode ou pas. La nouveauté d'un mode est soit nil, c.-à-d. c'est un mode (de base) qui n'est pas défini via un neumode, soit le nom de neumode via lequel il est défini.

Définition

La nouveauté d'un mode est définie de la manière suivante:

- si le mode est dénoté par un *nom de neumode*, sa nouveauté est ce nom de neumode, sinon,
- si le mode est dénoté par un *nom de synmode*, sa nouveauté est la nouveauté du *mode définissant*, dans sa définition, sinon,
- si le mode est dénoté par un *mode rangée paramétré*, *mode chaîne paramétré* ou *mode structure paramétré*, sa nouveauté est respectivement la nouveauté du *nom de mode rangée originel*, *nom de mode chaîne originel*, *nom de mode structure variable originel* contenu, sinon,
- si le mode est dénoté par un *mode intervalle*, sa nouveauté est la nouveauté de son mode parent, sinon,
- si le mode est dénoté par un mode parent virtuellement introduit, sa nouveauté est le nom de neumode qui a causé son introduction (voir section 3.2.3), sinon,
- si le mode est dénoté par *READ <mode>*, sa nouveauté est celle de *<mode>*,
- sinon, sa nouveauté est nil.

9.1.1.2 Modes protégés

Informel

Un mode est dit être protégé si un locus de ce mode, en tant que tout, est protégé, c.-à-d. on ne peut altérer le contenu ni du locus ni d'aucune de ses parties.

Définition

Un mode a la propriété héréditaire suivante: c'est un mode protégé si et seulement si une des conditions suivantes est remplie:

- Il est dénoté par un *mode* qui est de la forme *READ <mode>*.
- Il est dénoté par un *mode rangée paramétré*, un *mode chaîne paramétré*, ou *un mode structure paramétré*, contenant respectivement un *nom de mode rangée originel*, un *nom de mode chaîne originel*, ou un *nom de mode structure variable originel* qui dénote un mode protégé.

9.1.1.3 Propriété de protection

Informel

Un mode a la propriété de protection si un locus de ce mode est protégé ou contient un composant, sous-composant, etc. qui est protégé.

Définition

Un mode a la propriété de protection si et seulement si une des conditions suivantes est remplies:

- Le mode est un nom de mode, défini par un mode qui a la propriété de protection.
- Le mode est un mode rangée avec un mode des éléments qui a la propriété de protection ou un mode structure dont au moins un des modes de champ a la propriété de protection.
- Le mode est un mode protégé.

9.1.1.4 Propriété de repérer

Informel

Un mode a la propriété de repérer si un locus de ce mode a un mode repère ou contient un composant ou un sous-composant, etc. qui a un mode repère.

Définition

Un mode a la propriété de repérer si et seulement si une des propriétés suivantes est remplie:

- Le mode est un nom de mode défini par un mode qui a la propriété de repérer.
- Le mode est un mode rangée dont le mode des éléments a la propriété de repérer ou un mode structure dont au moins un des modes de champ a la propriété de repérer.
- Le mode est un mode repère.

9.1.1.5 Propriété de marquage et de paramétrage

Informel

Un mode a la propriété de marquage et de paramétrage si un locus de ce mode a un mode structure paramétré avec marqueurs ou contient un composant ou un sous-composant, etc. qui a un mode structure paramétré avec marqueurs.

Définition

Un mode a la propriété de marquage et de paramétrage si et seulement si une des conditions suivantes est remplie:

- Le mode est un nom de mode défini par un mode qui a la propriété de marquage et de paramétrage.
- Le mode est un mode rangée dont le mode des éléments a la propriété de marquage et de paramétrage ou un mode structure dont au moins un des modes de champ a la propriété de marquage et de paramétrage.
- Le mode est un mode structure paramétré avec marqueurs.

9.1.1.6 Propriété de synchronisation

Informel

Un mode a la propriété de synchronisation si un locus de ce mode a un mode de synchronisation ou contient un composant ou un sous-composant etc. qui a un mode de synchronisation.

Définition

Un mode a la propriété de synchronisation si et seulement si une des conditions suivantes est remplie:

- Le mode est un nom de mode défini par un mode qui a la propriété de synchronisation.
- Le mode est un mode rangée dont le mode des éléments a la propriété de synchronisation ou un mode structure dont au moins un des modes de champ a la propriété de synchronisation.
- Le mode est un mode événement ou un mode tampon.

9.1.1.7 Mode racine

Toute M-classe par valeur ou M-classe par dérivation où M n'est pas un mode composé, a un mode racine défini de la façon suivante:

- Si M n'est pas un mode intervalle, le mode racine est M.
- Si M est un mode intervalle alors le mode racine est le mode parent de M.

9.1.1.8 Classe résultante

Etant données deux classes compatibles (voir section 9.1.2.6), qui sont soit la classe toute, soit une M-classe par valeur, soit une M-classe par dérivation, où M est soit un mode discret, un mode ensembliste, ou un mode chaîne, la classe résultante est définie comme:

- la classe résultante de la M-classe par dérivation et de la N-classe par dérivation est la M-classe par dérivation;
- la classe résultante de la M-classe par valeur et de la N-classe par dérivation, est, si M n'est pas un mode intervalle, la M-classe par valeur, sinon la P-classe par valeur, où P est le mode parent de M;
- la classe résultante de la M-classe par valeur et de la N-classe par valeur est, si M n'est pas un mode intervalle, alors la M-classe par valeur, sinon la P-classe par valeur, où P est le mode parent de M;
- la classe résultante de la classe toute et de n'importe quelle autre classe est cette seconde.

Etant donnée 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.

(A noter que CHILL est défini de manière à ce que l'ordre dans lequel on prend les classes C_i est indifférent, c.-à-d. toutes les classes résultantes ainsi obtenues sont compatibles.)

9.1.2 RELATIONS ENTRE MODES ET CLASSES

Dans les sections suivantes, 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 de quelques autres relations, qui sont employées principalement dans le chapitre 9 dans ce but.

9.1.2.1 La relation "défini par"

Informel

Un nom de mode est dit être défini par son mode définissant, et, transitivement, si ce dernier est aussi un nom de mode, le premier est aussi défini par le mode définissant de son mode définissant, etc.

Définition

Un nom de mode N est dit être défini par un mode M si et seulement si:

- M est le mode définissant de N, ou
- le mode définissant de N est un nom de mode défini par M.

9.1.2.2 Relations d'équivalence sur les modes

Généralités

Informel

Les relations d'équivalence suivantes jouent un rôle dans la formulation des relations de compatibilité:

- Deux modes sont dits être similaires s'il sont de la même sorte, c.-à-d. s'ils ont les mêmes propriétés héréditaires.

- Deux modes sont dits être v-équivalents (équivalents en valeur) s'ils sont similaires et ont aussi la même nouveauté.

- Deux modes sont dits être 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 dits être l-équivalents (équivalents en locus) s'ils sont équivalents et ont la même spécification de protection.

Définition

Dans les sections suivantes, on donne les relations d'équivalence entre modes sous la forme d'un ensemble de relations (partielles). L'algorithme d'équivalence complet est obtenu 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.)

La relation "similaire"

Deux modes sont similaires si et seulement si une des conditions suivantes est remplie:

- ce sont des modes entier;

- ce sont des modes booléen;

- ce sont des modes caractère;

- ce sont des modes ensemble qui définissent le même nombre de valeurs, les mêmes ensembles de noms d'élément d'ensemble, et tels que pour les mêmes noms l'appel à l'opération prédéfinie *NUM* donne le même résultat;

- 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;

- l'un est un mode booléen et l'autre un mode chaîne de bits de longueur 1;

- l'un est un mode caractère et l'autre un mode chaîne de caractères de longueur 1;

- ce sont des modes ensembliste dont les modes primitifs sont équivalents;

- ce sont des modes repère lié tels que leurs mode repérés sont équivalents;

- ce sont des modes repère libre;

- ce sont des modes descripteur tels que leurs modes repérés originels sont équivalents;

- ce sont des modes procédure tels que:

  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 et, si présentes, les mêmes spécifications de registres;

  2. les deux ont ou n'ont pas une spec de résultat. Si présentes, les deux specs de résultat ont des modes l-équivalents, les mêmes attributs et, si présentes, les mêmes spécifications de registres;

  3. ils ont les mêmes ensembles de noms d'exceptions;

  4. ils ont la mêmes récursivité;

- ce sont des modes exemplaire;

- ce sont des modes événement tels que ils ont soit la même longueur, soit n'en ont pas;

- ce sont des modes tampon tels que:

  1. les deux n'ont pas de longueur ou ont la même;

  2. ils ont des modes des éléments de tampon qui sont l-équivalents;

- ce sont des modes chaîne tels que:

  1. ce sont tous les deux des modes chaîne de bits ou chaîne de caractères;

  2. ils ont la même longueur. Cette vérification est dynamique si un ou les deux modes est (sont) dynamique(s). Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL*;

- ce sont des modes rangée tels que:

  1. leurs modes d'indice sont v-équivalents;

2. leurs modes des éléments sont équivalents;

3. leurs implantations d'élément sont équivalentes (voir section 9.1.2.2);

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 tels que:

  1. ils ont le même nombre de champs et les champs correspondants (par position) sont équivalents (voir section 9.1.2.2);

  2. si ce sont tous les deux des modes structure variable paramétrables, leurs listes des classes sont compatibles;

- ce sont des modes structure paramétrés tels que:

  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 résultera en l'exception *TAGFAIL*.

La relation "v-équivalent"

Deux modes sont v-équivalents si et seulement s'ils sont similaires et ont la même nouveauté.

La relation "équivalent"

Deux modes sont équivalents si et seulement s'ils sont v-équivalents et:

- si l'un est un mode booléen, l'autre l'est aussi;
- si l'un est un mode caractère, l'autre l'est aussi;
- 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.

La relation "l-équivalent"

Deux modes sont l-équivalents si et seulement s'ils sont équivalents et, si l'un a la propriété de protection, l'autre l'a aussi, et:

- si les deux sont des modes repère lié, leurs modes repérés doivent être l-équivalent;

- 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 les deux sont des modes structure, les champs correspondants (par position) doivent être l-équivalents.

La relation "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, tous deux correspondants, de la manière suivante:

1. Champs fixes et champs récurrents

   a. Les deux champs doivent avoir des implantations équivalentes.

   b. Les modes de deux champs doivent être 1. équivalents, 2. l-équivalents.

2. Choix de champs

   a. Les deux choix de champs ont des marqueurs ou les deux n'en ont pas. Dans le premier cas, les marqueurs doivent avoir le méme nombre de noms de champ marqueur et les noms de champ marqueur correspondants (par position) doivent dénoter des champs fixes correspondants.

   b. Les deux doivent avoir le même nombre de choix de champs et les choix de champs correspondants (par position) doivent être 1. équivalents, 2. l-équivalents.

   c. 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 à choisir doit suivre et les champs à choisir correspondants (par position) doivent être 1. équivalents, 2. l-équivalents.

3. Champs à choisir

a. 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 *indifférent*, ou toutes les deux *(ELSE)*, ou toutes les deux définir le même ensemble de valeurs.

b. Les deux champs à choisir doivent avoir le même nombre de champs récurrents et des champs récurrents correspondants (par position) doivent être 1. équivalents, 2. 1-équivalents.

La relation équivalent pour les implantations

Dans la suite, il sera supposé que chaque *pos* est de la forme:
POS(<mot> , <bit initial> , <longueur>)

et que chaque *pas* est de la forme:
STEP(<pos> , <taille de pas> , <taille de patron>)

La section 3.10.6 donne les règles appropriées pour donner à *pos* ou *pas* la forme désirée.

1. 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).

2. 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 le dernier cas, le *pos* dans le premier *pas* doit être équivalent au *pos* dans le second *pas* (voir plus loin) et *NUM(taille de pas)* doit donner la même valeur pour les deux implantations d'élément et *NUM(taille de patron)* doit donner la même valeur pour les deux implantations d'élément.

3. Pos

   Un *pos* est équivalent à un autre *pos* si et seulement si les deux occurrences de *NUM(mot)* donnent la même valeur, les deux occurrences de *NUM(bit initial)* donnent la même valeur, et les deux occurrences de *NUM(longueur)* donnent la même valeur.

9.1.2.3 La relation "compatible en lecture"

Informel

Un mode M est dit être compatible en lecture avec un mode N, si et

seulement si M et N sont équivalents et M et ses (sous)-composants possibles ont des spécifications de protection plus restrictives. De ce fait, la relation est asymmétrique.

Exemple:
READ REF READ CHAR est compatible en lecture avec *REF CHAR*

Définition

Un mode M est dit être compatible en lecture avec un mode N (une 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 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 mode 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, chaque mode de champ de M doit être compatible en lecture avec le mode de champ correspondant de N.

9.1.2.4 La relation "limitable à"

Informel

La relation "limitable à" est applicable à des modes équivalents qui ont la propriété de repérer. Un mode M est dit être limitable à un mode N si lui ou ses possibles sous-composants repèrent des locus qui ont des spécifications de protection égales ou moins restrictives que ceux repérés par N. De ce fait cette relation est asymétrique. La relation est employée pour les affectations (voir section 9.1.2.5).

Exemple:
REF INT est limitable à *REF READ INT*
STRUCT(Q REF BOOL) est limitable à *STRUCT(Q REF READ BOOL)*

Definition

Un mode M est limitable à un mode N (une relation asymétrique) si et seulement si l'une des conditions suivantes est remplie:

- M a la propriété de repérer et M est équivalent à N.
- M et N sont des modes repère lié et le mode repéré de N est compatible en lecture avec le mode repéré de M.

- M et N sont des modes repère libre et M et N sont équivalents.

- M et N sont des modes descripteur et le mode repéré originel de N est compatible en lecture avec le mode repéré originel de M.

- M et N sont des modes rangée et le mode des éléments de M est limitable au mode des éléments de N.

- M et N sont des modes structure et chaque mode de champ de M est limitable au mode de champ correspondant de N.

9.1.2.5 Compatibilité entre un mode et une classe

- tous les modes sont compatibles 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 satifaite:

  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 descripteur dont le mode repéré originel est appelé V et:

     - si V est un mode chaîne, N doit être un mode chaîne tel que V(p) est compatible en lecture avec N, où p est la longueur (éventuellement dynamique) de N;

     - si V est un mode rangée, N doit être un mode rangée tel que V(p) est compatible en lecture avec N, où p est la borne supérieure (peut être dynamique) de N;

     - si V est un mode structure variable, N doit être un mode structure paramétré tel que $V(p_1,\ldots,p_n)$ est compatible en lecture avec N, où $p_1,\ldots,p_n$ dénote la liste de valeurs de 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 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, N doit être limitable à M.

9.1.2.6 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 dérivation ou la N-classe par valeur si et seulement si N est un mode repère et l'une des conditions suivantes est remplie:

  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 original est appelé V et:
     - si V est un mode chaîne, M doit être un mode chaîne tel que V(p) est équivalent à M, où p est la longueur (éventuellement dynamique) de M;
     - si V est un mode rangée, M doit être un mode rangée tel que V(p) est équivalent à M, où p est la borne supérieure (éventuellement dynamique) de M;
     - si V est un mode structure variable, M doit être un mode structure paramétré tel que $V(p_1,\ldots,p_n)$ est équivalent à M, où $p_1,\ldots,p_n$ dénote la liste de valeurs de N.

- 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.

9.1.3 *SELECTION 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)
   | (ELSE) | (<indifférent>)                               (2.2)

<étiquette de cas> ::=                                     (3)
     <expression littérale discrète>                       (3.1)
   | <intervalle littéral>                                 (3.2)
   | <nom de mode discret>                                 (3.3)

<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 section 3.10.4) auquel cas une liste de champs récurrents est sélectionnée,
- des multiplets de rangée avec indice (voir section 5.2.5) auquel cas une valeur élément de rangée est sélectionnée,
- des actions de cas (voir section 6.4) auquel cas une liste d'énoncés d'action est sélectionnée.

Dans la première et dernière situation, chaque alternative est étiquetée avec une spécification d'étiquettes de cas; pour le multiplet de rangée avec indice, chaque valeur est étiquetée avec une liste d'étiquettes de cas. Pour faciliter l'explication, la liste d'étiquettes de cas pour le multiplet de rangée avec indice sera considérée dans cette section comme une spécification d'étiquettes de cas réduite à seulement une liste d'étiquettes de cas.

La sélection de cas sélectionne l'alternative qui est étiquetée par la spécification d'étiquette 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,

- la liste d'étiquettes de cas consiste en *(ELSE)* et la valeur est une des valeurs implicitement indiquée par *(ELSE)*, ou

- la liste d'étiquettes de cas consiste en *indifférent*.

Les valeurs indiquées explicitement par une étiquette de cas sont les valeurs de toute *expression 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 indice*, 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 *champs 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*.

- A une *liste d'étiquettes de cas* on attache une classe qui est, s'il s'agit de *(ELSE)* ou *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 de classes résultantes (pourvu que les spécifications d'étiquettes de cas aient le même nombre de classes; ce qui sera toujours le cas). Cette liste de classes résultantes est formée en formant, pour chaque position dans 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 section 3.10.4).

- 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 où la classe du sélecteur correspondant est la M-classe par valeur ou la M-classe par dérivation, c'est l'ensemble des valeurs défini par M.

conditions statiques: Pour chaque *spécification d'étiquettes de cas*, le nombre d'occurrences de *liste d'étiquettes de cas* doit être le même.

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 cohérents, c.-à-d. chaque liste de valeurs des sélecteurs possible ne corréspond qu'a une spécification d'étiquettes de cas.

exemples:

```
11.7     (occupied)                    (3.1)
11.62    (rook),(*)                    (1.1)
8.24     (ELSE)                        (2.2)
```

9.1.4 *DEFINITION ET RESUME DES CATEGORIES SEMANTIQUES*

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.

9.1.4.1 Noms

Noms de mode

nom de mode:	voir section 3.2.1.
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 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 variable.

nom de mode tampon: un *nom* défini par un mode tampon.

nom de neumode: voir section 3.2.3.

nom de synmode: voir section 3.2.2.

Noms d'accès

nom basé: voir section 4.1.4.

nom de loc-identité: voir sections 4.1.3, 7.4.

nom de locus: voir sections 4.1.2, 7.4.

nom de locus faire-avec: voir section 6.5.4.

nom d'énumération de locus: voir section 6.5.2.

Noms de valeur

nom d'énumeration de valeur: voir section 6.5.4.

nom de synonyme: voir section 5.1.

nom de valeur faire-avec: voir section 6.5.2.

nom de valeur reçue: voir sections 6.19.2, 6.19.3.

Noms divers

liste de noms réservés: une *liste de noms* ne contenant que des noms réservés (voir Appendice C1).

nom de champ: voir section 3.10.4.

nom de champ marqueur: voir section 3.10.4.

nom d'élément d'ensemble: voir section 3.4.5.

nom de locus repère lié ou libre: un *nom de locus* dont le mode est un mode repère lié ou un mode repère libre.

nom de module: voir section 7.6.

nom de procédure: voir section 7.4.

nom de procédure générale: un *nom de procédure* dont la généralité est générale.

nom de processus: voir section 7.5.

nom de région: voir section 7.7.

nom de registre: un *nom* défini par l'implémentation dénotant un registre de la machine.

nom de signal: voir section 3.5.2.

nom de synonyme indéfini: voir section 5.1.

nom d'étiquette: voir section 6.1.

nom d'opération prédéfinie: un *nom* défini par l'implémentation dénotant une opération prédéfinie par l'implémentation.

nom non réservé: un *nom* qui n'est aucun des noms réservés mentionnés à l'Appendice C1.

9.1.4.2 Locus

locus chaîne: un *locus* qui a un mode chaîne.
locus événement: un *locus* qui a un mode événement.
locus exemplaire: un *locus* qui a un mode exemplaire.
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.

9.1.4.3 Expressions

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

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

expression descripteur: une *expression* 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.

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

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

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

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

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

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

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

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

conditions statiques: Ni une *expression* *booléenne* ni une *expression* *discrète* (quand indiqué dans la syntaxe) ne peuvent avoir une classe dynamique. C.-à-d. la vérification de ce que la classe de l'*expression* est compatible avec un mode booléen ou mode discret, peut se faire statiquement.

9.1.4.4 Catégories sémantiques diverses

appel d'opération prédéfinie par l'implémentation rendant valeur: voir section 11.1.3.

appel de procédure rendant locus: voir section 6.7.

appel de procédure rendant valeur: voir section 6.7.

caractère différent d'apostrophe: un *caractère* qui n'est pas une apostrophe.

énoncé d'action module: un *énoncé d'action* dans lequel l'*action* immédiatement contenue est un *module*.

9.2 VISIBILITE ET IDENTIFICATION

9.2.1 GENERALITES

Les constructions spécifiques de CHILL mentionnées dans la section 7.1. créent de nouveaux noms dans un programme. Les énoncés de structuration du programme et les énoncés de visibilité déterminent la visibilité des noms dans l'ensemble du programme. Cette section traite de la visibilité des noms à l'exception des noms d'exceptions, c.-à-d. chaque nom est considéré n'être pas dans le contexte d'un nom d'exception. Pour les noms d'exception, voir le chapitre 10.

Pour permettre une description précise de la structure de visibilité d'un programme, on introduit, juste pour cette section 9.2., les raffinements de terminologie suivants:

- Une chaîne de nom (d'un nom) est une chaîne de caractères (employée comme dénotation pour le nom) vue comme un élément lexical en dehors de tout contexte. Un nom est une chaîne de nom associée à une définition (occurrence de définition, voir section 9.2.2.) de cette chaîne de nom.

Exemple:

```
B : BEGIN
   MODULE DCL I INT ; END ;
   MODULE DCL I PTR ; END ;
   END ;
```

Dans le corps de bloc du bloc début-fin étiqueté *B*, deux noms sont introduits, chacun avec la chaîne de nom *I*.

Dans un domaine, chaque nom possède l'un des quatre degrés de visibilité suivants:

Table 1: Degrés de visibilité

Visibilité	Propriétés (informel)
Directement fortement visible	Le nom est visible par création, octroi ou saisie
Indirectement fortement visible	Le nom est hérité via une imbrication de bloc ou son attribut d'envahissement
Faiblement visible	Le nom est impliqué par un nom fortement visible
Invisible	Le nom ne peut pas être utilisé

Un nom est dit être fortement visible s'il est soit directement fortement visible soit indirectement fortement visible. Un nom est dit être visible s'il est soit faiblement, soit fortement visible, autrement, le nom est dit être invisible. Les énoncés de structuration du programme et les énoncés de visibilité déterminent d'une façon univoque à quelle classe de visibilité chaque nom appartient. Les propriétés précises des classes de visibilité sont expliquées dans les sections suivantes.

L'identification est le mécanisme par lequel on associe un nom unique à chaque chaîne de nom, c.-à-d. on associe une signification unique à chaque chaîne de nom.

9.2.2 VISIBILITE ET CREATION DE NOMS

Les noms sont créés par les constructions mentionnées à la section 7.1. A l'exception des noms de champ et des noms d'éléments d'ensemble, les noms ont une occurrence de définition unique, qui est la construction qui introduit le nom. De manière à présenter un traitement uniforme pour la détermination de la visibilité et de l'identification de chaque nom, on considère que le mécanisme suivant pour donner une occurrence de définition unique à chaque nom créé s'applique:

- Dans le domaine d'un groupe, on considère que chaque occurrence de *mode* est une occurrence d'utilisation d'un nom de synmode virtuel défini à l'intérieur de ce domaine. Pour les définitions de procédures la définition de synmode virtuel du mode résultat est placée dans le domaine du groupe qui inclut la procédure. Les définitions de synmodes virtuels des modes de paramètres formels sont placées dans le domaine de la procédure.

Les règles de visibilité et d'identification s'appliquent en tenant compte de ces définitions virtuelles.

Example:

```
DCL I SET(A,B),
    K INT,
    J ARRAY (SET(A,B)) INT;
```

est considéré être remplacé par:

```
SYNMODE &1 = SET(A,B), &2 = INT,
        &3 = ARRAY(&1) &2;
 DCL I &1, K &2 J &3;
```

&1, *&2* et *&3* sont des noms de synmode virtuels. Les règles de visibilité s'appliquent à ces remplacements virtuels. Les remplacements virtuels ont pour conséquence que les modes créateurs de noms (*SET*, *STRUCT*) n'apparaissent qu'une fois dans un domaine, en partie droite de définitions de synmode virtuels. Cette définition de synmode est considérée comme l'occurrence de définition unique des noms d'éléments d'ensemble ou des noms de champ.

Les propriétés de visibilité et d'identification des noms de champ sont différentes (plus simples) que celles des autres noms. De ce fait, dans la suite de la section 9.2, le mot "nom" n'inclut pas les noms de champ, sauf mention contraire.

9.2.3 *NOMS IMPLIQUES*

Dans un domaine, chaque nom fortement visible a un ensemble (éventuellement vide) de noms impliqués défini de la manière suivante:

- Chaque mode a un ensemble (éventuellement vide) de noms impliqués, énumérés dans la table 2.

Les noms impliqués d'un nom (fortement visible) sont:

- Si le nom est un nom d'accès, les noms impliqués sont les noms impliqués par le mode du nom d'accès.
- Si le nom est un nom de mode, les noms impliqués sont les noms impliqués du mode définissant.
- Si le nom est un nom de procédure, les noms impliqués sont les noms impliqués du mode de la spec de résultat.
- Si le nom est un nom de signal, les noms impliqués sont les noms impliqués par ses modes attachés.
- Sinon il n'y a pas de noms impliqués.

Table 2. Noms impliqués par les modes

Modes	Ensembles de noms impliqués
INT, BIN, CHAR *INSTANCE, PTR* *BOOL, EVENT* *CHAR(n), BIN(n)* *BIT(n), RANGE(....)*	Vide
nom de mode	L'ensemble des noms impliqués par son mode définissant
M(m:n)	L'ensemble des noms impliqués de *M*
REF M, ROW M, *READ M, POWERSET M* *PROC(---) (M)* *BUFFER M*	L'ensemble des noms impliqués de *M*
ARRAY (M) N	L'union des ensembles des noms impliqués de *M* et *N*
STRUCT(N_1 M_1,..,N_n M_n)	L'union des ensembles des noms impliqués de M_1 jusqu'à M_n Pour les structures variables c'est l'union des noms impliqués de tous les champs de la structure variable.
VM(---) paramétré	L'ensemble des noms impliqués de *VM*
SET(...)	L'ensemble des noms d'éléments d'ensemble

(A noter que les noms impliqués, qui sont toujours des noms d'éléments d'ensemble, n'ont jamais eux-mêmes de noms impliqués)

9.2.4 *VISIBILITE DANS LES DOMAINES*

Un nom qui est fortement visible dans un domaine, y est soit directement fortement visible soit indirectement fortement visible.

Un nom n'est directement fortement visible dans un domaine que dans les cas suivants:

- Le nom a son occurrence de définition dans le domaine.

- Le nom est saisi dans le domaine (voir section 9.2.6.3).

- Le nom est octroyé dans le domaine (voir section 9.2.6.2).

Un nom n'est indirectement fortement visible dans un domaine que dans les cas suivants:

- Le domaine est un domaine bloc et le nom est hérité (voir section 9.2.5).

- Le nom est fortement visible dans le domaine englobant immédiatement et a la propriété d'envahissement dans ce domaine englobant (voir section 9.2.6.2) et le domaine ne contient pas de nom directement fortement visible avec la même chaîne de nom. Dans ce cas, le nom a aussi la propriété d'envahissement dans le domaine.

Un nom n'est faiblement visible dans un domaine que dans le cas suivant:

- Le nom est impliqué par un nom fortement visible dans ce domaine.

Les règles de visibilité sont définies de telle manière que dans chaque domaine, tous les noms fortement visibles ont des chaînes de nom différentes. Cependant, deux noms ou plus, faiblement visibles, peuvent avoir la même chaîne de nom. Un tel nom ne peut alors pas être employé dans certains cas (voir section 9.2.8).

9.2.5 *VISIBILITE ET BLOCS*

La règle de visibilité suivante s'applique aux blocs:

- Un nom, fortement visible dans le domaine d'un groupe, est indirectement fortement visible dans le domaine de chaque bloc englobé immédiatement qui n'a pas de nom directement fortement visible avec la même chaîne de nom (héritage de noms par les blocs).

9.2.6 *VISIBILITE ET MODULIONS*

9.2.6.1 Generalités

syntaxe:

<énoncé de visibilité> ::= (1)
<énoncé d'octroi> (1.1)
| *<énoncé de saisie>* (1.2)

sémantique: Les énoncés de visibilité, qui ne sont autorisés que dans le domaine des modulions, contrôlent la visibilité des noms qui y sont explicitement mentionnés (et implicitement de leurs noms impliqués).

9.2.6.2 Enoncés d'octroi

syntaxe:

```
<énoncé d'octroi> ::=                                          (1)
     GRANT <fenêtre d'octroi> [PERVASIVE];                     (1.1)

<fenêtre d'octroi> ::=                                         (2)
     <élément octroyé> {, <élément octroyé>}*                  (2.1)
   | ALL                                                       (2.2)

<élément octroyé> ::=                                          (3)
     <nom non réservé>                                         (3.1)
   | <nom de neumode> <clause interdire>                       (3.2)

<clause interdire> ::=                                         (4)
     FORBID {<liste de noms interdits> | ALL}                  (4.1)

<liste de noms interdits> ::=                                  (5)
     (<nom de champ> {, <nom de champ>}*)                      (5.1)
```

sémantique: Les énoncés d'octroi permettent d'étendre la visibilité de noms dans un domaine de modulion vers le domaine englobant immédiatement. *FORBID* ne peut être spécifié que pour des noms de neumodes qui sont des noms de mode structure. Cela signifie que tous les locus et les valeurs de ce mode ont des champs qui ne peuvent être sélectionnés qu'à l'intérieur du module octroyant, pas à l'extérieur.

Les règles de visibilité suivantes s'appliquent:

- Un nom, visible dans le domaine d'un modulion, est directement fortement visible dans le domaine du groupe englobant immédiatement s'il est mentionné dans un énoncé d'octroi dans le domaine du modulion. Le nom est dit être octroyé dans le domaine englobant.

- La notation *FORBID ALL* est une abréviation syntaxique interdisant tous les noms de champ du *nom de neumode* (voir section 9.2.7).

- La notation *GRANT ALL [PERVASIVE]* est une abréviation syntaxique pour l'octroi de tous les noms (avec la propriété d'envahissement, si spécifié), qui sont *fortement visibles* dans le domaine du modulion octroyant et dont l'occurrence de définition est à l'intérieur du modulion octroyant.

propriétés statiques: Un nom octroyé avec l'attribut *PERVASIVE* a la propriété d'envahissement dans le domaine englobant.

conditions statiques: L'occurrence de définition de chaque *nom non-réservé* doit être à l'intérieur du modulion octroyant.

Le *nom de neumode* qui est accompagné de la spécification *FORBID* doit avoir son occurrence de définition dans le domaine du modulion octroyant et chaque *nom de champ* dans la *liste de noms interdits* doit être un nom de champ du *nom de neumode*.

Si un énoncé d'octroi est placé dans le domaine d'une région il ne peut octroyer ni un nom qui est un nom de valeur régionale ni un nom d'accès régional.

exemples:

```
1.11    GRANT add,mult;                                    (1.1)
```

9.2.6.3 Enoncés de saisie

syntaxe:

```
<énoncé de saisie> ::=                                     (1)
     SEIZE <fenêtre de saisie>;                            (1.1)

<fenêtre de saisie> ::=                                    (2)
     <élément saisi> {, <élément saisi>}*                  (2.1)
   | ALL                                                   (2.2)

<élément saisi> ::=                                        (3)
     <nom de modulion> ALL                                 (3.1)
   | <nom non réservé>                                     (3.2)

<nom de modulion> ::=                                      (4)
     <nom de module>                                       (4.1)
   | <nom de région>                                       (4.2)
```

sémantique: Les énoncés de saisie sont une manière d'étendre la visibilité de noms dans le domaine de groupes vers le domaine de modulions englobés immédiatement.

Les règles de visibilité suivantes s'appliquent:

- Un nom, visible dans le domaine d'un groupe, est directement fortement visible dans le domaine d'un modulion directement englobé s'il est mentionné dans un énoncé de saisie dans le domaine du modulion. Le nom est dit être saisi dans le domaine du modulion.

- Si un nom, qui a la propriété d'envahissement dans le domaine englobant est saisi, il sera directement fortement visible dans le domaine du modulion qui saisit et il gardera sa propriété d'envahissement.

- La notation *SEIZE ALL* est une abréviation syntaxique pour saisir tous les noms qui sont fortement visibles dans le domaine du groupe englobant et dont l'occurrence de définition est à l'extérieur du modulion qui saisit.

- La notation *SEIZE <nom de modulion> ALL* est une abréviation syntaxique pour saisir tous les noms qui sont fortement visibles dans le domaine du groupe englobant et sont octroyés par le module ou la région dénoté par le *nom de modulion*.

conditions statiques: L'occurrence de définition de chaque *nom non réservé* ou *nom de modulion* doit être à l'extérieur du modulion qui saisit.

Un nom mentionné dans une *liste de saisie* ne peut être ni un *nom de valeur faire-avec* ni un *nom de locus faire-avec*.

exemples:

```
15.14   SEIZE /*external signals */
        ACQUIRE, RELEASE, CONGESTED, STEP, READOUT;          (1.1)
```

9.2.7 *VISIBILITE DE NOMS DE CHAMP*

Des noms de champ ne peuvent apparaître en dehors de leur occurrence de définition que dans les contextes suivants:

- La sélection d'un champ d'un locus structure ou d'une valeur structure.

- Les multiplets de structure avec noms de champs.

- Les clauses interdire dans les énoncés d'octroi.

Dans les deux premiers contextes seuls les noms de champ qui sont attachés au mode du locus structure, valeur structure (forte) ou multiplet sont visibles, sauf si la nouveauté de ce mode est un nom de neumode qui a été octroyé par un modulion avec une clause interdire. Dans le dernier cas, à l'extérieur du modulion octroyant, seuls les noms de champ qui ne sont pas mentionnés dans la *clause interdire* sont visibles.

Dans le dernier contexte tous les, et seulement les, noms de champ du nom de neumode octroyé sont visibles.

9.2.8 *IDENTIFICATION*

L'identification est le mécanisme par lequel on associe un nom unique à toute occurrence d'une chaîne de nom.

Les règles d'identification dépendent de ce que la chaîne de nom apparaît dans le contexte:

1. d'un *nom de directive*,

2. d'un *nom d'exception*,

3. d'un *nom réservé*,

4. d'un *nom de champ*,

5. d'un *nom non réservé*, *nom de module* ou *nom de région* dans un *élément saisi*,

6. tout autre *nom*.

Règles d'identification

1. Une chaîne de nom de directive suit un schéma d'identification défini par l'implémentation et qui ne peut pas influencer les règles d'identification de CHILL (voir section 2.6).

2. Une chaîne de nom d'exception est traitée suivant les régles d'identification de filet données en section 10.3.

3. Une chaîne de nom réservé qui n'est pas libéré par une directive de libération dans l'unité de compilation dans laquelle elle apparaît a sa signification réservée. Si elle est libérée, elle suit les règles de 6. Même dans ce cas elle ne peut être octroyée à l'extérieur de l'unité de compilation.

4. Une chaîne de nom de champ est identifiée comme suit, dépendant du contexte mentionné en section 9.2.7:

   - avec le nom de champ visible du mode structure du locus structure ou expression structure (forte);

   - avec le nom de champ visible du mode structure du multiplet (fort);

   - avec le nom de champ visible du nom de neumode

   Si la chaîne de nom ne peut être identifiée avec un tel nom de champ, le programme est en erreur.

5. Une chaîne de nom apparaissant dans le contexte d'un *élément saisi* est identifiée suivant les régles mentionnées en 6, mais dans le domaine qui englobe immédiatement le domaine dans lequel l'énoncé de saisie

est placé.

6. Pour toute autre occurrence d'une chaîne de nom dans le domaine d'un groupe:

   a. s'il y a plus d'un nom fortement visible dans le domaine avec cette chaîne de nom, le programme est en erreur;

   b. sinon, s'il y a un nom fortement visible dans le domaine avec cette chaîne de nom, la chaîne de nom est identifiée avec ce nom;

   c. sinon, s'il y a exactement un nom faiblement visible dans le domaine avec cette chaîne de nom, le nom est identifié avec ce nom;

   d. sinon, s'il y a plus d'un nom faiblement visible dans le domaine avec cette chaîne de nom, et si tous ces noms (d'éléments d'ensemble) ont des classes compatibles, alors la chaîne de nom est identifiée avec un de ces nom, arbitrairement;

   e. sinon, le programme est en erreur.

En plus des règles mentionnées ci-dessus, une chaîne de nom qui apparaît dans un énoncé d'octroi ou un énoncé de saisie doit être identifiée avec un nom dont l'occurrence de définition est à l'intérieur ou à l'extérieur du modulion octroyant, respectivement saisissant (s'il y a un choix, à cause de la règle d.).

10.0 FILETS D'EXCEPTION

10.1 GENERALITES

Une exception est soit une exception définie par le langage, auquel cas elle peut avoir un nom défini par le langage, une exception définie par l'utilisateur, ou 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. 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, il est connu statiquement quelles exceptions pourraient arriver (c.-à-d. on sait statiquement quelles exceptions ne peuvent pas arriver) et pour quelles exceptions un filet qui convient peut être trouvé, ou quelles exceptions 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.

10.2 FILETS

syntaxe:

<filet> ::= *(1)*
*ON {<choix d'exceptions>}**
[ELSE <liste d'énoncés d'action>] END *(1.1)*

<choix d'exceptions> ::= *(2)*
(<liste d'exeptions>) : <liste d'énoncés d'action> *(2.1)*

sémantique: Une liste d'énoncés d'action dans un choix d'exceptions sera entamée si une exception est causée dans l'énoncé terminé par le filet et dont le nom est mentionné dans la *liste d'exceptions* dans le choix d'exceptions. Si *ELSE* est spécifié, la liste d'énoncés d'action qui le suit sera exécutée si une exception est causée dans l'énoncé terminé par le filet et dont le nom n'apparaît dans aucune *liste d'exceptions* directement contenue dans le filet.

Si un filet termine une action, quand la fin de la liste d'énoncés d'action d'un choix d'exceptions est atteinte, le contrôle passe à l'action qui suit l'énoncé d'action terminé par le filet.

Si un filet termine une définition de procédure, le contrôle reviendra au point d'appel quand la fin d'une liste d'énoncés d'action est atteinte. Si le filet termine une définition de processus, le processus s'exécutant se terminera quand la fin d'une liste d'énoncés d'action dans le choix d'exceptions est atteinte.

conditions statiques: Tous les noms 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 requêtes en mémoire ne peuvent être satifaites.

exemples:

```
10.43   ON
          (no_space): CAUSE overflow;
        END                                              (1.1)
```

10.3 IDENTIFICATION DE FILET

Quand une exception E arrive dans une action A, ou un énoncé descriptif ou une région D, l'exception sera traitée par le filet qui convient, c.-à-d. une liste d'énoncés d'action dans le filet sera exécutée, ou l'exception sera passée au point d'appel de la procédure, ou, si rien n'est possible, le programme est en erreur.

Pour toute action A, ou énoncé descriptif 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 termine A ou D et mentionne E dans une liste d'exceptions ou spécifie *ELSE*, alors ce filet est le filet qui convient par rapport à E;

2. sinon, si A ou D sont englobés immédiatement par une action parenthésée, le filet qui convient (si présent) est le filet qui convient pour l'action parenthésée par rapport à E;

3. sinon, si A ou D sont placés dans le domaine d'une définition de procédure alors:

   - si un filet est spécifié après la définition de procédure, et spécifie E dans une liste d'exceptions ou spécifie *ELSE*, alors ce filet est le filet qui convient,

   - 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 sont placés dans le domaine d'une définition de processus (éventuellement l'imaginaire), alors

   - si un filet est spécifié après la définition de processus, et spécifie E dans une liste d'exceptions ou spécifie *ELSE* alors ce filet est le filet qui convient,

   - sinon, il n'y a pas de filet;

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 descriptif D' par rapport à E que le filet termine 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.

11.0 OPTIONS POUR L'IMPLEMENTATION

11.1 OPERATIONS PREDEFINIES

syntaxe:

```
<appel d'opération prédéfinie> ::=                                  (1)
      <nom d'opération prédéfinie>
      ([<liste de paramètres d'opération prédéfinie>])              (1.1)

<liste de paramètres d'opération prédéfinie> ::=                    (2)
      <paramètre d'opération prédéfinie>
      {, <paramètre d'opération prédéfinie>}*                       (2.1)

<paramètre d'opération prédéfinie> ::=                              (3)
      <valeur>                                                      (3.1)
    | <locus>                                                       (3.2)
    | <nom non réservé>                                             (3.3)
```

sémantique: Une implémentation peut fournir un ensemble d'opérations prédéfinies par l'implémentation en plus de l'ensemble des opérations prédéfinies par le langage.

Une valeur, un locus, ou tout nom défini dans le programme qui n'est pas un nom réservé peut être passé comme paramètre. L'appel à l'opération prédéfinie peut rendre une valeur ou un locus. La mécanisme de passage de paramètre est défini par l'implémentation.

Une opération prédéfinie peut être générique, c.-à-d. 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) peuvent 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 nom *d'opération prédéfinie* est un nom défini par l'implémentation qui est considéré défini dans le module prélude standard (voir section 7.8). Il peut avoir un ensemble de noms d'exceptions défini 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).

11.2 MODES ENTIER DEFINIS PAR L'IMPLEMENTATION

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.

11.3 NOMS DE REGISTRE DEFINIS PAR L'IMPLEMENTATION

Une implémentation peut définir un ensemble de noms de registre prédéfinis (voir sections 3.7 et 7.8).

11.4 NOMS D'EXCEPTION ET DE PROCESSUS DEFINIS 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 être placée dans le domaine du module prélude standard. Les processus ainsi définis peuvent être démarrés et des valeurs exemplaire les dénotant peuvent être manipulées.

Une implémentation peut définir un ensemble de noms d'exception pour tout nom de processus ou groupe de noms de processus. Ces exceptions peuvent arriver quand les processus sont démarrés (voir section 6.14).

11.5 FILETS DEFINIS PAR L'IMPLEMENTATION

Une implémentation peut spécifier que un filet défini par l'implémentation termine la définition de processus imaginaire la plus externe (voir section 7.8). Les noms d'exceptions et les actions dans un filet défini par l'implémentation peuvent être n'importe quel nom d'exception légal en CHILL ou n'importe quelle action. A noter qu'un choix d'exceptions d'un tel filet ne peut s'entamer que par l'arrivée d'une exception dans le processus le plus externe et non dans un des processus internes.

11.6 OPTIONS DE SYNTAXE

A certains endroits, CHILL offre plusieurs syntaxes pour la description d'une même sémantique. Le choix d'une des options suivantes devrait être unique dans le programme tout entier.

Symbole d'affectation

Le symbole d'affectation est soit *:=* soit =

ARRAY

Le nom réservé *ARRAY* devrait être soit obligatoire soit interdit.

RETURNS

Dans des définitions de procédure avec une spec de résultat, le nom réservé *RETURNS* devrait être soit obligatoire soit interdit.

Modes structure

Les modes structure doivent se noter soit dans la notation emboîtée soit dans la notation étagée.

Parenthèses de littéral et multiplet

Si des crochets carrés sont disponibles dans l'alphabet de représentation, les chrochets *[* et *]* peuvent s'employer au lieu de respectivement *(:* et *:)*.

APPENDICE A: ENSEMBLES DE CARACTERES POUR PROGRAMMES CHILL

A.1 ALPHABET CCITT NO. 5 VERSION INTERNATIONALE DE REFERENCE

Avis V.3 (la représentation interne est le nombre binaire formé par les bits b7 à b1, où b1 est le bit le moins significatif).

b_4 b_3 b_2 b_1		b_7 0	0	0	0	1	1	1	1
		b_6 0	0	1	1	0	0	1	1
		b_5 0	1	0	1	0	1	0	1
		0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	TC_7 (DLE)	SP	0	@	P		p
0 0 0 1	1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0 0 1 0	2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0 0 1 1	3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0 1 0 0	4	TC_4 (EOT)	DC_4	¤	4	D	T	d	t
0 1 0 1	5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0 1 1 0	6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0 1 1 1	7	BEL	TC_{10} (ETB)	'	7	G	W	g	w
1 0 0 0	8	FE_0 (BS)	CAN	(	8	H	X	h	x
1 0 0 1	9	FE_1 (HT)	EM	)	9	I	Y	i	y
1 0 1 0	10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1 0 1 1	11	FE_3 (VT)	ESC	+	;	K	[	k	{
1 1 0 0	12	FE_4 (FF)	IS_4 (FS)	,	<	L	\	l	|
1 1 0 1	13	FE_5 (CR)	IS_3 (GS)	-	=	M	]	m	}
1 1 1 0	14	SO	IS_2 (RS)	.	>	N	^	n	~
1 1 1 1	15	SI	IS_1 (US)	/	?	O	_	o	DEL

A.2 ALPHABET MINIMAL POUR REPRESENTER LES PROGRAMMES CHILL

Le sous-ensemble suivant de l'alphabet CCITT no. 5, code de base, est employé dans ce document pour représenter les programmes CHILL.

b_4	b_3	b_2	b_1		0	1	2	3	4	5	6	7
					0	1	2	3	4	5	6	7
0	0	0	0	0			SP	0		P		
0	0	0	1	1				1	A	Q		
0	0	1	0	2				2	B	R		
0	0	1	1	3				3	C	S		
0	1	0	0	4				4	D	T		
0	1	0	1	5				5	E	U		
0	1	1	0	6				6	F	V		
0	1	1	1	7			'	7	G	W		
1	0	0	0	8			(	8	H	X		
1	0	0	1	9			)	9	I	Y		
1	0	1	0	10			*	:	J	Z		
1	0	1	1	11			+	;	K			
1	1	0	0	12			,	<	L			
1	1	0	1	13			-	=	M			
1	1	1	0	14			.	>	N			
1	1	1	1	15			/	?	O	_		

APPENDICE B: SYMBOLES SPECIAUX

	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 relationel
<=	inférieur ou égal à	opérateur relationel
=	égal à	opérateur relational, affectation, initialisation
/=	différent de	opérateur relationnel
>=	supérieur ou égal à	opérateur relationnel
>	supérieur à	opérateur relationnel
+	plus	opérateur d'addition
-	moins	opérateur de soustraction
*	astérisque	opérateur de multiplication, valeur indéfinie, valeur anonyme
/	solidus	opérateur de division
//	double solidus	opérateur de concaténation
->	fleche	repérage ou dérepérage
<>	diamant	début ou fin d'une clause de directive
/*	crochet d'ouverture de commentaire	début de commentaire
*/	crochet fermant de commentaire	fin de commentaire
'	apostrophe	symbole de début ou de fin de divers littéraux
''	apostrophe double	apostrophe dans un littéral de caractère ou chaîne de caractères
_	soulignement	espacement dans les noms et les littéraux

APPENDICE C: NOMS SPECIAUX DE CHILL

C.1 NOMS RESERVES

ALL
ARRAY
ASSERT

BASED
BEGIN
BUFFER
BY

CALL
CASE
CAUSE
CONTINUE

DCL
DELAY
DO
DOWN

ELSE
ELSIF
END
ENTRY
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MODULE

NEWMODE
NOPACK

OD
OF
ON
OUT

PACK
PERVASIVE
POS
POWERSET
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

THEN
TO

UP

WHILE
WITH

C.2 NOMS PREDEFINIS

ABS	FALSE	NOT	SIZE
ADDR		NULL	SUCC
AND		NUM	
	GETSTACK		
			THIS
BIN		OR	TRUE
BIT	INSTANCE		
BOOL	INT		
		PRED	UPPER
		PTR	
CARD	MAX		
CHAR	MIN		XOR
	MOD	REM	

C.3 NOMS D'EXCEPTIONS DE CHILL

ASSERTFAIL
DELAYFAIL
EMPTY
EXTINCT
MODEFAIL
OVERFLOW
RANGEFAIL
RECURSEFAIL
SPACEFAIL
TAGFAIL

C.4 DIRECTIVES DE CHILL

FREE

1. Opérations sur des entiers

```
integer_operations:
MODULE
    add:
    PROC (i,j INT)(INT) EXCEPTIONS (OVERFLOW);
       RESULT i+j;
    END add;
    mult:
    PROC (i,j INT)(INT) EXCEPTIONS (OVERFLOW);
       RESULT i*j;
    END mult;
    GRANT add, mult;
    GRANT add, mult;
    SYNMODE operand_mode=INT;
    GRANT operand_mode;
    SYN neutral_for_add=0,
       neutral_for_mult=1;
    GRANT neutral_for_add,
          neutral_for_mult;
END integer_operations;
```

2. Mêmes opérations sur des fractions

```
fraction_operations:
MODULE
    NEWMODE fraction=STRUCT (num,denum INT);
    add:
    PROC (f1,f2 fraction)(fraction) EXCEPTIONS (OVERFLOW);
       RETURN [f1.num*f2.denum+f2.num*f1.denum,
           f1.denum*f2.denum];
    END add;
    mult:
    PROC (f1,f2 fraction)(fraction) EXCEPTIONS (OVERFLOW);
       RETURN [f1.num*f2.num,f2.denum*f1.denum];
    END mult;

    GRANT add, mult;
    SYNMODE operand_mode=fraction;
    GRANT operand_mode;
    SYN neutral_for_add fraction=[0,1],
       neutral_for_mult fraction=[1,1];
    GRANT neutral_for_add,
         neutral_for_mult;

END fraction_operations;
```

3. Mêmes opérations sur des nombres complexes

```
1   complex_operations
2   MODULE
3      NEWMODE complex=STRUCT (re,im INT);
4      add:
5      PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
6         RETURN [c1.re+c2.re,c1.im+c2.im];
7      END add;
8      mult:
9      PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
10        RETURN [c1.re*c2.re-c1.im*c2.im ,
11                c1.re*c2.im+c1.im*c2.re];
12     END mult;
13
14     GRANT add, mult
15     SYNMODE operand_mode=complex;
16     GRANT operand_mode;
17     SYN neutral_for_add=complex [0,0],
18        neutral_for_mult=complex [1,0];
19     GRANT neutral_for_add,
20           neutral_for_mult;
21
22  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 INOUT, b,c,order INT) EXCEPTIONS (wrong_input) RECURSIVE;
5         DCL d INT;
6         ASSERT b>0 AND c>0 AND order>0
7            ON (ASSERTFAIL):
8               CAUSE wrong_input;
9            END;
10        CASE order OF
11           (1): a :=b+c;
12              RETURN;
13           (2): d :=0;
14           (ELSE): d :=1;
15        ESAC;
16        DO FOR i :=1 TO c;
17           op (a,b,d,order-1);
18           d :=a;
19        OD;
20        RETURN;
21     END op;
22
23     GRANT op;
24
25  END general_order_arithmetic;
```

5. Additionner bit à bit et vérifier le résultat

```
add_bit_by_bit:
MODULE
      adder:
      PROC (a STRUCT(a2,a1 BOOL) IN, b STRUCT(b2,b1 BOOL) IN)
          RETURNS(STRUCT(c4,c2,c1 BOOL));

      DCL c STRUCT (c4,c2,c1 BOOL);
      DCL k2,x,w,t,s,r BOOL;
      DO WITH a,b,c;
          k2 :=a1 AND b1;
          c1 :=NOT k2 AND (a1 OR b1);
          x :=e2 AND b2 AND k2;
          w :=a2 OR b2 OR k2;
          t :=b2 AND k2;
          s :=a2 AND k2;
          r :=a2 AND b2;
          c4 :=r OR s OR t;
          c2 :=x OR (w AND NOT c4);
      OD;
      RETURN c;
   END adder;
   GRANT adder;
END add_bit_by_bit;

exhaustive_checker:
MODULE
   SEIZE adder;
   DCL a STRUCT (a2,a1 BOOL),
      b STRUCT (b2,b1 BOOL),
   SYNMODE res=ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
   DCL r INT, results res;
   DO WITH a,b;
      r :=0;
      DO FOR a2 IN BOOL;
         DO FOR a1 IN BOOL;
            DO FOR b2 IN BOOL;
               DO FOR b1 IN BOOL;
                  r+ :=1;
                  results (r) :=adder (a,b);
               OD;
            OD;
         OD;
      OD;
      ASSERT result=res [[FALSE,FALSE,FALSE],[FALSE,FALSE,TRUE],
                         [FALSE,FALSE,TRUE],[FALSE,TRUE,TRUE],
                         [FALSE,FALSE,TRUE],[FALSE,TRUE,FALSE],
                         [FALSE,TRUE,TRUE],[TRUE,FALSE,FALSE],
                         [FALSE,TRUE,FALSE],[FALSE,TRUE,FALSE],
                         [TRUE,FALSE,FALSE],[TRUE,FALSE,TRUE.],
                         [FALSE,TRUE,TRUE],[TRUE,FALSE,FALSE],
                         [TRUE,FALSE,TRUE],[TRUE,TRUE,FALSE]];
END exhaustive_checker;
```

6. Jouer avec des dates

```
playing_with_dates:
MODULE /* from collected algorithms from CACM no. 199 */
   SYNMODE month=SET(jan,feb,mar,apr,may,jun,
                     jul,aug,sep,oct,nov,dec);
   NEWMODE date=STRUCT (day INT (1:31), mo month, year INT);

   gregorian-date:
   PROC (julian_day_number INT)(date);
      DCL j INT :=julian_day_number,
          d,m,y INT;
      j-:=1_721_119;
      y :=(4 * j - 1) / 146_097;
      j :=4 * j - 1 - 146_097 * y;
      d :=j / 4;
      j :=(4 * d + 3) / 1_461;
      d :=4 * d + 3 - 1_461 * j;
      d :=(d + 4) / 4;
      m :=(5 * d - 3) / 153;
      d :=5 * d - 3 - 153 * m;
      d :=(d + 5) / 5;
      y :=100 * y + j;
      IF m<100 THEN m+:=3;
               ELSE m-:=9;
                    y+:=1;
      FI;
      RETURN [d,month (m+1), y];
   END gregorian_date;

   julian_day_number
   PROC (d date)(INT);
      DCL c,y,m INT;
      DO WITH d;
         m :=NUM (mo)+1;
         IF m>2 THEN m-:=3;
                ELSE m+:=9;
                     year-:=1;
         FI;
         c :=year/100;
         y :=year-100*c;
         RETURN (146_097*c)/4+(1_461*y)/4
            +(153+m+c)/5+day+1_721_119;
      OD;
   END julian_day_number;
   GRANT gregorian_date, julian_day_number;
END playing_with_dates

test:
MODULE
  SEIZE gregorian_date, julian_day_number;
  ASSERT julian_day_number ([10,dec,1979])=julian_day_number(
       gregorian_date(julian_day_number([10,dec,1979])));
END test;
```

7. Nombres romains

```
1   Roman:
2   MODULE
3      SEIZE n,rn;
4      convert:
5      PROC () EXCEPTIONS (string_too_small);
6         DCL r INT :=0;
7         DO WHILE n>=1_000;
8            rn(r):='M';
9            r+:=1;
10           n-:=1_000;
11        OD;
12        IF n>500 THEN rn(r):='D';
13                      n-:=500;
14                      r+:=1;
15        FI;
16        IF n>=100 THEN rn(r):='C';
17                       n-:=100;
18                       r+:=1;
19        FI;
20        IF n>=50 THEN rn(r):='L';
21                      n-:=50;
22                      r+:=1;
23        FI;
24        IF n>=10 THEN rn(r):='X';
25                      n-:=10;
26                      r+:=1;
27        FI;
28        DO WHILE n>=1;
29           rn(r):='I';
30           r+:=1;
31           n-:=1;
32        OD;
33        RETURN;
34     END ON (RANGEFAIL): DO FOR i :=0 TO UPPER (rn);
35                            rn(i) :='.';
36                         OD;
37                         CAUSE string_too_small;
39     END convert;
40  END Roman;
41  test:
42  MODULE
43     SEIZE convert;
44     DCL n INT INIT :=1979;
45     DCL rn CHAR (20) INIT :=(20)' ';
46     GRANT n,rn;
47
48     convert ();
49
50     ASSERT rn='MDCCCCLXXVIIII'//(6)' ';
51
52  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 CHAR (max) IN, output ARRAY('A':'Z') INT OUT);
7           DO FOR i :=0 TO UPPER (input ->);
8               IF input -> (i) IN letter
9                   THEN
10                      output (input -> (i))+:=1;
11              FI;
12          OD;
13      END count;
14      GRANT count;
15  END letter-count;
16  test:
17  MODULE
18      DCL c CHAR (10) INIT :='A-B<ZAA9K''';
19      DCL output ARRAY ('A' : 'Z') INT;
20      SYN max=10_000;
21      GRANT max;
22      SEIZE count;
23      count (-> c,output);
24      ASSERT output=[('A') : 3,('B','K','Z') : 1, (ELSE) : 0];
25
26  END test;
```

9. Nombres premiers

```
1   prime:
2   MODULE
3       SEIZE max;
4       NEWMODE number_list =POWERSET INT(2:max);
5       SYN  empty = number_list [];
6       DCL sieve number_list INIT := [2:max];
7           primes number_list INIT :=empty;
8       GRANT primes;
9       DO WHILE sieve/=empty;
10          primes OR :=[MIN (sieve)];
11          DO FOR j :=MIN (sieve) BY MIN (sieve) TO max;
12              sieve-:=[j];
13          OD;
14      OD;
15  END prime;
```

10. Implémenter des piles de deux manières différentes

```
1   stacks_1:
2   MODULE
3       SEIZE element
4       SYN max=10_000,min=1;
5       DCL stack ARRAY (min : max) element,
6          stackindex INT INIT :=min;
7       push:
8       PROC (e element) EXCEPTIONS (overflow);
9          IF stackindex=max
10            THEN CAUSE overflow;
11         FI;
12         stackindex+:=1;
13         stack (stackindex) :=e;
14         RETURN;
15      END push;
16      pop:
17      PROC () EXCEPTIONS (underflow);
18         IF stackindex=min
19            THEN CAUSE underflow;
20         FI;
21         stackindex-:=1;
22         RETURN;
23      END pop;
24
25      elem:
26      PROC (i INT)(element LOC) EXCEPTIONS (bounds);
27         IF i<min OR i>max
28            THEN CAUSE bounds;
29         FI;
30         RETURN stack (i);
31      END elem;
32
33      GRANT push,pop,elem;
34  END stacks-1;
```

```
35  stacks_2:
36  MODULE
37      SEIZE element;
38      NEWMODE cell=STRUCT (pred,succ REF cell,
39                           info element);
40      DCL p,last,first REF cell INIT :=NULL;
41      push:
42      PROC (e element) EXCEPTIONS (overflow);
43          p :=allocate (cell) ON
44                               (nospace) : CAUSE overflow;
45                               END;
46          IF last=NULL
47              THEN first,last :=p;
48              ELSE last ->. succ :=p;
49                  p ->. pred :=last;
50                  last :=p;
51          FI;
52          last ->. info :=e;
53          RETURN;
54      END push;
55      pop:
56      PROC () EXCEPTIONS (underflow);
57          IF last=NULL;
58              THEN CAUSE underflow;
59          FI;
60          last :=last ->. pred; IF last = NULL THEN first := NULL FI;
61          last ->. succ :=NULL;
62          RETURN;
63      END pop;
64      elem:
65      PROC (i INT) (element LOC) EXCEPTIONS (bounds);
66          IF first=NULL
67              THEN CAUSE bounds;
68          FI;
69          p :=first;
70          DO FOR j=2 TO i;
71              IF p ->. succ=NULL
72                  THEN CAUSE bounds;
73              FI;
74              p :=p ->. succ;
75          OD;
76          RETURN p ->. info;
77      END elem;
78
79      GRANT push,pop,elem;
80
81  END stacks_2;
```

11. Fragments pour jouer aux échecs

```
1   NEWMODE piece=STRUCT(color SET(white,black),
2                        kind SET(pawn,rook,knight,bishop,queen,king));
3   NEWMODE column=SET (a,b,c,d,e,f,g,h);
4   NEWMODE line=INT (1 : 8);
5   NEWMODE square=STRUCT (status SET (occupied,free),
6                          CASE status OF
7                          (occupied) : p piece,
8                          (free) :
9                          ESAC);
10  NEWMODE board=ARRAY (line) ARRAY (column) square;
11  NEWMODE move=STRUCT (lin_1,lin_2 line,
12                       col_1,col_2 column);
13
14  initialise:
15  PROC (bd board INOUT);
16     bd :=[(1) : [(a,h):[.status: occupied, .p : [white,rook]],
17           (b,g):[.status: occupied, .p : [white,knight]],
18           (c,f):[.status: occupied, .p : [white,bishop]],
19             (d):[.status: occupied, .p : [white,queen]],
20             (e):[.status: occupied, .p : [white,king]]],
21    (2):[(ELSE) : [.status: occupied, .p : [white,pawn]]],
22   (3:6):[(ELSE) : [.status: free]],
23     (7):[(ELSE) : [.status: occupied, .p : [black,pawn]]],
24     (8):[(a,h) : [.status: occupied, .p : [black,rook]],
25       (b,g):[.status: occupied, .p : [black,knight]],
26              (e,f) : [.status: occupied, .p : [black,bishop]],
27                (d) : [.status: occupied, .p : [black,king]],
28                (e) : [.status: occupied, .p : [black,queen]]]];
29     RETURN;
30
31  END initialise;
```

```
32  register_move:
33  PROC (b board LOC,m move) EXCEPTIONS (illegal);
34     DCL starting square LOC :=b (m.lin_1)(m.col_2),
35         arriving square LOC :=b (m.lin_1)(m.col_2);
36
37     DO WITH m;
38        IF starting.status=free
39           OR (lin_2<1 OR lin_2>8 OR col_2<a OR col_2>h)
40           OR (arriving.status/=free AND arriving.p.kind=king)
41           THEN
42              CAUSE illegal;
43        FI;
44        CASE starting.p.kind, starting.p.color OF
45
46         (pawn),(white):
47            IF col_1 = col_2 AND (arriving.status/=free
48               OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
49               OR (col_2=PRED(col_1) OR col_2=SUCC(col_1))
50               AND arriving.status=free OR arriving.p.color=white
51            THEN
52                CAUSE illegal; /*capturing en passant not implemented*/
53            FI;
54         (pawn),(black):
55            IF col_1=col_2 AND (arriving.status/=free
56               OR NOT (lin_2=lin_1-1 OR lin_2=lin_1-2 AND lin_1=7))
57               OR (col_2=PRED(col_1) OR col_2=SUCC(col_1))
58               AND arriving.status=free OR arriving.p.color=black
59               THEN
60                  CAUSE illegal; /* same remark */
61            FI;
62         (rook),(*):
63            IF NOT ok_rook (b,m)
64               THEN
65                  CAUSE illegal;
66            FI;
67         (bishop),(*):
68            IF NOT ok_bishop (b,m)
69               THEN
70                   CAUSE illegal;
71            FI;
72         (queen),(*):
73            IF NOT ok_rook (b,m)
74               THEN
75                  IF NOT ok_bishop (b,m)
76                     THEN
77                        CAUSE illegal
78                  FI;
79           FI;
80         (knight,*):
81           IF ABS(ABS(NUM(col_2)-NUM(col_1))
82              -ABS(lin_2-lin_1)) /= 1
83              OR ABS(NUM(col_2)-NUM((col_1))
84                 +ABS(lin_2-lin_1) =/ 3
85              OR arriving.status/=free AND
```

```
86                arriving.p.color=starting.p.color
87             THEN CAUSE illegal;
88             FI;
89               (king),(*):
90                IF ABS(NUM(col_2)-NUM(col_1)) > 1
91                   OR ABS(lin_2-lin_1) > 1
92                   OR lin_2=lin_1 AND col_2=col_1
93                   OR arriving.status/=free AND
94                      arriving.p.color=starting.p.color
95                THEN CAUSE illegal;
96                FI; /*checking king moving to check not implemented*/
97           ESAC;
98       OD;
99       arriving :=starting;
100      RETURN;
101   END register_move;
102   ok_rook:
103   PROC (b board,m move)(BOOL);
104      DO WITH m;
105         IF NOT (col_2=col_1 OR lin_1=lin_2)
106            OR arriving.status/=free AND
107            arriving.p.color=starting.p.color
108         THEN RETURN FALSE;
109         FI;
110         IF col_1=col_2
111            THEN IF lin_1<lin_2
112                    THEN DO FOR l := lin_1+1 TO lin_2-1;
113                            IF board (l)(col_1).status/=free
114                               THEN RETURN FALSE;
115                            FI;
116                         OD;
117                    ELSE DO FOR l := lin_1-1 DOWN TO lin_2+1;
118                            IF board (l)(col_1).status/=free
119                               THEN RETURN FALSE;
120                            FI;
121                         OD;
122                 FI;
123            ELSE IF col_1<col_2
124                    THEN DO FOR c := SUCC(col_1) TO PRED(col_2);
125                            IF board (lin_1)(c).status/=free
126                               THEN RETURN FALSE;
127                            FI;
128                         OD;
129                    ELSE DO FOR c := SUCC(col_2) DOWN TO PRED(col_1);
130                            IF board (lin_1)(c).status/=free
131                               THEN RETURN FALSE;
132                            FI;
133                         OD;
134                 FI;
135         FI;
136         RETURN TRUE;
137      OD;
138   END ok_rook;
```

```
ok_bishop:
PROC (b board,m move)(BOOL);
    DO WITH m;
        CASE lin_2>lin_1,col_2>col_1 OF
            (TRUE),(TRUE): c := col_1
                        DO FOR  l := lin_1+1 TO lin_2-1;
                            c := SUCC(c);
                            IF board (l)(c).status/=free
                                THEN RETURN FALSE;
                            FI;
                        OD;
                        IF SUCC(c)/=col_2
                            THEN RETURN FALSE;
                        FI;
            (TRUE),(FALSE): c := col_1
                         DO FOR l := lin_1+1 TO lin_2-1;
                             c := PRED(c);
                             IF board (l)(c).status/=free
                                 THE RETURN FALSE;
                             FI;
                         OD;
                         IF PRED(c)/=col_2
                             THEN RETURN FALSE;
                         FI;
            (FALSE),(TRUE): c := col_1
                         DO FOR l := lin_1-1 DOWN TO lin_2+1;
                             c := SUCC(c)
                             IF board (l)(c).status/=free
                                 THEN RETURN FALSE;
                             FI;
                         OD;
                         IF SUCC(c)/=col_2
                             THEN RETURN FALSE;
                         FI:
            (FALSE),(FALSE): c := col_1;
                          DO FOR l := lin_1-1 DOWN TO lin_2+1;
                              c := PRED(c);
                              IF board (l)(c).status/=free
                                  THEN RETURN FALSE;
                              FI;
                          OD;
                          IF PRED (c)/=col_2
                              THEN RETURN FALSE;
                          FI;
         ESAC;
         RETURN arriving.status=free OR
                arriving.p.color/=starting.p.color;
    OD;
END ok_bishop;
```

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           INSERT:
            PROC(NEW NODE);
10              /* INSERT ACTIONS */
11          END INSERT;

12          REMOVE:
            PROC();
13              /* REMOVE ACTIONS */
14          END REMOVE;

15          INITIALIZE_LIST:
16          BEGIN
17              DCL LAST REF NODE:= ->HEAD;
18              DO FOR NEW IN POOL;
19                  NEW.PRED := LAST;
20                  LAST->.SUC:= ->NEW;
21                  LAST:= ->NEW;
22                  NEW.VALUE:=0;
23              OD;
24              HEAD.PRED:=LAST;
25              LAST->.SUC:= ->HEAD;
26          END INITIALIZE_LIST

27      END HANDLE_LIST;

28      DCL NODE_A NODE:=(: NULL,NULL,536 :);
29      REMOVE();
30      REMOVE();
31      INSERT(NODE_A);
32  END CIRCULAR_LIST;
```

13. Une région pour donner des accès compétitifs à une resource

```
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       ALLOCATE:
8       PROC()(INT);
9           DO FOR EVER;
10              DO FOR I IN RESOURCE_SET;
11                  IF NOT ALLOCATED(I)
12                  THEN
13                      ALLOCATED(I) := TRUE;
14                      RETURN I;
15                  FI;
16              OD;
17              DELAY RESOURCE_FREED;
18          OD;
19      END ALLOCATE;

20      DEALLOCATE:
21      PROC(I INT);
22          ALLOCATE(I) := FALSE;
23          CONTINUE RESOURCE_FREED;
24      END DEALLOCATE;

25  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 calla 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      CALL_DISTRIBUTOR:
12      PROCESS();
13          DO FOR EVER;
14              WAIT(10 /*seconds*/);
15              CONTINUE OPERATOR_IS_READY;
16          OD;
17      END CALL_DISTRIBUTOR;

18      CALL:
19      PROCESS();
20          DELAY CASE
21          (OPERATOR_IS_READY): /* some actions */
22          (SWITCH_IS_CLOSED): DO FOR I IN INT(1:100);
23                                  CONTINUE OPERATOR_IS_READY;
24                                  /*empty the queue*/
25                              OD;
26          ESAC;
27      END CALL;

28      OPERATOR:
29      PROCESS();
30          DO FOR_EVER;
31              IF TIME = 1700
32                  THEN
33                      CONTINUE SWITCH_IS_CLOSED;
34              FI;
35          OD;
36      END OPERATOR;

37      START CALL_DISTRIBUTOR():
38      START OPERATOR();
39      DO FOR I INT(1:100);
40          START CALL();
41      OD;
42  END SWITCHBOARD;
```

15. Allouer et désallouer un ensemble de resources

```
1   <> FREE (STEP);
2   COUNTER MANAGER:
3   MODULE
4   /* To illustrate the use of signals and the receive case, (buffers
5     might have been instead) we will look at an example where an
6     ALLOCATOR manages a set of resources, in this case a set of
7     COUNTERs. The module is part of a larger system where there are
8     USERs, that can request the services of the COUNTER_MANAGER. The
9     module is made to consistof two process definitions, one for the
10    ALLOCATION and one for the COUNTERS. INITIATE and TERMINATE
11    are internal signals sent from the ALLOCATOR
12    to the COUNTERs. All the other signals are external, being sent
13    from or to the USERs. */

14     SEIZE /* external signals */
15        ACQUIRE, RELEASE, CONGESTED,STEP,READOUT;
16     SIGNAL INITIATE = (INSTANCE),
17           TERMINATE;

18     ALLOCATOR:
19     PROCESS();
20        NEWMODE NO_OF_COUNTERS = INT(1:100);
21     DCL COUNTERS ARRAY (NO_OF_COUNTERS)
22              STRUCT (COUNTER INSTANCE,
23                      STATUS SET (BUSY,IDLE));
24     DO FOR EACH IN COUNTERS;
25     EACH:= (: START COUNTER(), IDLE :);
26     OD;

27     DO FOR EVER;
28     BEGIN
29        DCL USER INSTANCE;
30        AWAIT_SIGNALS:
31        RECEIVE CASE SET USER;
32        (ACQUIRE):
33           DO FOR EACH IN COUNTERS;
34              DO WITH EACH;
35                 IF STATUS = IDLE
36                 THEN
37                    STATUS:=BUSY;
38                    SEND INITIATE (USER) TO COUNTER;
39                    EXIT AWAIT_SIGNALS;
40                 FI;
41              OD;
42           OD;
43           SEND CONGESTED TO USER;
44        (RELEASE IN THIS_COUNTER);
45           SEND TERMINATE TO THIS_COUNTER;
```

```
46              FIND_COUNTER:
47              DO FOR EACH IN COUNTERS;
48                 DO WITH EACH;
49                    IF THIS_COUNTER = COUNTER
50                    THEN
51                       STATUS:= IDLE;
52                       EXIT FIND_COUNTER;
53                    FI;
54                 OD;
55              OD FIND_COUNTER;
56           ESAC AWAIT_SIGNALS;
57        END;
58        OD;
59        END ALLOCATOR;

60        COUNTER:
61        PROCESS();
62           DO FOR EVER;
63           BEGIN
64              DCL USER INSTANCE;
65                 COUNT:= 0;
66              RECEIVE CASE
67                 (INITIATE IN RECEIVED_USER):
68                    SEND READY TO RECEIVED_USER;
69                    USER:= RECEIVED_USER;
70              ESAC;
71              WORK_LOOP:
72              DO FOR EVER;
73                 RECEIVE CASE
74                    (STEP): COUNT +:=1;
75                    (TERMINATE):
76                       SEND READOUT(COUNT) TO USER;
77                       EXIT WORK_LOOP;
78                 ESAC;
79              OD WORK_LOOP;
80           END;
81           OD;
82        END COUNTER;

83        START ALLOCATOR();
84     END COUNTER_MANAGER;
```

16. Allouer et désallouer un ensemble de resources en employant des tampons

```
<> FREE(STEP);
USER_WORLD:
MODULE
/* This example is the same as no.15 except that buffers are
   used for communication in stead of signals.
   The main difference is that processes are now identified
   by means of references to local message buffers rather than
   by instance values. There is one message buffer declared
   local to each process. There is one set of message types
   for each process definition. When started each process must
   identify its buffer address to the starting process.
   The USER_WORLD module sketches some of the environment in
   which the COUNTER_MANAGER is used. */

GRANT USER_BUFFERS,
      ALLOCATOR_MESSAGES, ALLOCATOR_BUFFERS,
      COUNTER_MESSAGES, COUNTER_BUFFERS;
NEWMODE
  USER_MESSAGES =
      STRUCT(TYPE SET(CONGESTED, READY,
                      READOUT, ALLOCATOR_ID),
             CASE TYPE OF
             (CONGESTED)   :
             (READY)       : COUNTER REF COUNTER_BUFFERS,
             (READOUT)     : COUNT INT,
             (ALLOCATOR_ID): ALLOCATOR REF ALLOCATOR_BUFFERS
             ESAC),
  USER_BUFFERS = BUFFER(1) USER_MESSAGES,
  ALLOCATOR_MESSAGES =
      STRUCT(TYPE SET(ACQUIRE, RELEASE, COUNTER_ID),
             CASE TYPE OF
             (ACQUIRE)   : USER REF USER_BUFFERS,
             (RELEASE,
              COUNTER_ID): COUNTER REF COUNTER_BUFFERS
             ESAC),
  ALLOCATOR_BUFFERS = BUFFER(1) ALLOCATOR_MESSAGES,
  COUNTER_MESSAGES =
      STRUCT(TYPE SET(INITIATE, STEP, TERMINATE),
             CASE TYPE OF
             (INITIATE) : USER REF USER_BUFFERS,
             (STEP,
              TERMINATE):
             ESAC,
  COUNTER_BUFFERS = BUFFER(1) COUNTER_MESSAGES;
DCL USER_BUFFER USER_BUFFERS,
    ALLOCATOR_BUF REF ALLOCATOR_BUFFERS,
    COUNTER_BUF REF COUNTER_BUFFERS;
START ALLOCATOR(->USER_BUFFER);
ALLOCATOR_BUF := (RECEIVE USER_BUFFER).ALLOCATOR;
END_USER_WORLD;
```

```
51  COUNTER_MANAGER:
52  MODULE
53  SEIZE USER_BUFFERS,
54        ALLOCATOR_MESSAGES, ALLOCATOR_BUFFERS,
55        COUNTER_MESSAGES, COUNTER_BUFFERS;
56
57  ALLOCATOR:
58  PROCESS(STARTER REF USER_BUFFERS);
59     DCL ALLOCATOR_BUFFER ALLOCATOR_BUFFERS;
60     NEWMODE NO_OF_COUNTERS = INT(1:10);
61     DCL COUNTERS ARRAY(NO_OF_COUNTERS)
62           STRUCT(COUNTER REF COUNTER_BUFFERS,
63                  STATUS SET(BUSY, IDLE)),
64        MESSAGE ALLOCATOR_MESSAGES;
65     SEND STARTER->([ALLOCATOR_ID, ->ALLOCATOR_BUFFER]);
66     DO FOR EACH IN COUNTERS;
67        START COUNTER(->ALLOCATOR_BUFFER);
68        EACH := [(RECEIVE ALLOCATOR_BUFFER).COUNTER, IDLE];
69     OD;
70     DO FOR EVER;
71        BEGIN
72        DCL USER REF USER_BUFFERS;
73        MESSAGE := RECEIVE ALLOCATOR_BUFFER;
74        HANDLE_MESSAGES:
75        CASE MESSAGE.TYPE OF
76        (ACQUIRE):
77           USER := MESSAGE.USER;
78           DO FOR EACH IN COUNTERS;
79              DO WITH EACH;
80                 IF STATUS= IDLE
81                 THEN STATUS := BUSY;
82                      SEND COUNTER->([INITIATE, USER]);
83                      EXIT HANDLE_MESSAGES;
84                 FI;
85              OD;
86           OD;
87           SEND USER->([CONGESTED]);
88        (RELEASE):
89           SEND MESSAGE.COUNTER([TERMINATE]);
90           FIND_COUNTER:
91           DO FOR EACH IN COUNTERS;
92              DO WITH EACH;
93                 IF MESSAGE.COUNTER = COUNTER
94                 THEN STATUS := IDLE;
95                      EXIT FIND_COUNTER;
96                 FI;
97              OD;
98           OD FIND_COUNTER;
99        ESAC HANDLE_MESSAGES;
100       END;
101    OD;
102 END ALLOCATOR;
```

```
103 COUNTER:
104 PROCESS(STARTER REF ALLOCATOR_BUFFERS);
105   DCL COUNTER_BUFFER ALLOCATOR_BUFFERS;
106   SEND STARTER->([COUNTER_ID, ->COUNTER_BUFFER]);
107   DO FOR EVER;
108     BEGIN
109       DCL USER REF USER_BUFFERS,
110           COUNT INT := 0,
111           MESSAGE COUNTER_MESSAGES;
112       MESSAGE := RECEIVE COUNTER_BUFFER;
113       CASE MESSAGE.TYPE OF
114       (INITIATE): USER := MESSAGE.USER;
115                   SEND USER->([READY, ->COUNTER_BUFFER]);
116       ELSE /* some error action */
117       ESAC;
118       WORK_LOOP:
119       DO FOR EVER;
120         MESSAGE := RECEIVE COUNTER_BUFFER;
121         CASE MESSAGE.TYPE OF
122         (STEP)     : COUNT +:= 1;
123         (TERMINATE):SEND USER->([READOUT, COUNT]);
124                     EXIT WORK_LOOP;
125         ELSE /* some error action */
126         ESAC;
127       OD WORK_LOOP;
128   END;
129   OD;
130 END COUNTER;
131 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
37    scanner;
38
39 END string_scanner;
```

18. Parcours de chaîne 2

```
1  string_scanner2: /* This example is the same as no.18 but it uses
2                      character string in stead of packed arrays.*/
3  MODULE
4    SYN
5      blanks = (10)' ', linelength = 132;
6    SYNMODE
7      stringptr = ROW CHAR(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 (CHAR(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
38     scanner;
39
40 END string_scanner;
```

19. Enlever un élément d'une liste doublement chaînée circulairement

```
1  QUEUE_REMOVAL:
2  MODULE
3     SEIZE INFO;
4     GRANT REMOVE;
5     REMOVE: PROC(P PTR) RETURNS(INFO) EXCEPTIONS(EMPTY);
6     /* This procedure removes the item referred to by
7        P from a queue and returns the information contents
8        of that queue element.*/
9          DCL 1 X BASED (P),
10              2 I INFO POS(0,8:31),
11              2 PREV PTR POS(1,0:15),
12              2 NEXT PTR POS(1,16:31);
13         DCL PREV, NEXT PTR;
14         PREV := X.PREV;
15         NEXT := X.NEXT;
16         X.PREV, X.NEXT := NULL;
17         RESULT X.INFO;
18         P := PREV;
19         X.NEXT := NEXT;
20         P := NEXT;
21         X.PREV := PREV;
22    END REMOVE;
23
24 END QUEUE_REMOVAL;
```

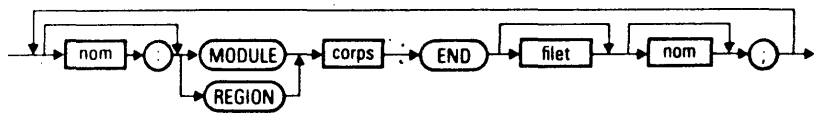
APPENDICE E: DIAGRAMMES SYNTAXIQUES

Les diagrammes de cet appendice décrivent la syntaxe de CHILL.

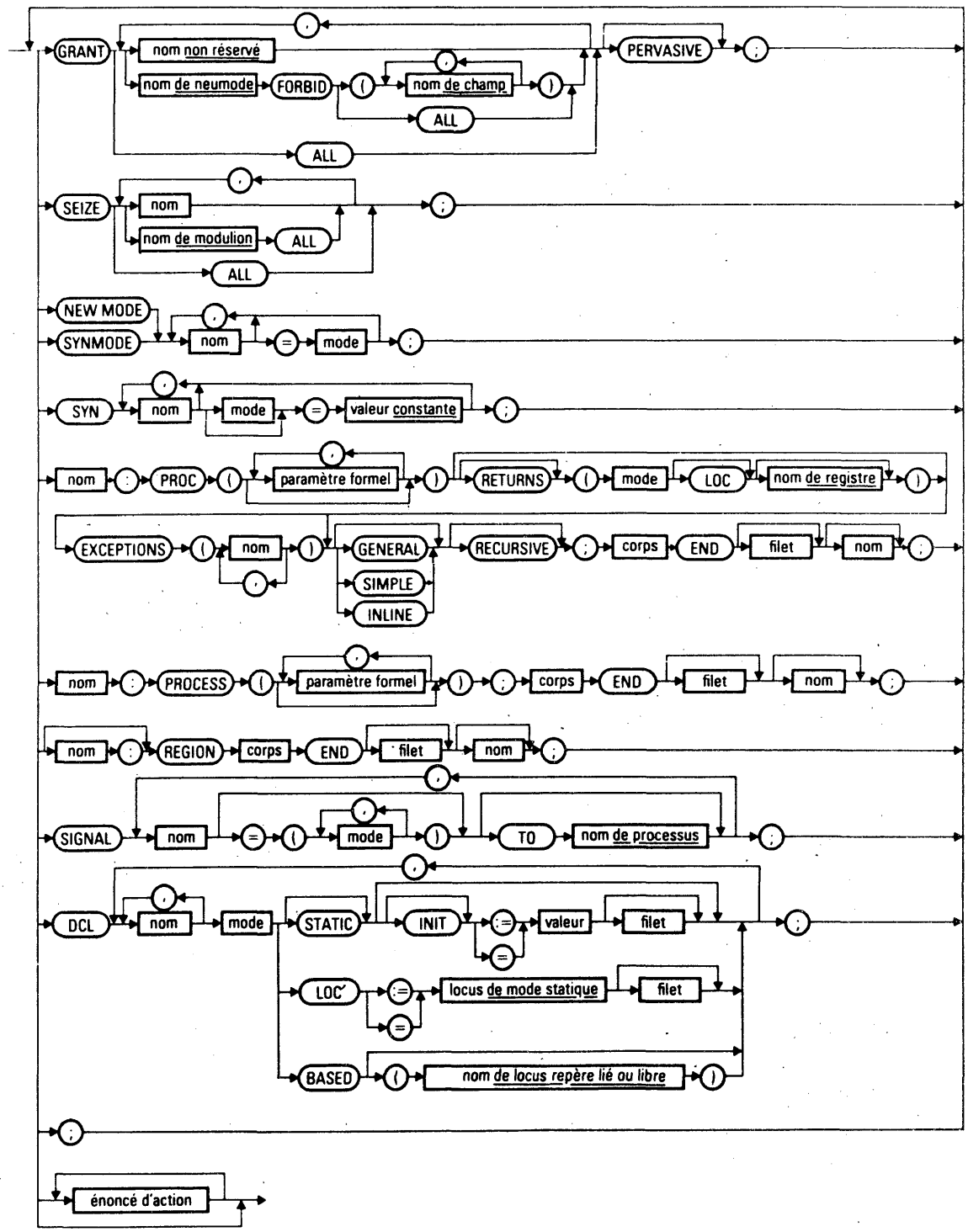
Les diagrammes ont été conçus pour être lus par des êtres humains pas pour servir de base à un algorithme d'analyse.

Des simplifications ont été faites de manière à améliorer la lisibilité et, de ce fait, ils ne peuvent être considérés comme définitifs, seulement comme une aide à la compréhension de CHILL. La définition de la syntaxe acontextuelle est spécifiée en forme de Backus-Naur à d'autress endroits dans ce document.

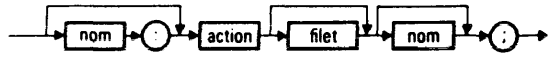
programme



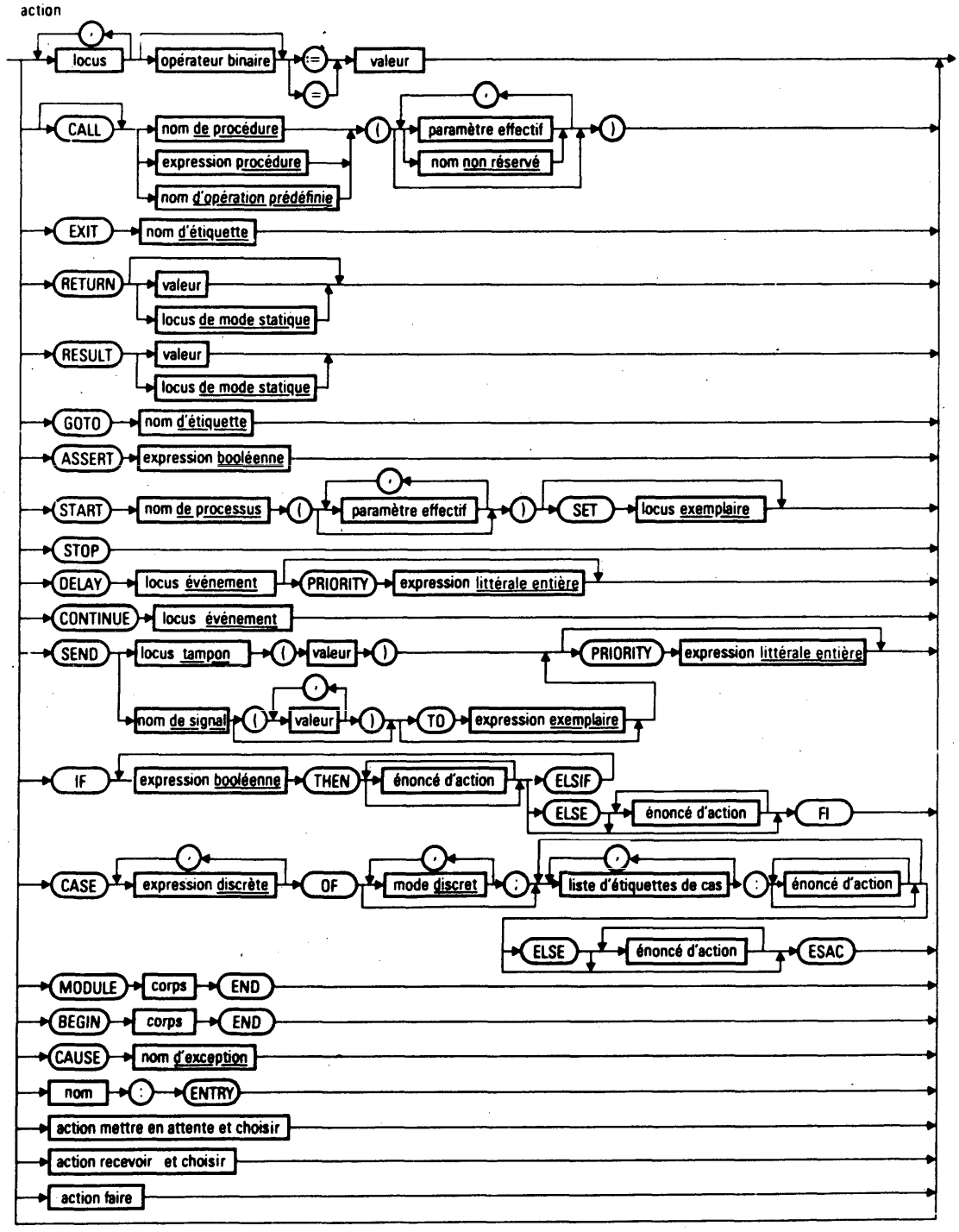
corps



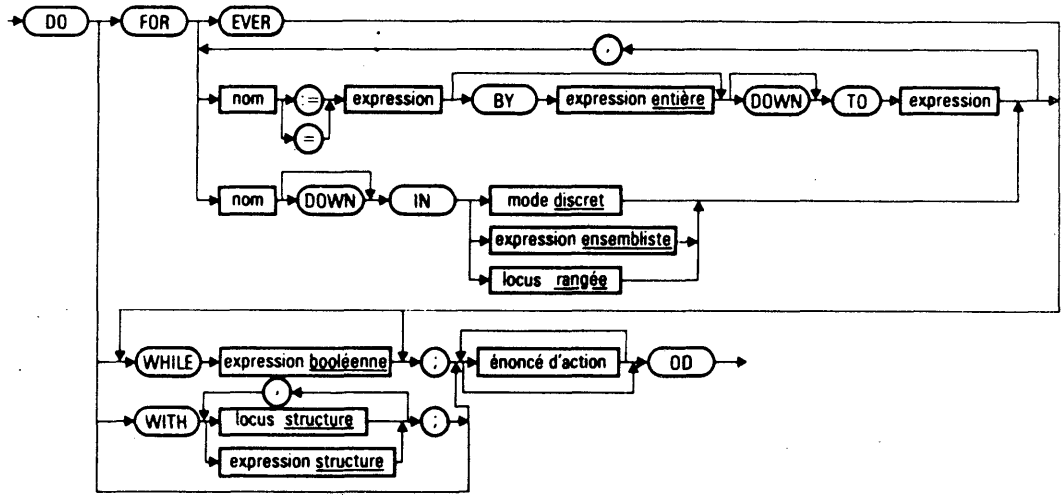
énoncé d'action



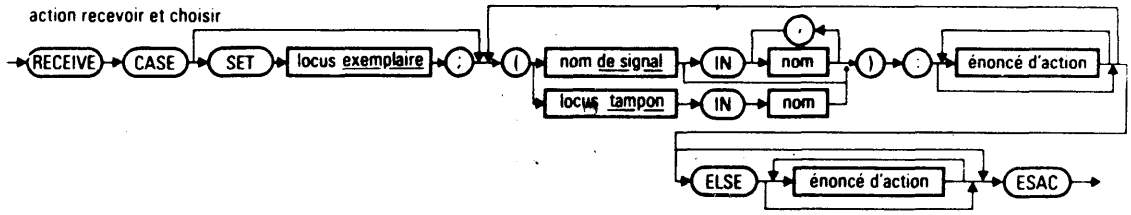
action



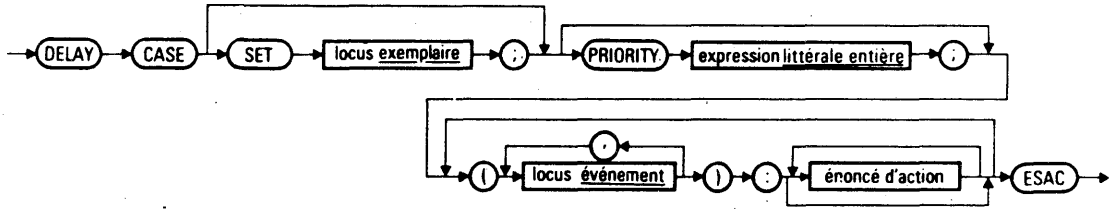
action faire



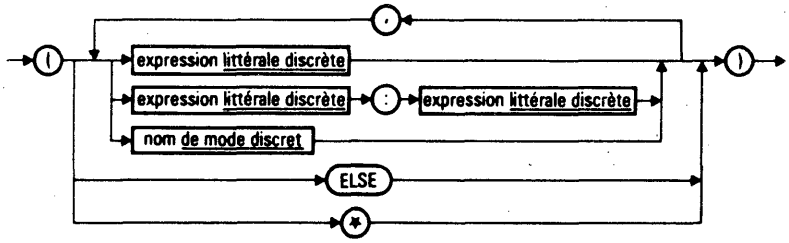
action recevoir et choisir



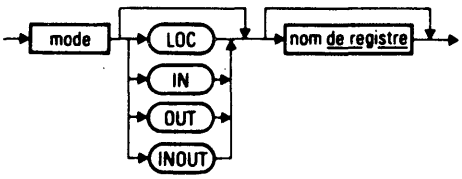
action mettre en attente et choisir



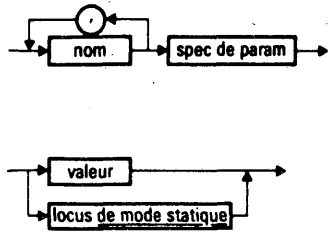
liste d'étiquettes de cas

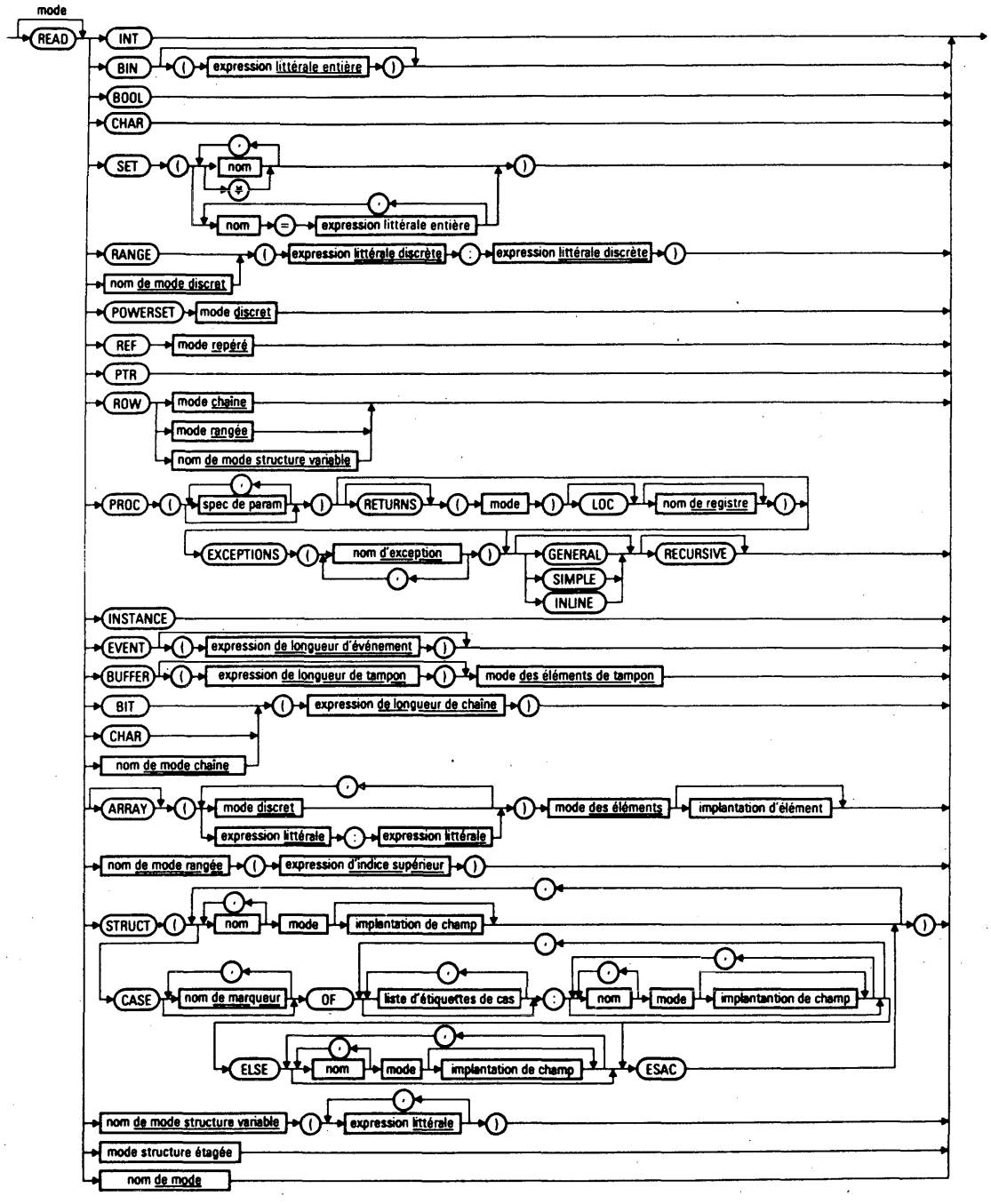


spec de param



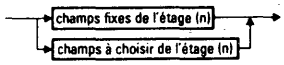
paramètre formel



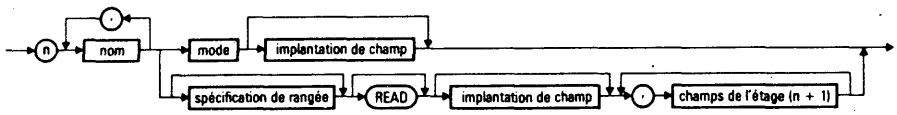

mode
READ
INT
BIN
(
expression littérale entière
)
BOOL
CHAR
SET
(
nom
¤
)
nom
=
expression littérale entière
RANGE
(
expression littérale discrète
:
expression littérale discrète
)
nom de mode discret
POWERSET
mode discret
REF
mode repéré
PTR
ROW
mode chaine
mode rangée
nom de mode structure variable
PROC
(
spec de param
)
RETURNS
(
mode
)
LOC
nom de registre
)
EXCEPTIONS
(
nom d'exception
)
GENERAL
SIMPLE
INLINE
RECURSIVE
INSTANCE
EVENT
(
expression de longueur d'événement
)
BUFFER
(
expression de longueur de tampon
)
mode des éléments de tampon
BIT
CHAR
(
expression de longueur de chaine
)
nom de mode chaine
ARRAY
(
mode discret
expression littérale
:
expression littérale
)
mode des éléments
implantation d'élément
nom de mode rangée
(
expression d'indice supérieur
)
STRUCT
(
nom
mode
implantation de champ
)
CASE
nom de marqueur
OF
liste d'étiquettes de cas
:
nom
mode
implantation de champ
ELSE
nom
mode
implantation de champ
ESAC
nom de mode structure variable
(
expression littérale
)
mode structure étagée
nom de mode

mode structure étagée

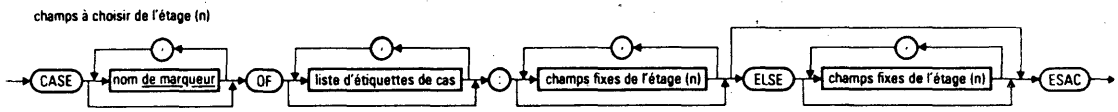
champs de l'étage (n)



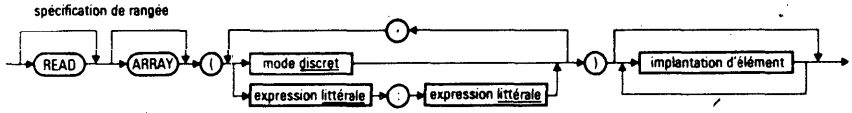
champs fixes de l'étage (n)



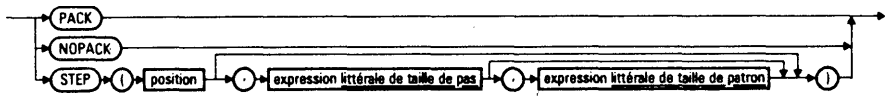
champs à choisir de l'étage (n)



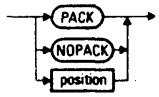
spécification de rangée



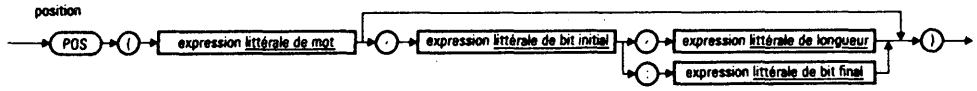
implantation d'élément

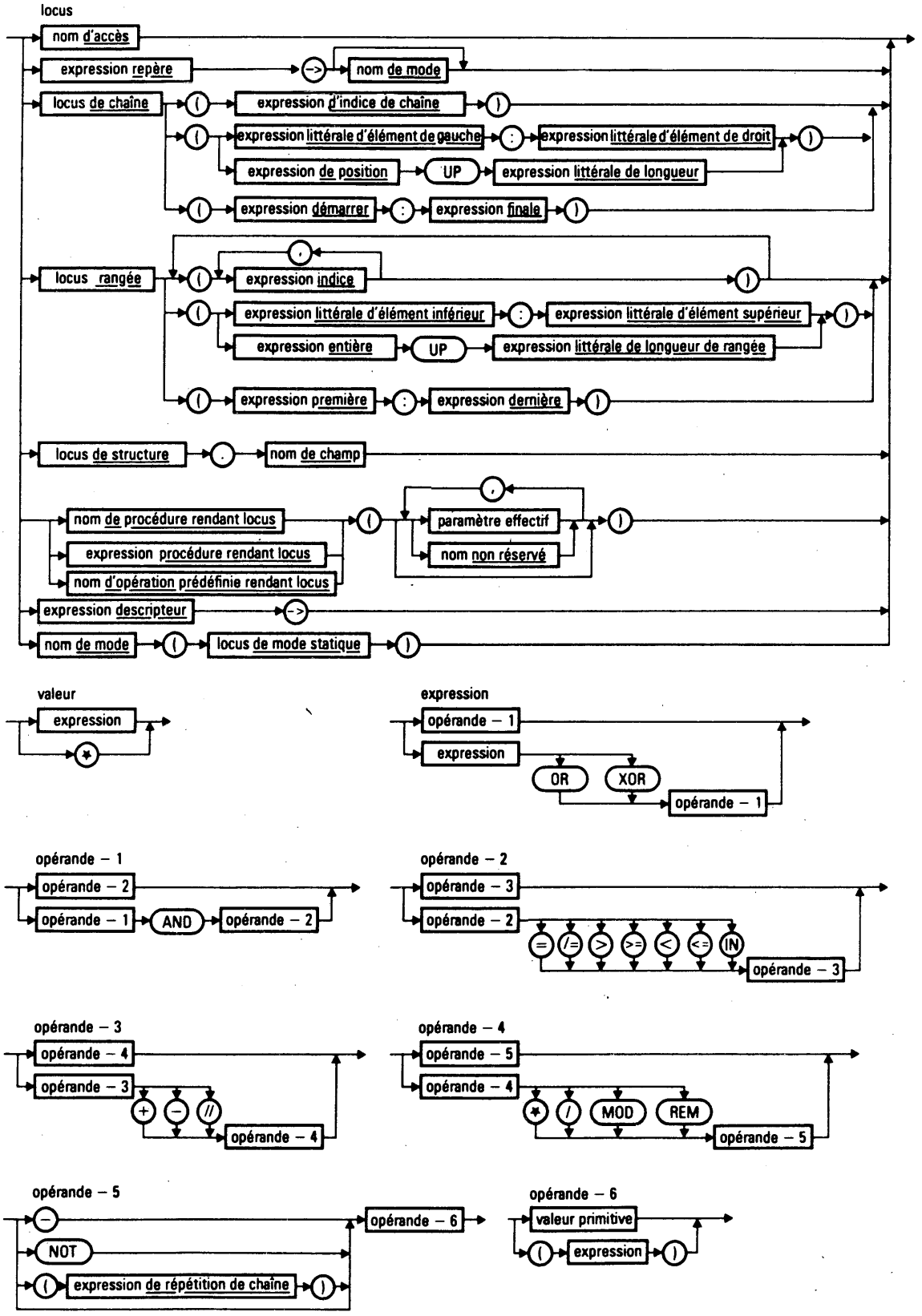


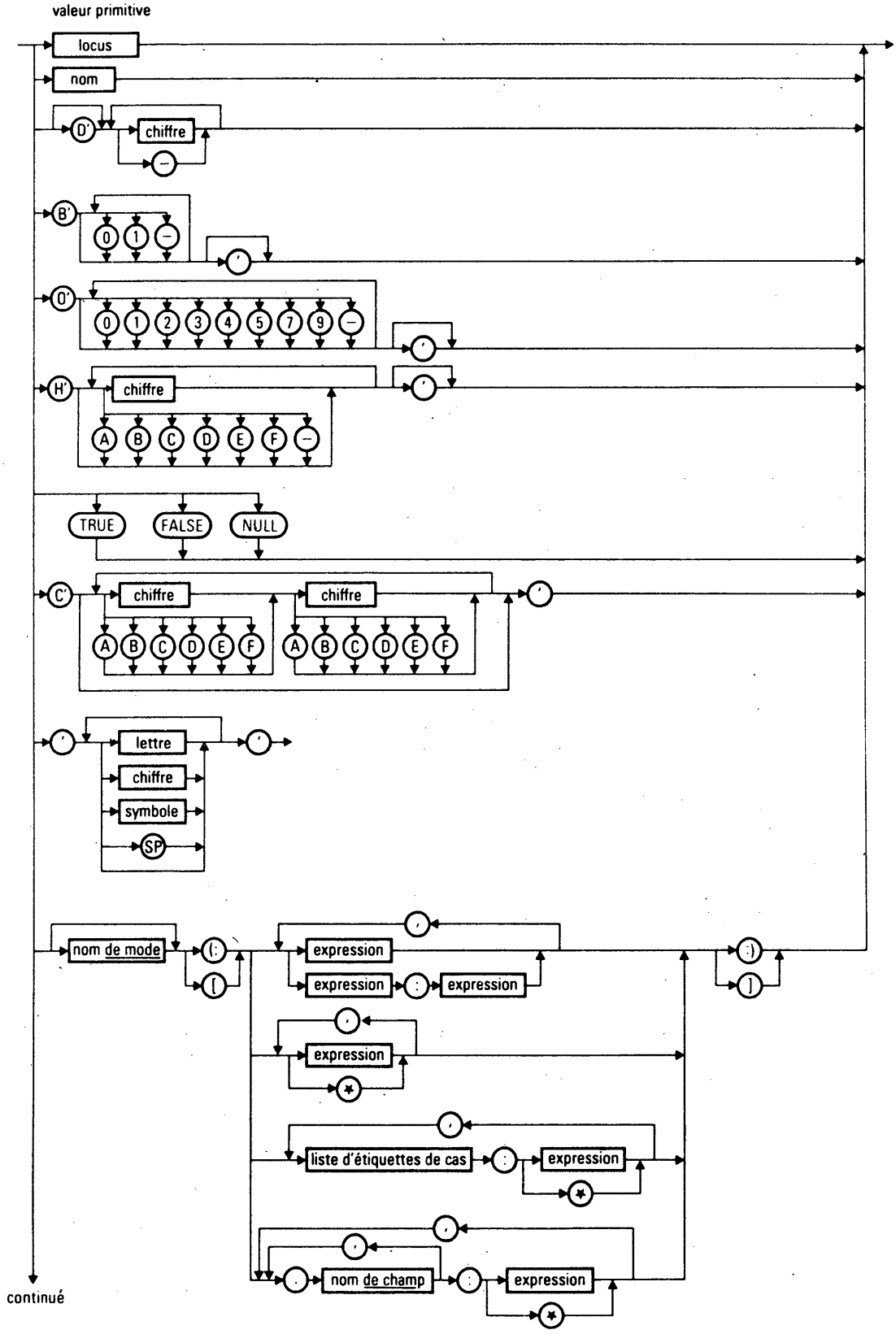
implantation de champ



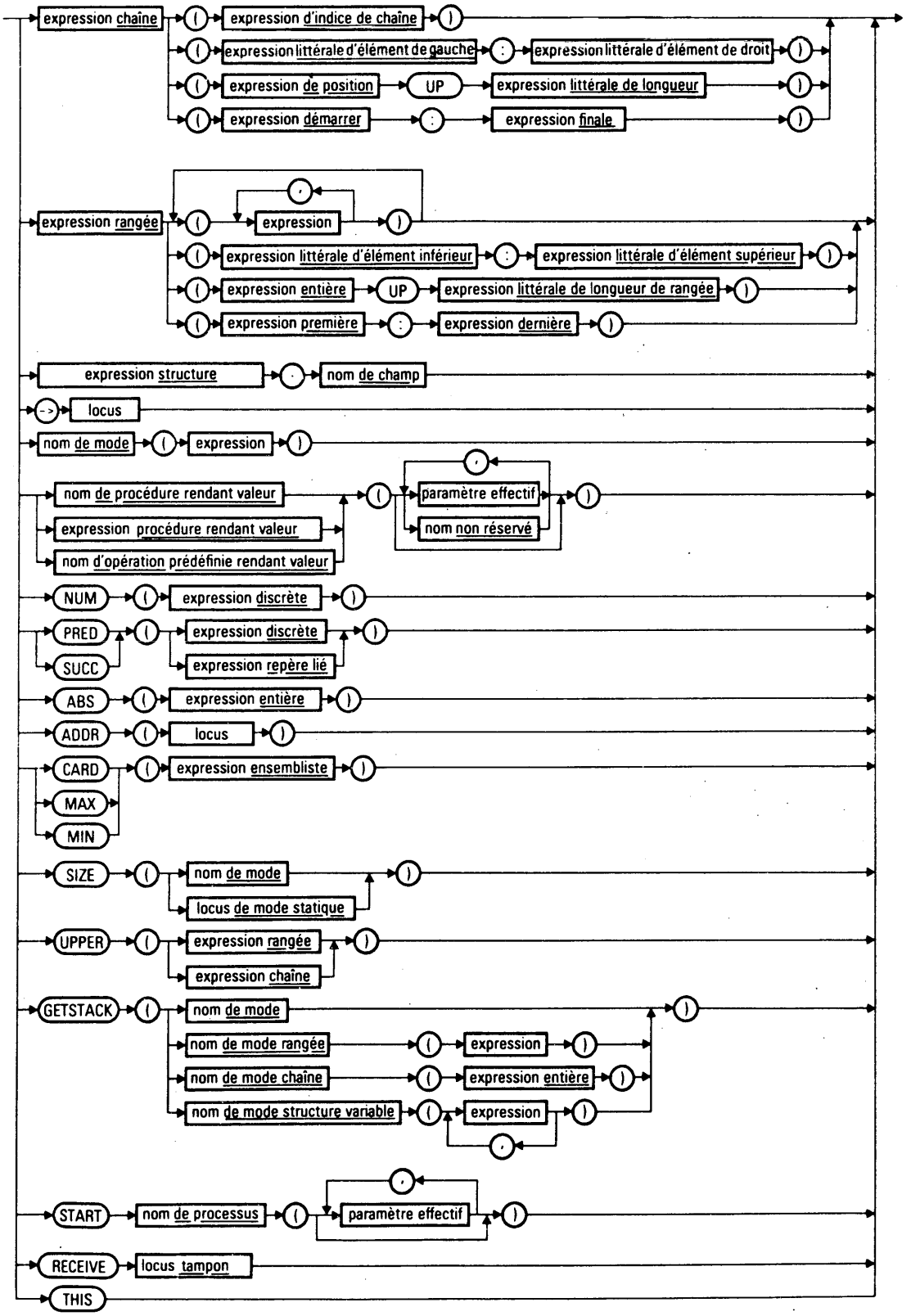
position



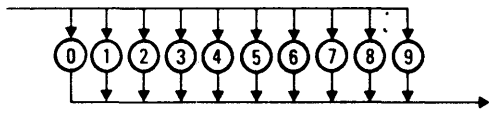

locus
nom d'accès
expression repère
->
nom de mode
locus de chaîne
(
expression d'indice de chaîne
)
expression littérale d'élément de gauche
:
expression littérale d'élément de droit
expression de position
UP
expression littérale de longueur
expression démarrer
expression finale
locus rangée
,
expression indice
expression littérale d'élément inférieur
expression littérale d'élément supérieur
expression entière
expression littérale de longueur de rangée
expression première
expression dernière
locus de structure
.
nom de champ
nom de procédure rendant locus
expression procédure rendant locus
nom d'opération prédéfinie rendant locus
paramètre effectif
nom non réservé
expression descripteur
nom de mode
locus de mode statique
valeur
expression
*
expression
opérande – 1
OR
XOR
opérande – 1
opérande – 1
opérande – 2
opérande – 1
AND
opérande – 2
opérande – 2
opérande – 3
opérande – 2
=
/=
>
>=
<
<=
IN
opérande – 3
opérande – 3
opérande – 4
opérande – 3
+
–
//
opérande – 4
opérande – 4
opérande – 5
opérande – 4
*
/
MOD
REM
opérande – 5
opérande – 5
–
NOT
(
expression de répétition de chaîne
)
opérande – 6
opérande – 6
valeur primitive
(
expression
)


valeur primitive
locus
nom
D'
chiffre
B'
0
1
O'
2
3
4
5
7
9
H'
A
B
C
D
E
F
TRUE
FALSE
NULL
C'
lettre
symbole
SP
nom de mode
expression
liste d'étiquettes de cas
nom de champ
continué

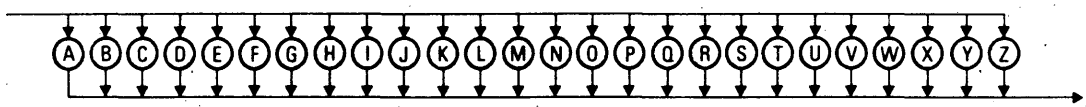
valeur primitive (continué)



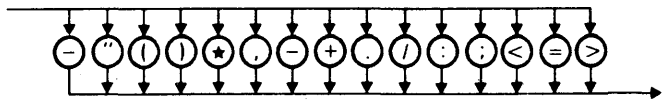
chiffre



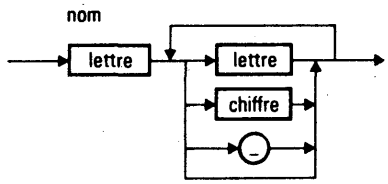
lettre



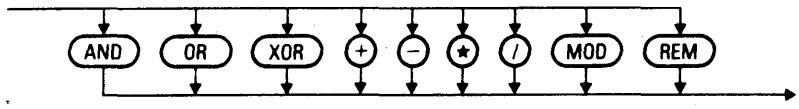
symbole



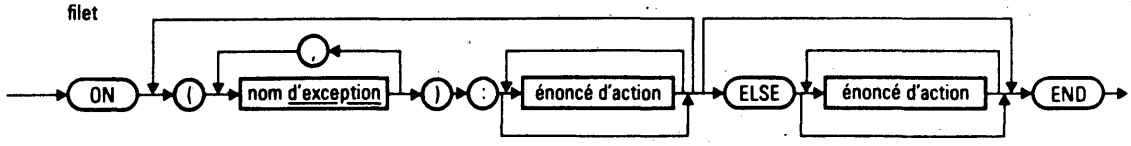
nom



operateur binaire



filet



APPENDICE F: INDEX DES REGLES DE PRODUCTION

non terminal	défini section	page	employé page(s)
<action>	6.1	107	107
<action affirmer>	6.10	123	107
<action aller>	6.9	123	107
<action appeler>	6.7	119	107
<action arrêter>	6.14	125	107
<action causer>	6.12	124	107
<action conditionnelle>	6.3	109	107
<action continuer>	6.15	125	107
<action d'affectation>	6.15	108	107
<action d'affectation multiple>	6.2	108	108
<action d'affectation simple>	6.2	108	108
<action de cas>	6.4	110	107
<action démarrer>	6.13	124	107
<action envoyer>	6.18.1	127	107
<action envoyer signal>	6.18.2	127	127
<action envoyer tampon>	6.18.3	128	127
<action faire>	6.5.1	112	107
<action mettre en attente>	6.16	125	107
<action mettre en attente et choisir>	6.17	126	107
<action parenthésée>	6.1	107	107
<action recevoir et choisir>	6.19.1	129	107
<action recevoir signal et choisir>	6.19.2	130	129
<action recevoir tampon et choisir>	6.19.3	131	129
<action résulter>	6.8	122	107
<action revenir>	6.8	122	107
<action sortir>	6.6	119	107
<action vide>	6.11	124	107
<apostrophe>	5.2.4.7	75	75
<appel de procédure>	6.7	119	64,90,119
<appel de procédure rendant locus>	4.2.10	64	56
<appel de procédure rendant valeur>	5.2.15	90	90
<appel d'opération prédéfinie>	11.1	189	64,90,119
<appel d'opération prédéfinie par CHILL rendant valeur>	5.2.16	90	70
<appel d'opération prédéfinie rendant locus>	4.2.11	64	56
<appel d'opération prédéfinie rendant valeur>	5.2.16	90	70
<argument pour getstack>	5.2.16	91	91
<attribut de paramètre>	3.7	29	29
<attributs de procédure>	7.4	139	139

non-terminal	défini section	page	employé page(s)
<début>	4.2.13	65	65,84
<déclaration>	4.1.1	53	53
<déclaration de loc-identité>	4.1.3	55	53
<déclaration de locus>	4.1.2	53	53
<déclaration de locus avec base>	4.1.4	55	53
<définition de mode>	3.2.1	18	20
<définition d'entrée>	7.4	139	139
<définition de procédure>	7.4	139	139
<définition de processus>	7.5	143	143
<définition de signal>	8.5	154	154
<définition de synonyme>	5.1	69	69
<dernier>	4.2.14	66	66,87
<descripteur dérepéré>	4.2.15	67	57
<directive>	2.6	14	14
<directive CHILL>	2.6	14	14
<directive de libération>	2.6	14	14
<directive d'implémentation>	---	--	14
<élément de chaîne>	4.2.5	60	56
<élément d'ensemble>	3.4.5	24	24
<élément d'ensemble avec numéro>	3.4.5	24	24
<élément de rangée>	4.2.7	61	56
<élément de droite>	4.2.6	60	60,83
<élément de gauche>	4.2.6	60	60,83
<élément inférieur>	4.2.8	62	62,86
<élément octroyé>	9.2.6.2	181	181
<élément saisi>	9.2.6.3	182	182
<élément supérieur>	4.2.8	62	62,86
<énoncé d'action>	6.1	107	136,145
<énoncé déclaratif>	4.1.1	53	136
<énoncé de définition de neumodes>	3.2.3	20	136
<énoncé de définition de procédure>	7.4	139	136
<énoncé de définition de processus>	7.5	143	136
<énoncé de définition de signal>	8.5	154	136

non-terminal	défini section	page	employé page(s)
<énoncé de définition de synmodes>	3.2.2	19	136
<énoncé de définition de synonymes>	5.1	69	136
<énoncé définissant>	7.2	136	136
<énoncé d'entrée>	7.4	139	136
<énoncé de saisie>	9.2.6.3	182	180
<énoncé de visibilité>	9.2.6.1	180	136
<énoncé d'octroi>	9.2.6.2	181	180
<énoncé informatif>	7.2	136	136
<énumération de locus>	6.5.2	114	113
<énumération de valeur>	6.5.2	113	113
<énumération ensembliste>	6.5.2	114	113
<énumération par intervalle>	6.5.2	113	113
<énumération par pas>	6.5.2	113	113
<espace>	5.2.4.7	75	75
<étiquette de cas>	9.1.3	169	169
<événement à choisir>	6.17	126	126
<expression>	5.3.2	97	24,25,32,34,35 38,45,46,58,59 60,61,62,65,66 67,77,83,84,85 86,87,88,89,90 91,96,97,104 106,109,110,113 114,117,118,119 123,125,127,169
<expression parenthésée>	5.3.8	106	106
<expression démarrer>	5.2.17	95	70,124
<expression recevoir>	5.2.18	95	70
<extension d'ensemble>	3.4.5	24	23
<extension d'ensemble avec numéros>	3.4.5	24	24
<extension d'ensemble sans numéros>	3.4.5	24	24
<fenêtre de saisie>	9.2.6.3	182	182
<fenêtre d'octroi>	9.2.6.2	181	181
<filet>	10.2	186	53,55,107,139 143,145
<fin>	4.2.13	65	65,84
<généralité>	7.4	139	139
<genre de chaîne>	3.10.2	34	34

non terminal	défini section	page	employé page(s)
<implantation de champ>	3.10.6	45	37,43
<implantation d'élément>	3.10.6	45	35,44
<indice supérieur>	3.10.3	35	35
<indifférent>	9.1.3	169	169
<initialisation>	4.1.2	53	53
<initialisation domaniale>	4.1.2	53	53
<initialisation viagère>	4.1.2	53	53
<intervalle>	5.2.5	77	77
<intervalle littéral>	3.4.6	25	25,35,169
<itération>	6.5.2	113	113
<lettre>	5.2.4.7	75	12,75
<liste de noms>	2.6	14	14,18,37,43,53 55,69,130,139
<liste de noms de champ>	5.2.5	77	77
<liste de noms interdits>	9.2.6.2	181	181
<liste d'énoncés d'action>	7.2	136	110,112,126,130 131,132,136,186
<liste d'énoncés informatifs>	7.2	136	136
<liste de paramètres>	3.7	29	29
<liste de paramètres d'opération prédéfinie>	11.1	189	189
<liste de paramètres effectifs>	6.7	120	95,119
<liste de paramètres formels>	7.4	139	139,143
<liste de sélecteurs de cas>	6.4	110	110
<liste d'étiquettes de cas>	9.1.3	169	77,169
<liste d'événements>	6.17	126	126
<liste d'exceptions>	3.7	30	29,139,186
<liste d'expressions>	4.2.7	61	61,85,91
<liste d'expressions littérales>	3.10.4	38	37
<liste d'intervalles>	6.4	110	110
<littéral>	5.2.4.1	72	70
<littéral binaire de chaîne de bits>	5.2.4.8	76	76
<littéral binaire d'entier>	5.2.4.2	73	73
<littéral de booléen>	5.2.4.3	74	72
<littéral de chaîne de bits>	5.2.4.8	76	72
<littéral de chaîne de caractères>	5.2.4.7	75	72
<littéral décimal d'entier>	5.2.4.2	73	73
<littéral d'entier>	5.2.4.2	73	72
<littéral d'ensemble>	5.2.4.4	74	72
<littéral de procédure>	5.2.4.6	75	72
<littéral de vide>	5.2.4.5	74	72
<littéral hexadécimal de chaîne de bits>	5.2.4.8	76	76
<littéral hexadécimal d'entier>	5.2.4.2	73	73
<littéral octal de chaîne de bits>	5.2.4.8	76	76
<littéral octal d'entier>	5.2.4.2	73	73

non terminal	défini section	page	employé page(s)
<locus>	4.2.1	56	60,61,62,63,65 66,71,89,91,95 108,114,118,124 125,126,128,130 130,131,132,180
<locus de mode dynamique>	4.2.1	56	56
<locus de mode statique>	4.2.1	56	55,56,65,91,120 122
<locus repéré>	5.2.13	89	70
<longueur>	3.10.6	46	45
<longueur de chaîne>	3.10.2	34	34,60,83
<longueur de rangée>	4.2.8	62	62,86
<longueur de tampon>	3.9.3	32	32
<longueur d'événement>	3.9.2	32	32
<marqueurs>	3.10.4	37	37,43
<mode>	3.3	21	18,28,29,30,32 35,37,43,53,55 69,154
<mode booléen>	3.4.3	22	21
<mode caractère>	3.4.4	23	21
<mode chaîne>	3.10.2	34	29,33
<mode chaîne paramétré>	3.10.2	34	34
<mode composé>	3.10.1	33	21
<mode définissant>	3.2.1	18	18
<mode descripteur>	3.6.4	29	27
<mode des éléments>	3.10.3	35	35
<mode des éléments de tampon>	3.9.3	32	32
<mode de synchronisation>	3.9.1	31	21
<mode d'indice>	3.10.3	35	35,43
<mode discret>	3.4.1	21	21,27,35,110,113
<mode ensemble>	3.4.5	23	21
<mode ensembliste>	3.5	27	21
<mode entier>	3.4.2	22	21
<mode événement>	3.9.2	32	31
<mode exemplaire>	3.8	31	21
<mode intervalle>	3.4.6	25	21
<mode primitif>	3.5	27	27
<mode procédure>	3.7	29	21
<mode rangée>	3.10.3	35	29,33
<mode rangée paramétré>	3.10.3	35	35

non terminal	défini section	page	employé page(s)
<mode repère>	3.6.1	27	21
<mode repéré>	3.6.2	28	28
<mode repère libre>	3.6.3	28	27
<mode repère lié>	3.6.2	28	27
<mode simple>	3.3	21	21
<mode structure>	3.10.4	37	33
<mode structure emboîtée>	3.10.4	37	37
<mode structure étagée>	3.10.5	43	37
<mode structure paramétré>	3.10.4	37	37
<mode tampon>	3.9.3	32	31
<module>	7.6	144	107
<mot>	3.10.6	45	45
<multiplet>	5.2.5	77	70
<multiplet de rangée>	5.2.5	77	77
<multiplet de rangée avec indices>	5.2.5	77	77
<multiplet de rangée sans indices>	5.2.5	77	77
<multiplet de structure>	5.2.5	77	77
<multiplet de structure avec noms de champ>	5.2.5	77	77
<multiplet de structure sans noms de champ>	5.2.5	77	77
<multiplet ensembliste>	5.2.5	77	77
<nom>	2.2	12	14,22,23,24,25 27,28,29,30,31 32,34,35,37,55 57,58,59,63,65 71,74,75,77,88 89,91,95,96,107 113,119,123,127 130,132,139,143 145,154,169,181 182,189
<nom d'accès>	4.2.2	57	56
<nom de mode chaîne originel>	3.10.2	34	34
<nom de mode rangée originel>	3.10.3	35	35
<nom de mode structure variable originel>	3.10.4	37	37
<nom de modulion>	9.2.6.3	182	182
<nom de valeur>	5.2.3	71	70
<nom d'exception>	3.7	30	30,124

non terminal	défini section	page	employé page(s)
			27,28,29,30,31
<opérande-1>	5.3.3	99	97,99
<opérande-2>	5.3.4	100	99,100
<opérande-3>	5.3.5	101	100,101
<opérande-4>	5.3.6	103	101,103
<opérande-5>	5.3.7	104	103
<opérande-6>	5.3.8	106	104
<opérateur-3>	5.3.4	100	100
<opérateur-4>	5.3.5	101	101
<opérateur affectant>	6.2	108	108
<opérateur arithmétique additif>	5.3.5	101	101,108
<opérateur arithmétique multiplicatif>	5.3.6	103	103,108
<opérateur binaire fermé>	6.2	108	108
<opérateur d'appartenance>	5.3.4	100	100
<opérateur de concaténation de chaîne>	5.3.5	101	101
<opérateur de différence ensembliste>	5.3.5	102	101,108
<opérateur de répétition de chaîne>	5.3.7	104	104
<opérateur d'inclusion ensembliste>	5.3.4	100	100
<opérateur nullaire>	5.2.19	96	70
<opérateur relationnel>	5.3.4	100	100
<opérateur unaire>	5.3.7	104	104
<paramètre d'opération prédéfinie>	11.1	189	189
<paramètre effectif>	6.7	120	120
<paramètre formel>	7.4	139	139
<partie avec>	6.5.4	118	112
<pas>	3.10.6	45	45
<pos>	3.10.6	45	45
<position>	4.2.6	60	60,83
<premier>	4.2.14	66	66,87
<priorité>	6.16	125	125,126,127,128
<programme>	7.8	145	--

non terminal	défini section	page	employé page(s)
<région>	7.7	145	136,145
<repère libre dérepéré>	4.2.4	59	56
<repère lié dérepéré>	4.2.5	58	56
<résultat>	6.8	122	122
<signal à choisir>	6.19.2	130	130
<sous-chaîne>	4.26	60	56
<sous-expression>	5.3.2	97	97
<sous-opérande-1>	5.3.3	99	99
<sous-opérande-2>	5.3.4	100	100
<sous-opérande-3>	5.3.5	101	101
<sous-opérande-4>	5.3.6	103	103
<sous-rangée>	4.2.8	62	56
<spec de paramètre>	3.7	29	29 139
<spec de résultat>	3.7	30	29,139
<spécification de rangée>	3.10.5	43	43
<spécification d'étiquettes de cas>	9.1.3	169	37,43,110
<symbole>	5.2.4.7	75	75
<symbole d'affectation>	6.2	108	53,55,108,113
<taille de pas>	3.10.6	45	45
<taille de patron>	3.10.6	45	45
<tranche de chaîne>	4.2.13	65	56
<tranche de rangée>	4.2.14	66	57
<tampon à choisir>	6.19.3	132	131
<valeur>	5.3.1	96	53,69,77,108,120 122,127,128,189
<valeur anonyme>	3.4.5	24	24
<valeur champ de structure>	5.2.12	88	70
<valeur de pas>	6.5.2	113	113
<valeur élément de chaîne>	5.2.6	83	70
<valeur élément de rangée>	5.2.9	85	70
<valeur finale>	6.5.2	113	113
<valeur indéfinie>	5.3.1	96	96
<valeur initiale>	6.5.2	113	113
<valeur primitive>	5.2.1	70	106
<valeur sous-chaîne>	5.2.7	83	70
<valeur sous-rangée>	5.2.10	86	70
<valeur tranche de chaîne>	5.2.8	84	70
<valeur tranche de rangée>	5.2.11	87	70
<vide>	6.11	124	124,136

APPENDICE G: INDEX

Imprimé en Suisse — ISBN 92-61-01122-5