UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

LIVRE ROUGE

TOME VI – FASCICULE VI.12

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

RECOMMANDATION Z.200

VIII^e ASSEMBLÉE PLÉNIÈRE
MALAGA-TORREMOLINOS, 8-19 OCTOBRE 1984

Genève 1985

UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

LIVRE ROUGE

TOME VI – FASCICULE VI.12

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

RECOMMANDATION Z.200

VIII^e ASSEMBLÉE PLÉNIÈRE
MALAGA-TORREMOLINOS, 8-19 OCTOBRE 1984

Genève 1985

ISBN 92-61-02252-9

CONTENU DU LIVRE DU CCITT
EN VIGUEUR APRÈS LA HUITIÈME ASSEMBLÉE PLÉNIÈRE (1984)

LIVRE ROUGE

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

Vœux et résolutions.

Recommandations 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 des Questions mises à l'étude.

Tome II – *(Divisé en 5 fascicules vendus séparément)*

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

FASCICULE II.2 – Service téléphonique international – Exploitation – Recommandations E.100 à E.323 (Commission d'études II).

FASCICULE II.3 – Service téléphonique international – Gestion du réseau – Ingénierie du trafic – Recommandations E.401 à E.600 (Commission d'études II).

FASCICULE II.4 – Services télégraphiques – Exploitation et qualité de service – Recommandations F.1 à F.150 (Commission d'études I).

FASCICULE II.5 – Services de télématique – Exploitation et qualité de service – Recommandations F.160 à F.350 (Commission d'études I).

Tome III – *(Divisé en 5 fascicules vendus séparément)*

FASCICULE III.1 – Caractéristiques générales des communications et des circuits téléphoniques internationaux – Recommandations G.101 à G.181 (Commissions d'études XV, XVI et CMBD).

FASCICULE III.2 – Systèmes internationaux analogiques à courants porteurs – Caractéristiques des moyens de transmission – Recommandations G.211 à G.652 (Commissions d'études XV et CMBD).

FASCICULE III.3 – Réseaux numériques – Systèmes de transmission et équipement de multiplexage – Recommandations G.700 à G.956 (Commissions d'études XV et XVIII).

FASCICULE III.4 – Utilisation des lignes pour les transmissions des signaux autres que téléphoniques – Transmissions radiophoniques et télévisuelles – Recommandations des séries H et J (Commission d'études XV).

FASCICULE III.5 – Réseau numérique avec intégration des services (RNIS) – Recommandations de la série I (Commission d'études XVIII).

Tome IV – *(Divisé en 4 fascicules vendus séparément)*

FASCICULE IV.1 – Maintenance: principes généraux, systèmes de transmission internationaux, circuits téléphoniques internationaux – Recommandations M.10 à M.762 (Commission d'études IV).

FASCICULE IV.2 – Maintenance des circuits internationaux pour la transmission de télégraphie harmonique ou de télécopie – Maintenance des circuits internationaux loués – Recommandations M.800 à M.1375 (Commission d'études IV).

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

FASCICULE IV.4 – Spécifications des appareils de mesure – Recommandations de la série O (Commission d'études IV).

Tome V – Qualité de la transmission téléphonique – Recommandations de la série P (Commission d'études XII).

Tome VI – *(Divisé en 13 fascicules vendus séparément)*

FASCICULE VI.1 – Recommandations générales sur la commutation et la signalisation téléphoniques – Interface avec le service maritime et le service mobile terrestre – Recommandations Q.1 à Q.118 *bis* (Commission d'études XI).

FASCICULE VI.2 – Spécifications des Systèmes de signalisation n° 4 et 5 – Recommandations Q.120 à Q.180 (Commission d'études XI).

FASCICULE VI.3 – Spécifications du Système de signalisation n° 6 – Recommandations Q.251 à Q.300 (Commission d'études XI).

FASCICULE VI.4 – Spécifications des Systèmes de signalisation R1 et R2 – Recommandations Q.310 à Q.490 (Commission d'études XI).

FASCICULE VI.5 – Centraux numériques de transit dans les réseaux numériques intégrés et les réseaux mixtes analogiques-numériques. Centraux numériques locaux et mixtes – Recommandations Q.501 à Q.517 (Commission d'études XI).

FASCICULE VI.6 – Interfonctionnement des systèmes de signalisation – Recommandations Q.601 à Q.685 (Commission d'études XI).

FASCICULE VI.7 – Spécifications du Système de signalisation n° 7 – Recommandations Q.701 à Q.714 (Commission d'études XI).

FASCICULE VI.8 – Spécifications du Système de signalisation n° 7 – Recommandations Q.721 à Q.795 (Commission d'études XI).

FASCICULE VI.9 – Système de signalisation avec accès numérique – Recommandations Q.920 à Q.931 (Commission d'études XI).

FASCICULE VI.10 – Langage de spécification et de description fonctionnelles (LDS) – Recommandations Z.101 à Z.104 (Commission d'études XI).

FASCICULE VI.11 – Langage de spécification et de description fonctionnelles (LDS), annexes aux Recommandations Z.101 à Z.104 (Commission d'études XI).

FASCICULE VI.12 – Langage évolué du CCITT (CHILL) – Recommandation Z.200 (Commission d'études XI).

FASCICULE VI.13 – Langage homme-machine (LHM) – Recommandations Z.301 à Z.341 (Commission d'études XI).

Tome VII – *(Divisé en 3 fascicules vendus séparément)*

FASCICULE VII.1 – Transmission télégraphique – Recommandations de la série R (Commission d'études IX). – Equipements terminaux pour les services de télégraphie – Recommandations de la série S (Commission d'études IX).

FASCICULE VII.2 – Commutation télégraphique – Recommandations de la série U (Commission d'études IX).

FASCICULE VII.3 – Equipements terminaux et protocoles pour les services de télématique – Recommandations de la série T (Commission d'études VIII).

Tome VIII – *(Divisé en 7 fascicules vendus séparément)*

FASCICULE VIII.1 – Communication de données sur le réseau téléphonique – Recommandations de la série V (Commission d'études XVII).

FASCICULE VIII.2 – Réseaux de communications de données; services et facilités – Recommandations X.1 à X.15 (Commission d'études VII).

FASCICULE VIII.3 – Réseaux de communications de données; interfaces – Recommandations X.20 à X.32 (Commission d'études VII).

FASCICULE VIII.4 – Réseaux de communications de données; transmission, signalisation et commutation, réseau, maintenance et dispositions administratives – Recommandations X.40 à X.181 (Commission d'études VII).

FASCICULE VIII.5 – Réseaux de communications de données: interconnexion de systèmes ouverts (OSI), techniques de description du système – Recommandations X.200 à X.250 (Commission d'études VII).

FASCICULE VIII.6 – Réseaux de communications de données: interfonctionnement entre réseaux, systèmes mobiles de transmission de données – Recommandations X.300 à X.353 (Commission d'études VII).

FASCICULE VIII.7 – Réseaux de communications de données: systèmes de traitement des messages – Recommandations X.400 à X.430 (Commission d'études VII).

Tome IX – Protection contre les perturbations – Recommandations de la série K (Commission d'études V) – Construction, installation et protection des câbles et autres éléments d'installations extérieures – Recommandations de la série L (Commission d'études VI).

Tome X – *(Divisé en 2 fascicules vendus séparément)*

FASCICULE X.1 – Termes et définitions.

FASCICULE X.2 – Index du Livre rouge.

TABLE DES MATIÈRES DU FASCICULE VI.12 DU LIVRE ROUGE

Partie I – Recommandation Z.200

Langage évolué du CCITT (CHILL)

Partie II – Suppléments à la Recommandation Z.200

PARTIE I

Recommandation Z.200

Langage évolué du CCITT (CHILL)

Recommandation Z.200

LANGAGE ÉVOLUÉ DU CCITT (CHILL)

(Genève, 1984)

TABLE DES MATIERES

1 INTRODUCTION

La présente Recommandation 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 le manuel CCITT intitulé "Définition formelle de CHILL". Un autre manuel CCITT, intitulé "Introduction to CHILL", sert d'introduction au langage.

Le présent chapitre donne une description informelle des possibilités de CHILL. La définition du langage commence au chapitre 2.

1.1 GÉNÉRALITÉS

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 à utiliser.

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 GÉNÉRALE DU LANGAGE

Un **programme** CHILL se compose essentiellement de trois parties:

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

Les objets informatifs sont décrits par des **énoncés informatifs** (énoncés déclaratifs et définissant), 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

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

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

CHILL a les catégories de mode suivantes:

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

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 ACCÈS

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

Les **énoncés déclaratifs** définissent les noms à employer pour accéder à un locus.

Ce sont:

1. les déclarations de locus;
2. les déclarations de loc-identité;
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'est-à-dire 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 OPÉRATIONS

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

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

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

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

Les **énoncés de définition de 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 séquentielles suivantes:

l'action conditionnelle pour une 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 une 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 une 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 ENTRÉE ET SORTIE

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

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

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

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

1.8 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'est-à-dire où l'objet qu'il dénote est identifié par le nom.

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

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

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

Les **processus** et les **régions** fournissent les moyens de réaliser une structure d'exécutions concurrentes.

Un **programme** CHILL complet est une liste de modules 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.

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

1.9 EXÉCUTION 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.

1.10 PROPRIÉTÉS SÉMANTIQUES GÉNÉRALES

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

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

Les règles de portée définissent la visibilité des noms, déterminée par la structure du programme et par des énoncés explicites de visibilité. Ces derniers déterminent la visibilité des noms qui y sont mentionnés et aussi d'éventuels **noms impliqués** des noms mentionnés.

Les noms introduits dans un programme ont un endroit où ils sont définis ou déclarés. Cet endroit est appelé l'**occurrence de définition** du nom. Les endroits où le nom est utilisé sont appelés **occurrences d'utilisation** du nom. Les règles d'**identification** associent une occurrence de définition unique à chaque occurrence d'utilisation d'un nom.

1.11 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'est-à-dire si l'exception a un nom) et spécifié. On peut déterminer statiquement si un filet est ou non spécifié pour une exception en un point donné. Si aucun filet explicite n'est spécifié, le contrôle peut être transmis à un filet d'exception défini par l'implémentation.

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

1.12 OPTIONS POUR L'IMPLÉMENTATION

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

Un mode entier défini par l'implémentation doit être dénoté par un nom de mode défini par l'implémentation. Ce nom est considéré comme défini dans un énoncé de définition de neumode non spécifié en CHILL. Il est permis d'étendre aux modes entiers définis par l'implémentation les opérations arythmétiques 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 occurence d'une exception, l'implémentation peut décider des actions à accomplir.

Si une condition dynamique définie par l'implémentation est violée, il en résulte une exception définie par l'implémentation.

2 PRÉLIMINAIRES

2.1 MÉTALANGAGE

La description 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 }). Un groupe entre accolades peut contenir une ou plusieurs barres verticales, indiquant le choix entre des éléments syntaxiques. 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 sémantique

Pour chaque catégorie syntaxique (symbole non terminal), la description sémantique est donnée dans les sections intitulées **sémantique**, **propriétés statiques**, **propriétés dynamiques**, **conditions statiques** et **conditions dynamiques**.

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

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

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

La section **conditions statiques** décrit les conditions dépendant du contexte contrôlables statiquement qui doivent être remplies lorsque la catégorie syntaxique est utilisée. Certaines conditions statiques sont exprimées dans la syntaxe au moyen d'une partie soulignée du symbole non terminal (voir 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 Règles d'identification dans le métalangage

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, Recommandation 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.

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

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

syntaxe:

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

Le caractère souligné (_) fait partie de la représentation textuelle de nom simple, c'est-à-dire que la représentation textuelle de nom simple *life_time* est différente de la représentation textuelle de nom simple *lifetime.* Dans le cas où un alphabet comprenant les lettres minuscules est disponible, celles-ci peuvent être illustrées dans les représentations textuelles de nom simple. Lettres majuscules et minuscules sont différentes, par exemple, *Status* et *status* sont deux représentations textuelles de nom simple différentes.

Le langage possède un certain nombre de représentations textuelles de nom simple **spéciales** à signification prédéterminée, voir Appendice C. Certaines d'entre elles sont **réservées**, c'est-à-dire qu'elles ne peuvent être utilisées pour d'autres usages que si elles sont explicitement libérées par la directive de libération.

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

En ce qui concerne les **qualificateurs de littéraux** (voir Appendice B) et les lettres d'un *chiffre hexadécimal* (voir section 5.2.4.2) l'implémentation doit permettre d'utiliser soit des lettres majuscules, soit des lettres majuscules et minuscules.

2.3 ESPACEMENTS

Un espace délimite tout élément lexical ou toute combinaison de caractères spéciaux. Les éléments lexicaux sont terminés par le premier caractère qui ne peut pas en faire partie. Par exemple, *IFBTHEN* sera considéré comme *représentation textuelle de nom simple* et non comme le début d'une action **IF** *B* **THEN** , //* sera considéré comme le symbole de concaténation (//) suivi d'un astérisque (*) et non comme un symbole de division (/) suivi du crochet ouvrant d'un commentaire (/*). 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 représentations textuelles de noms simples réservés>) *(4.1)*

<liste de représentations textuelles de noms simples> ::= (5)
<représentation textuelle de nom simple>
{,*<représentation textuelle de nom simple>*} * (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'est-à-dire qu'un programme contenant des directives d'implémentation est correct, au sens de CHILL, si et seulement si il est correct sans ces directives.

Une directive de libération libérera les représentations textuelles de noms simples **réservées** spécifiées dans la *liste de représentations textuelles de noms simples réservées* de telle façon qu'elles puissent être redéfinies.

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 10.2).

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)

2.7 NOMS ET LEURS DÉFINITIONS

syntaxe:

<nom> ::= (1)
<représentation textuelle de nom> (1.1)

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

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

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

<préfixe simple> ::= (5)
<représentation textuelle de nom simple> (5.1)

<définition> ::= (6)
<représentation textuelle de nom simple> (6.1)

<liste de définitions> ::= (7)
<définitions >{, *<définitions>* }* (7.1)

<nom de champ> ::= (8)
<représentation textuelle de nom simple> (8.1)

<définition du nom de champ> ::= (9)

<représentation textuelle de nom simple> (9.1)

<liste de définitions de nom de champ> ::= (10)
<définition de nom de champ> {, <définition de nom de champ> } * (10.1)

<nom d'exception> ::= (11)
<représentation textuelle de nom simple> (11.1)
| <représentation textuelle de nom préfixe> (11.2)

<nom de registre> ::= (12)
<représentation textuelle de nom simple > (12.1)
| <représentation textuelle de nom préfixe> (12.2)

<nom de repère de texte> ::= (13)
<représentation textuelle de nom simple> (13.1)
| <représentation textuelle de nom préfixe> (13.2)

<nom de repère d'implantation> ::= (14)
<représentation textuelle de nom simple> (14.1)
| <représentation textuelle de nom préfixe> (14.2)

syntaxe dérivée: Dans le jeu de caractères réduit, /. est utilisé pour remplacer ! (opérateur de préfixe).

sémantique: Les noms d'un programme désignent des objets. Etant donné l'occurrence d'un *nom* (formellement: l'occurrence d'une production terminale de *nom*) dans un programme, les règles d'identification de la section 10.2 donnent les *définitions* auxquelles ce (cette occurrence de) *nom* est identifié (formellement: occurrences de productions terminales de *définition*). Ainsi, le *nom* désigne l'objet défini ou déclaré par les *définitions*. (Il ne peut y avoir plus d'une *définition* pour un *nom* que dans le cas de *noms d'éléments d'ensemble* ou de *noms* avec *quasi-définitions*.) On dit des *définitions* qu'elles définissent le *nom*.

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

Les *noms d'exception* sont utilisés pour identifier des filets d'exceptions selon les règles énoncées dans le chapitre 11.

Les *noms de registre* servent à identifier des registres d'une manière définie par l'implémentation (voir section 8.4).

Les *noms de référence de texte* servent à identifier les parties de texte source d'une manière définie par l'implémentation, sous réserve des règles énoncées dans la section 8.10.1.

Les *noms de référence d'implantation* servent à spécifier l'implantation d'une manière définie par l'implémentation (voir section 3.11.6).

Lorsqu'un *nom* est identifié à plus d'une *définition*, chacune des *définitions* auxquelles le *nom* est identifié définit ou énonce le même objet (voir les règles exactes en 10.2.2 et 8.10).

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

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

Dans le reste de ce document:

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

- la représentation textuelle de nom d'une *définition*, d'un *nom de champ* ou d'une *définition de nom de champ* sert à désigner la représentation textuelle de nom canonique de la *représentation textuelle de nom simple* dans cette *définition*, ce *nom de champ* ou cette *définition de nom de champ*, respectivement.

Les règles d'identification sont telles que:

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

- les *noms* appartenant à une *représentation textuelle de nom préfixée* sont identifiés aux *définitions* d'une *représentation textuelle de nom* identique à la *représentation textuelle de nom simple*, précisément, de la *représentation textuelle de nom préfixée* du *nom*;

- les *noms de champ* sont identifiés à des *définitions* de *nom de champ* ayant la même *représentation textuelle de nom* que les *noms de champ*.

définition de notation: Soit une *représentation textuelle de nom* NS, et une chaîne de caractères P, soit avec un *préfixe* soit vide, on définit que le résultat du préfixe NS par P, écrit P ! NS, est le suivant:

- si P est vide, P ! NS est NS;

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

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

propriétés statiques: Un *nom* a les propriétés statiques rattachées aux *définitions* auxquelles il est identifié. Un *nom de champ* a les propriétés statiques rattachées à la *définition de nom de champ* à laquelle il est identifié.

3 MODES ET CLASSES

3.1 GÉNÉRALITÉS

A un locus est attaché un mode, à une valeur une classe. Le mode attaché à un locus définit l'ensemble des valeurs que le locus peut contenir, les méthodes d'accès au locus et les opérations permises sur les valeurs. La classe attachée à une valeur est un moyen de déterminer les modes des locus qui peuvent contenir la valeur. Certaines valeurs sont **fortes**. A une valeur **forte**, on attache une classe et un mode. 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 ont 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 vituelles 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 un mode M, il peut exister la **M-classe par valeur**. Toutes les valeurs d'une telle classe et seules ces valeurs sont **fortes** et le mode attaché à ces valeurs est M.
- Pour un mode M, il peut exister la **M-classe par dérivation**.
- Pour tout mode M, il existe la **M-classe par repère**.
- La **classe nulle**.
- La **classe toute**.

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

3.1.3 Propriétés des modes, des classes et leurs relations

Les modes CHILL ont des propriétés. Celles-ci peuvent être héréditaires ou non héréditaires. Une propriété héréditaire est transmise d'un mode définissant à un nom de mode défini par celui-ci. Un résumé est donné ci-après des propriétés qui s'appliquent à tous les modes (sauf pour la première, elles sont toutes définies dans la section 10.1):

1. Un mode a une **nouveauté** (définie dans les sections 3.2.2, 3.2.3 et 3.3).
2. Un mode peut avoir la **propriété d'être protégé**.

3. Un mode peut être **paramétrable**.

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

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

6. Un mode peut avoir la **propriété de non-valeur**.

En CHILL, des classes peuvent avoir des propriétés suivantes (définies dans la section 10.1):

1. Une classe peut avoir un mode **racine**.

2. Une ou plusieurs classes peuvent avoir une **classe résultante**.

En CHILL, les opérations sont déterminées par les modes et les classes des locus et des valeurs. Cela est exprimé par les règles de vérification de mode définies dans la section 10.1 sous la forme d'un certain nombre de relations entre les modes et les classes. Les relations suivantes peuvent exister:

1. Deux modes peuvent être **similaires**.

2. Deux modes peuvent être **v-équivalents**.

3. Deux modes peuvent être **équivalents**.

4. Deux modes peuvent être **l-équivalents**.

5. Deux modes peuvent être **semblables**.

6. Deux modes peuvent être **N-semblables**.

7. Deux modes peuvent être **compatibles en lecture**.

8. Deux modes peuvent être **compatibles en lecture dynamique**.

9. Un mode peut être **limitable** à un mode.

10. Un mode peut être **compatible** avec une classe.

11. Une classe peut être **compatible** avec une classe.

3.2 DÉFINITIONS DE MODES

3.2.1 Généralités

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

NEWMODE *dollar, pound = INT* ;

est dérivé de:

NEWMODE *dollar = INT , pound = INT* ;

sémantique: Les définitions de mode définissent un ou plusieurs noms comme étant des noms de mode. La plupart des propriétés d'un nom de mode sont héritées de son mode définissant.
Des définitions de mode apparaissent dans les exposés de définition de synmodes et de neumodes. Une définition de synmode est synonyme de son mode définissant. Une définition de neumode n'est pas synonyme de son mode définissant. La différence est définie en fonction de la nouveauté de la propriété, qui est utilisée dans la vérification de mode (voir section 10.1).

propriétés statiques: Dans une *définition de mode* une *définition* définit un nom de **mode**.

Les noms de **mode** sont aussi les noms de mode prédéfinis *INT* , *BIN* , *BOOL* , *CHAR* , *PTR*, *INSTANCE*, *ASSOCIATION* et les noms de mode d'entier définis par l'implémentation (s'il en existe voir section 3.4.2).

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

Un ensemble de définitions récursives est un ensemble de définitions de modes 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'endroit marqué.

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

exemples:

1.15 *operand_mode = INT* *(1.1)*
3.3 *complex =* **STRUCT** *(re,im INT)* *(1.1)*

3.2.2 Définitions de synmodes

syntaxe:

<énoncé de définition de 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.

propriétés statiques: Une *définition* figurant dans une *définition de mode* dans un *énoncé de définition de synmodes* définit un nom de **synmode** (qui est aussi un nom de **mode**). 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.

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

exemples:

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

3.2.3 Définitions des 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 de modes qui ne sont pas **synonyme** de leur mode définissant.

propriétés statiques: Une *définition* apparaissant dans une *définition de mode* apparaissant dans un *énoncé de définition neumode* définit un nom de **neumode** (qui est aussi un nom de **mode**).
La **nouveauté** du nom de **neumode** est la *définition* qui le définit.
Si le **mode définissant** du nom de **neumode** est un mode intervalle, le mode virtuel *&nom* est introduit comme le mode **parent** du nom de **neumode**. Le **mode définissant** de *&nom* est le mode **parent** du mode intervalle et la **nouveauté** de *&nom* est nom de **neumode**.

exemples:

11.6 **NEWMODE** *line = INT (1:8);* *(1.1)*
11.12 **NEWMODE** *board* = **ARRAY** *(line)* **ARRAY** *(column) square;* *(1.1)*

3.3 CLASSIFICATION DES MODES

syntaxe:

<mode> ::= *(1)*
[**READ**] *<mode simple >* *(1.1)*
| [**READ**] *<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)*
| *<mode d'entrée-sortie>* *(2.7)*

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

propriétés statiques: Un mode a les propriétés héréditaires suivantes:

- Il est un mode **protégé** s'il est explicitement ou implicitement **protégé**.
- Il est un mode **protégé** explicitement si **READ** est spécifié ou s'il est mode rangée **paramétré**, un mode chaîne **paramétré** ou un mode structure **paramétré** dans lequel le nom de mode rangée **originel**, le nom de mode chaîne **originel** ou le nom de mode structure **variante originel**, respectivement, est un mode **protégé**.
- Il est un mode implicitement **protégé** si
  - c'est le mode **élément** d'un mode rangée **protégé** (voir section 3.11.3);
  - c'est un mode **champ** d'un mode structure **protégé** ou le mode d'un **champ marqueur** d'un mode structure **paramétré** (voir section 3.11.4).

Les modes **protégés** ont les mêmes propriétés que leurs modes correspondants non **protégés** à l'exception de la **propriété d'être protégé** (voir section 10.1.1).

Un mode a les propriétés non héréditaires suivantes:

- Il a une **nouveauté** qui est soit **nulle** soit la *définition* existant dans une *définition de mode* figurant dans un *énoncé de définition de neumode*. La **nouveauté** d'un mode qui n'est pas un *nom de mode* (ni un *nom de mode* **READ**) est définie comme suit:
  - si c'est un mode chaîne **paramétré**, un mode rangée **paramétré** ou un mode structure **paramétré**, sa **nouveauté** est celle de son nom de mode chaîne **originel**, de son nom de mode rangée **originel** ou de son nom de mode structure **variante originel**, respectivement;
  - si c'est un mode rangée, sa **nouveauté** est celle de son mode **parent**;
  - sinon, sa **nouveauté** est **nulle**.

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

3.4 MODES DISCRETS

3.4.1 Généralités

syntaxe:

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

sémantique: Les modes 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, une borne inférieure et un nombre de valeurs.

3.4.2 Modes entier

syntaxe:

<mode entier> ::= *(1)*
INT *(1.1)*
| *BIN* *(1.2)*
| *<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). 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 12.2).

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

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

exemples:

1.5 *INT* *(1.1)*

3.4.3 Modes booléen

syntaxe:

<mode booléen> ::= *(1)*
BOOL *(1.1)*
| *<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). *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 caractère

syntaxe:

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

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

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

<élément d'ensemble avec numéro> ::= *(4)*
<nom> = *<nom 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 *définitions* apparaissant dans l'*extension d'ensemble*; les valeurs anonymes sont les autres valeurs. La représentation interne d'une valeur nommée est la valeur entière associée à la valeur nommée (voir ci-dessous). Cette représentation définit également l'ordre des valeurs.

propriétés statiques: Une *définition* d'une *extension d'ensemble* définit un nom **d'élément d'ensemble**.

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 la *définition 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:

11.7	**SET** *(occupied, free)*	*(1.1)*
6.3	*month*	*(1.2)*

3.4.6 Modes intervalle

syntaxe:

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

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

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

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

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

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

propriétés statiques: Un mode intervalle a la propriété 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 (U)- NUM (L) + 1*, où *U* et *L* 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.

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

exemples:

9.5	*INT (2:max)*	*(1.1)*
11.12	*line*	*(1.4)*
9.5	*2:max*	*(2.1)*

3.5 MODES ENSEMBLISTES

syntaxe:

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

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

sémantique: Un mode ensembliste définit des valeurs qui sont des ensembles de valeurs de son mode primitif. Les valeurs 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 primitif.*

exemples:

8.4	**POWERSET** *CHAR*	*(1.1)*
9.5	**POWERSET** *INT (2:max)*	*(1.1)*
9.6	*number_list*	*(1.2)*

3.6 MODES REPÈRE

3.6.1 Généralités

syntaxe:

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

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

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

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

3.6.2 Modes repère lié

syntaxe:

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

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

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

propriétés statiques: Un mode repère lié a la propriété héréditaire suivante:

- Il a un mode **repéré** unique qui lui est attaché et qui est dénoté par *mode repéré.*

exemples:

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

3.6.3 Modes repère libre

syntaxe:

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

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

exemples:

19.8 *PTR* *(1.1)*

3.6.4 Modes descripteur

syntaxe:

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

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

Une valeur descripteur peut repérer:

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

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

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

8.6 **ROW** *CHAR (max)* *(1.1)*

3.7 MODES PROCÉDURE

syntaxe:

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

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

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

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

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

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

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

syntaxe dérivée: Une *spec de résultat* sans représentation textuelle de nom **réservé** facultative **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.

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

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

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

3.8 MODES EXEMPLAIRE

syntaxe:

<mode exemplaire> ::= *(1)*
INSTANCE *(1.1)*
| *<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 5.2.14, 6.13 et 9.1) produit une valeur exemplaire unique comme identification pour le processus créé.

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

exemples:

15.39 *INSTANCE* *(1.1)*

3.9 MODES DE SYNCHRONISATION

3.9.1 Généralités

syntaxe:

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

sémantique: Les locus d'un mode de synchronisation donnent des moyens de synchronisation des processus et de communication entre eux (voir chapitre 9). Il n'existe pas d'expressions en CHILL dénotant une valeur définie par un mode de synchronisation. En conséquence, il n'y a pas d'opérations définies sur ces valeurs.

3.9.2 Modes événement

syntaxe:

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

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

sémantique: 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 est la valeur rendue par *longueur d'événement.*

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

exemples:

14.10 **EVENT** *(1.1)*

3.9.3 Modes tampon

syntaxe:

<mode tampon> ::= *(1)*
BUFFER [*(<longueur de tampon>)*]*<mode des éléments de tampon>* *(1.1)*
| *<nom de mode tampon>* *(1.2)*

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

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

Note: La syntaxe donnée ci-dessus est ambiguë à cause de la syntaxe des modes rangée. L'interpréta - tion par défaut suivante est à appliquer: si **BUFFER** est immédiatement suivi d'une parenthèse ouvrante, le texte qui suit immédiatement est considéré comme le début de l'indication de la *longu - eur 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 5.3.8.

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

- Il a une **longueur de tampon**, facultative, qui est la valeur rendue par *longueur de tampon.*
- Il a un mode des **éléments de tampon** qui est le *mode des éléments de tampon.*

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

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

exemples:

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

3.10 MODES D'ENTRÉE-SORTIE

3.10.1 Généralités

syntaxe:

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

sémantique: Les modes d'entrée-sortie sont utilisés pour des opérations d'entrée-sortie définies dans le chapitre 7. Il n'existe pas dans le titre d'expression désignant une valeur définie par un mode d'entrée-sortie. Il n'y a donc pas d'opérations définies sur les valeurs.

exemples:

20.17 *ASSOCIATION* *(1.1)*

3.10.2 Modes association

syntaxe:

<mode association> ::= *(1)*
ASSOCIATION *(1.1)*
| *<nom de mode association>* *(1.2)*

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

3.10.3 Modes accès

syntaxe:

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

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

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

Note: La syntaxe indiquée ci-dessus est syntaxiquement ambiguë par rapport à la syntaxe du mode rangée. Le défaut d'interprétation suivant est applicable: si **ACCESS** est immédiatement suivi d'une parenthèse ouverte, le texte qui suit est considéré comme étant le début de la désignation *mode index* facultatif et non comme appartenant au *mode enregistrement*.

syntaxe dérivée: La notation de mode index *intervalle de littéral* est tirée du mode discret **RANGE** (*intervalle de littéral*).

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

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

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

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

- Il a un mode **enregistrement** facultatif qui est le *mode enregistrement* s'il existe. C'est un mode **enregistrement dynamique** si **DYNAMIC** est spécifié ou, autrement, un mode **enregistrement statique**.
- Il a un mode **index** facultatif, qui est le *mode index*.

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

Si **DYNAMIC** est spécifié, le mode **enregistrement** doit être **paramétrable**.

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

exemples:

20.18 **ACCESS** *(index_set) record_type* *(1.1)*
22.20 **ACCESS** *string* **DYNAMIC** *(1.1)*
20.18 *record_type* *(2.1)*
20.18 *index_set* *(3.1)*

3.11 MODES COMPOSÉS

3.11.1 Généralités

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).

3.11.2 Modes chaîne

syntaxe:

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

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

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

<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 valeur chaîne de bits, l'ordre est l'ordre lexicographique tel que un bit qui est 1 est supérieur à un bit qui est 0.

Les valeurs chaîne soit sont vides soit ont des éléments qui sont numérotés à partir de 0. Deux valeurs chaîne vides sont égales.

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 nom chaîne de **bits** ou de **caractères**.

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

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

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

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:

7.51 CHAR (20) *(1.1)*

3.11.3 Modes rangée

syntaxe:

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

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

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

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

<mode des éléments> ::= *(5)*
<mode> *(5.1)*

syntaxe dérivée: La représentation textuelle de nom simple **réservée ARRAY** est facultative. Un *mode rangée* (qui n'est ni un *nom de mode rangée* ni un *mode rangée paramétré*) sans **ARRAY** est dérivé du mode rangée avec **ARRAY** .

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 3.11.6). 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**. Le mode **des éléments** est un mode **protégé implicite** s'il est **READ** *M*.

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

- 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,
  où *borne supérieure* et *borne inférieure* sont respectivement la **borne supérieure** et la **borne inférieure** de son mode **d'indice**.

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

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

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

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

exemples:

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

3.11.4 Modes structure

syntaxe:

<mode structure> ::= *(1)*
- *<mode structure emboîtée>* *(1.1)*
- | *<mode structure étagée>* *(1.2)*
- | *<mode structure paramétré>* *(1.3)*
- | *<nom de mode structure>* *(1.4)*

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

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

<champs fixes> ::= (4)
<liste de définitions de noms de champ> <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 définitions de noms de champ> <mode>
[<implantation de champ>] (8.1)

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

<nom de mode structure variable originel> ::= (10)
<nom de mode de structurevariable> (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 3.11.5.

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

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

est dérivé de:

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

sémantique: Les modes structure définissent des valeurs composées constituées d'une liste de valeurs sélectionnables par un nom de composante. Chacune de ces valeurs est définie par un mode attaché

au nom de composante. Les valeurs structure peuvent résider dans des locus structure (composés) où le nom de composante sert d'accès au sous-locus. Les composantes d'une valeur ou d'un locus structure sont appelées **champs** et leurs noms, noms **de champ**.

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

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

Les structures variables ont des champs récurrents, c.-à-d. des champs qui ne sont pas toujours présents. Pour les structures variables avec marqueurs, la présence de ces champs est connue seulement à l'exécution d'après la ou les valeurs de certains champs fixes associés, nommés champs **marqueurs**. Les structures variables sans marqueurs n'ont pas de champs **marqueurs**. Comme la composition d'une structure variable peut changer durant l'exécution, la taille d'un locus structure variable est basée sur le cas de taille 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 3.12.4.

L'implantation d'un locus ou d'une valeur structure peut être contrôlée au moyen d'une spécification d'implantation de champs (voir section 3.11.6).

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** donné d'un mode structure donné a un mode de **champ** qui lui est attaché, et qui est soit *M* soit **READ** *M*, où *M* est le *mode* qui suit le nom **de champ**. Le mode **de champ** sera **READ** *M* si *M* n'est pas un mode **protégé** et soit que le mode structure est un mode **protégé**, soit que le champ est un **champ marqueur** d'un mode structure **paramétré**. Le mode de **champ** est un mode **protégé** implicite s'il est **READ** *M*.

  Un nom **de champ** d'un mode structure donné a une **implantation de champ** qui lui est attachée et qui est l'*implantation de champ* qui suit le nom **de champ** si elle est présente, sinon l'implantation de champ par défaut, qui est **PACK** ou **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 définitions de noms de champ* dans les *champs fixes* avec l'ensemble des noms définis par toute *liste de définitions de noms de champ* dans les *choix de champs*. Les noms **de champ** définis par une *liste de noms* dans les *champs fixes* sont les noms **de champ fixe** du mode structure **variable**, ses autres noms **de champ** sont les noms **de champ récurrent**.

  Un nom **de champ** d'un mode structure **variable** est un nom **de champ marqueur** si et seulement s'il apparaît dans 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 définitions de noms de champ* 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 déterminées comme suit:

  - si c'est un mode structure **variable avec marqueurs**, la liste des M_i -classes par valeur, où les M_i représentent les modes des noms **de champ marqueur** dans l'ordre où ils sont définis dans les *champs fixes*;

  - si c'est un mode structure **variable sans marqueurs**, la liste est construite à partir des listes individuelles résultantes des classes de chaque *choix de champs* en les concaténant dans l'ordre où les *choix de champs* apparaissent. La **liste résultante des classes** d'une occurrence de *choix de champs* est la **liste résultante des classes** de la liste d'occurrences de *spécification d'étiquettes de cas* qu'elle contient (voir section 10.1.3).

structures paramétrées:

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

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

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**.

- Il a 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.

Le *mode* de *champ récurrent* peut n'avoir ni la **propriété de non-valeur** ni celle d'être **paramétrable** avec **marqueur**.

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 10.1.3) 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)
11.7    STRUCT (status SET (occupied, free),
                    CASE status OF
                     (occupied): p piece,
                     (free):
                    ESAC )                                            (2.1)
2.6     fraction                                                      (1.4)
11.7    status SET (occupied, free)                                   (4.1)
11.8    status                                                        (7.1)
11.9    p piece                                                       (8.1)
```

3.11.5 Notation étagée 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 définitions de noms de champ> <mode>
        [ <implantation de champ> ]                                   (3.1)
      | n <liste de définitions de noms de champ> [ <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 dérivée 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 3.11.4).

Si une structure contient des champs qui sont eux-mêmes des structures ou des rangées de structures, une hiérarchie de structures 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, spécifications de paramètres et de résultats, 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.

READ peut ne pas être spécifié immédiatement en face d'un *mode structure étagée.*

exemples:

```
19.12   DCL 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)
```

3.11.6 Description d'implantation pour modes rangée et modes structure

syntaxe:

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

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

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

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

<mot> ::= (5)
<expression littérale entière > (5.1)

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

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

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

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

sémantique: Il est possible de commander l'implantation d'une rangée ou d'une structure en donnant des informations de compactage ou de représentation dans son mode. L'information de compactage est soit **PACK** , soit **NOPACK** , l'information de représentation est soit un *pas* dans le cas de modes rangée, soit un *pos* dans le cas 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 aussi la **repérabilité**.

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

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

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

définit un décalage de bit de

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

et une longueur de *NUM (longueur)* bits, où *WIDTH* est le nombre (défini par l'implémentation) de bits d'un mot et *mot* est soit une *expression littérale entière* soit un *nom repère d'implantation* donnant une valeur entière définie par l'implémentation.

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

Un *pas* ayant la forme

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

définit une série de décalage de bits b_i lorsque i prend les valeurs 0 à $n-1$, où n est le nombre d'éléments de la rangée et

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

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

Défauts

La notation:

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

est sémantiquement équivalente à:

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

La notation:

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

est sémantiquement équivalente à:

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

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

La notation:

POS *(<numéro de mot>)*

est sémantiquement équivalente à:

POS *(<numéro de mot> , 0 , WTAILLE * LARGEUR)*

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

La notation:

STEP *(<pos>)*

est sémantiquement équivalente à:

STEP *(<pos> , STAILLE)*

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

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

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

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

- Le nom **de champ** est **repérable** si l'implantation de champ est **NOPACK** .

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

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

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

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

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

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 *champ à choisir*, ni tous deux suivre **ELSE** .

Faisabilité:

Le langage n'impose pas de conditions de faisabilité, sauf celle qui peut se déduire de la règle disant que la repérabilité d'un sous-locus de tout locus (**repérable** ou non) est déterminée seulement par l'implantation (de champ ou d'élément), ce qui est une propriété du mode du locus. Ceci restreint l'implantation de composants qui ont eux-mêmes des composants **repérables**.

exemples:

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

3.12 MODES DYNAMIQUES

3.12.1 Généralités

Un mode dynamique est un mode dont certaines propriétés ne sont connues qu'à l'exécution. Les modes dynamiques sont toujours des modes paramétrés avec un ou plusieurs paramètres connus à l'exécution. 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.

3.12.2 Modes chaîne dynamiques

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

sémantique: Un mode chaîne dynamique est un mode chaîne paramétré de longueur inconnue statiquement. La **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).

3.12.3 Modes rangée dynamiques

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

sémantique: Un mode rangée dynamique est un mode rangée paramétré de **borne supérieure** inconnue statiquement. 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 (expression discrète) - NUM (borne inférieure) + 1

  où *borne inférieure* est la **borne inférieure** du *nom de mode rangée originel.*

3.12.4 Modes structure paramétrés dynamiques

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

sémantique: Un mode structure paramétré dynamique est un mode structure paramétré aux paramètres inconnus statiquement. 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 LOCUS ET LEURS ACCÈS

4.1 DÉCLARATIONS

4.1.1 Généralités

syntaxe:

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

<déclaration> ::= *(2)*
<déclaration de locus> *(2.1)*
| *<déclaration de loc-identité>* *(2.2)*
| *<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)*
11.36 *starting_square* **LOC** *:= b(m.lin_1)(m.col_1)* *(2.2)*

4.1.2 Déclarations de locus

syntaxe:

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

<initialisation> ::= *(2)*
<initialisation domaniale> *(2.1)*
| *<initialisation viagère>* *(2.2)*

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

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

sémantique: Une déclaration de locus crée autant de locus qu'on spécifie de définitions apparaissant dans la *liste de définitions.*

Pour une *initialisation domaniale*, la *valeur* est évaluée chaque fois qu'on entre dans le domaine dans lequel la déclaration est placée (voir section 8.2) 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 paramétrage 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 8.2 et 8.9).

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

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

- propriété paramétré avec marqueur: le(s) sous-locus de champ avec **marqueurs** créés sont initialisés avec la valeur de paramètre correspondante.

- propriété de non-valeur:
  - l'événement créé et/ou le(s) sous-locus tampon sont initialisés comme étant "vides", c'est-à-dire qu'aucun processus retard ne s'attache à l'événement ou au tampon et qu'il n'y a pas de messages dans le tampon.
  - le(s) sous-locus d'association créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne contiennent pas d'association.
  - le(s) sous-locus d'accès créés sont initialisés comme étant "vides", c'est-à-dire qu'ils ne sont pas reliés à une association.

La sémantique de **STATIC** et de *filet* sera trouvée, respectivement à la section 8.9 et au chapitre 11.

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

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

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

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:

5.7 *k2, x, w, t, s, r BOOL* *(1.1)*
6.9 *:= julian_day_number* *(3.1)*
8.4 **INIT** *:= ['A':'Z']* *(4.1)*

4.1.3 Déclarations de loc-identité

syntaxe:

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

sémantique: Une déclaration de loc-identité crée autant de noms d'accès au locus spécifié qu'il y a de noms spécifiés dans la *liste de représentations textuelles* de noms simples. Le mode du locus ne peut être dynamique que si **DYNAMIC** est spécifié.

Si le *locus* est évalué dynamiquement, cette évaluation se fait chaque fois que le domaine, dans lequel la déclaration de loc-identité est placée, est entamé. Dans ce cas, un nom déclaré dénote un locus **indéfini** avant la première évaluation durant la durée de vie de l'accès dénoté par le nom déclaré (voir sections 8.2 et 8.9).

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

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

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

exemples:

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

4.1.4 Déclarations de locus avec base

syntaxe:

<déclaration de locus avec base> ::= (1)
<liste de définitions> *<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* ;

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

propriétés statiques: Une définition apparaissant dans une *déclaration de locus avec base* avec *nom de locus repère lié ou libre* définit un nom **basé**. 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**.

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:

19.12 *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 Généralités

syntaxe:

<locus> ::= (1)
<nom d'accès> (1.1)
| *<repère lié dérepéré>* (1.2)
| *<repère libre dérepéré>* (1.3)
| *<rangée dérepéré>* (1.4)
| *<élément de chaîne>* (1.5)
| *<tranche de chaîne>* (1.6)
| *<élément de rangée>* (1.7)
| *<tranche de rangée>* (1.8)
| *<champ de structure>* (1.9)
| *<appel de procédure rendant locus>* (1.10)
| *<appel d'opération prédéfinie rendant locus>* (1.11)
| *<conversion de locus>* (1.12)

sémantique: Un locus est un objet qui peut contenir des valeurs. Les locus doivent être accédés pour y placer ou en obtenir une valeur.

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

- Il peut être **statique** ou non (voir section 8.9).

- Il peut être **intrarégional** ou **extrarégional** (voir section 9.2.2).

- Il peut être **repérable** ou non. La définition du langage exige que certains locus soient **repérables**, comme indiqué dans les sections appropriées. Une implémentation est d'étendre la repérabilité à d'autres locus (voir chapitre 12).

- Il a un **mode**, tel que défini dans les sections appropriées. Ce mode est soit statique soit dynamique.

conditions dynamiques: 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'accès

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

Quand on accède à un locus via un *nom basé*, les mêmes conditions dynamiques ne doivent être remplies que si on dérepérait le *nom de locus repère lié ou libre* mentionné dans la *déclaration de locus avec base* (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) champs(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 associées indique que le champ n'existe pas.

exemples:

4.12 *a* *(1.1)*
11.39 *starting* *(1.2)*
19.17 *x* *(1.3)*
15.35 *each* *(1.4)*
5.10 *c1* *(1.5)*

4.2.3 Repères liés dérepérés

syntaxe:

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

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

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

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

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

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

exemples:

10.54 *p ->* *(1.1)*

4.2.4 Repères libres dérepérés

syntaxe:

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

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

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

conditions statiques: La *valeur primitive repère libre* doit être **forte**.

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

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

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

4.2.5 Rangées dérepérées

syntaxe:

<rangée dérepérée> ::= *(1)*
<valeur primitive rangée > -> *(1.1)*

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

propriétés statiques: Le mode dynamique attaché à une *rangée dérepérée* est construit comme suit:
& nom de mode originel (<paramètre>{ ,<*paramètre*> } *)
où le *nom de mode originel* est un nom virtuel de **synmode synonyme** du mode **originel repéré** du mode de la *valeur primitive rangée* (**forte**) et où les paramètres sont, selon le mode **originel repéré**:

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

Une *rangée dérepérée* est **repérable**.

conditions statiques: La *valeur primitive rangée* doit être **forte**.

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

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

exemples:
8.10 input -> *(1.1)*

4.2.6 Eléments de chaîne

syntaxe:
<*élément de chaîne*> ::= *(1)*
<*locus chaîne* > (<*élément de début*>) *(1.1)*

syntaxe dérivée: Un *élément de chaîne* est une syntaxe dérivée pour une *tranche de chaîne* de longueur 1 (voir section 4.2.7). Par exemple:
< *locus chaîne* > (<*élément de début*>)
est dérivé de:
< *locus chaîne* > (<*élément de début*> **UP** *1*)

exemples:
18.16 string ->(i) *(1.1)*

4.2.7 Tranches de chaîne

syntaxe:
<*tranche de chaîne*> ::= *(1)*
<*locus chaîne* > (<*élément de gauche*> : <*élément de droite*>) *(1.1)*
| <*locus chaîne* > (<*élément de début*> **UP** <*taille de chaîne*>) *(1.2)*

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

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

<*élément de début*> ::= *(4)*
<*expression entière* > *(4.1)*

<*taille de tranche*> ::= *(5)*
<*expression entière* > *(5.1)*

sémantique: Une tranche de chaîne donne un locus de chaîne (éventuellement dynamique) qui est la partie du locus de chaîne spécifié indiqué par *l'élément de gauche* et *l'élément de droite* ou par *l'élément de*

début et *la taille de la tranche*. La longueur (éventuellement dynamique) de la tranche de la chaîne est déterminée à partir des expressions spécifiées.

propriétés statiques: Le mode (éventuellement dynamique) attaché à une *tranche de chaîne* est un mode chaîne **paramétré** formé comme suit:
&nom (taille de tranche)
où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) du *locus de chaîne* et dans lequel la *taille de la chaîne* est soit

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

soit

NUM (taille de tranche).

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

conditions statiques: Si *l'élément de gauche* et *l'élément de droite* sont **littéraux** ou que la *taille de tranche* est **littérale**, ils doivent donner des valeurs entières telles que les relations suivantes soient valables:

$0 \leq$ *NUM (élément de gauche)* $\leq$ *NUM (élément de droite)* $\leq L - 1$

$1 \leq$ *NUM (taille de tranche)* $\leq L$

où L est la **longueur de chaîne** du *locus de chaîne*. Si le mode du *locus de chaîne* est dynamique, ces relations ne peuvent être vérifiées qu'à l'exécution; voir ci-dessous.

conditions dynamiques: L'exception *RANGEFAIL* est causée si l'une quelconque des relations ci-dessus n'est pas vérifiée dans le cas d'un *locus chaîne* de mode dynamique, ou si l'une quelconque des relations ci-après ne se vérifie pas:

$0 \leq$ *NUM (élément de gauche)* $\leq$ *NUM (élément de droite)* $\leq L - 1$

$0 \leq$ *NUM (élément de début)* $<$ *NUM (élément de début)* $+$ *NUM (taille de tranche)* $\leq L$

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

exemples:

18.26 *blanks –> (count : 9)* *(1.1)*
18.23 *string –>(scanstart* **UP** *10)* *(1.2)*

4.2.8 Eléments de rangée

syntaxe:

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

<liste d'expressions> ::= *(2)*
<expression> { , *<expression>*} * *(2.1)*

syntaxe dérivée: La notation: *(<liste d'expressions>)* est une syntaxe dérivée pour:
(<expression>) { *(<expression>)*} *
avec autant d'expressions entre parenthèses qu'il y a d'expressions dans la *liste d'expressions*. Ainsi, un *élément de rangée* en syntaxe stricte n'a qu'une seule expression (d'indice).

sémantique: Un élément de rangée donne un (sous-)locus qui est un élément du locus rangée spécifié.

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

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

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

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

$$L \leq expression \leq U$$

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

exemples:

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

4.2.9 Tranches de rangée

syntaxe:

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

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

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

<*premier élément*> ::= *(4)*
<*expression*> *(4.1)*

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

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

&nom (indice supérieur)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) du *locus* rangée et *l'indice supérieur* est soit une expression dont la classe est **compatible** avec les classes de *l'élément inférieur* et de *l'élément supérieur* et donne une valeur telle que:

NUM (indice supérieur) = NUM (L) + NUM (élément supérieur) − NUM (élément inférieur)

soit une expression dont la classe est **compatible** avec la classe du *premier élément* et donne une valeur telle que:

NUM (indice supérieur) = NUM (L) + NUM (taille de tranche) − 1

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

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

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

conditions statiques: Les classes de *l'élément inférieur* et de *l'élément supérieur* ou la classe du *premier élément* doivent être **compatibles** avec le mode **indice** du *locus* rangée.

Si *l'élément inférieur* et *l'élément supérieur* sont tous deux **littéraux**, ou si la *taille de tranche* est **littérale**, ils doivent donner des valeurs telles que les relations suivantes se vérifient:

$L \leq$ *élément inférieur* $\leq$ *élément supérieur* $\leq U$

$1 \leq$ *NUM (taille de tranche)* $\leq$ *NUM (U) – NUM (L) + 1*

où L et U sont respectivement la **borne inférieure** et la **borne supérieure** du mode du *locus* rangée. Si le mode du *locus* rangée est dynamique, ces relations ne peuvent se vérifier que lors d'un passage; voir ci-dessous.

conditions dynamiques: L'exception *RANGEFAIL* est causée si l'une quelconque des relations ci-dessus ne se vérifie pas pour un *locus* rangée de mode dynamique ou si l'une quelconque des relations ci-après ne se vérifie pas:

$L \leq$ *élément inférieur* $\leq$ *élément supérieur* $\leq U$

NUM (L) $\leq$ *NUM (premier élément)* $\leq$ *NUM (premier élément) +*
NUM (taille de tranche) – 1 $\leq$ *NUM (U)*

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

exemples:

17.27 res (0 : count – 1) *(1.1)*

4.2.10 Champs de structure

syntaxe:

<*champ de structure*> ::= *(1)*
<*locus structure*> . <*nom de champ*> *(1.1)*

sémantique: Un champ de structure donne un (sous-)locus qui est un champ du locus structure spécifié. Si le *locus* structure a un mode **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** si le *nom* de champ est **repérable**. Un *champ de structure* est **repérable** si **l'implantation de champ** du *nom* de champ est **NOPACK** .

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

conditions dynamiques: 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:

10.57 last ->.info *(1.1)*

4.2.11 Appels de procédure rendant locus

syntaxe:

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

sémantique: 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* si **DYNAMIC** n'y est pas spécifié; sinon, c'en est une version paramétrée dynamiquement qui a les mêmes paramètres que le mode du locus rendu.

L'*appel de procédure rendant locus* est **repérable** si **NONREF** n'est pas spécifié dans la **spec de résultat** de *l'appel de procédure rendant locus*

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

4.2.12 Appels d'opération prédéfinie rendant locus

syntaxe:

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

<appel d'opération prédéfinie par CHILL rendant locus> ::= *(2)*
<io CHILL appel d'opération prédéfinie par io CHILL rendant locus> *(2.1)*

sémantique: Un locus est donné comme résultat d'un appel d'opération prédéfinie par l'implémentation rendant locus ou d'un appel d'opération prédéfinie par CHILL rendant locus. Pour l'appel d'opération prédéfinie par io CHILL rendant locus, voir section 7.4.

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* ou l'*appel d'opération prédéfinie par CHILL rendant locus.*

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

4.2.13 Conversions de locus

syntaxe:

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

sémantique: Une conversion de locus 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 spécifié.

La sémantique dynamique précise d'une conversion de locus est définie par l'implémentation.

propriétés statiques: Le mode de la *conversion de locus* est le *nom* de *mode*.

Une *conversion de locus* est **repérable**.

conditions statiques: Le *locus de mode statique* doit être **repérable**.

La relation suivante doit se vérifier:

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

5 VALEURS ET LEURS OPÉRATIONS

5.1 DÉFINITIONS DE SYNONYMES

syntaxe:

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

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

syntaxe dérivée: Une *définition de synonyme*, où la *liste de définitions* comporte plus d'une définition, est dérivée de plusieurs occurrences de *définition de synonyme*, une pour chaque définition, avec la même *valeur constante* et, s'il est présent, le même *mode*. Par exemple: **SYN** *i , j = 3;* est dérivé de: **SYN** *i = 3, j = 3;*

sémantique: Une définition de synonyme définit un nom comme étant une dénotation pour la valeur constante spécifiée.

propriétés statiques: Une *définition* définie dans une *définition de synonyme* est un nom **de synonyme**.

La classe du nom **de synonyme** est, si un *mode* est spécifié, la M-classe par valeur, où M est le *mode*, sinon la classe de la *valeur constante*.

Un nom **de synonyme** est **indéfini** si et seulement si la *valeur constante* est une *valeur indéfinie* (voir 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:

1.17 **SYN** *neutral_for_add = 0,*
neutral_for_mult = 1; *(1.1)*
2.18 *neutral_for_add fraction = [0,1]* *(2.1)*

5.2 VALEUR PRIMITIVE

5.2.1 Généralités

syntaxe:

<valeur primitive> ::= *(1)*
<contenu de locus> *(1.1)*
| *<nom de valeur>* *(1.2)*
| *<littéral>* *(1.3)*
| *<multiplet>* *(1.4)*
| *<valeur élément de chaîne>* *(1.5)*
| *<valeur tranche de chaîne>* *(1.6)*
| *<valeur élément de rangée>* *(1.7)*
| *<valeur tranche de rangée>* *(1.8)*
| *<valeur champ de structure>* *(1.9)*
| *<conversion d'expression>* *(1.10)*
| *<appel de procédure rendant valeur>* *(1.11)*

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

sémantique: Une valeur primitive est le constituant de base d'une expression. Certaines valeurs primitives ont une classe dynamique, c.-à-d. une classe basée sur un mode dynamique. Pour ces valeurs primitives, les vérifications de compatibilité ne peuvent 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 **constante** si et seulement si c'est un *nom de valeur*, un *littéral*, un *multiplet*, un *locus repéré*, une *conversion d'expression* ou un *appel d'opération prédéfinie rendant valeur* **constant**.

Une *valeur primitive* est **littérale**, si et seulement si c'est un *nom de valeur*, 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 non-valeur**.

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

exemples:

3.7 c2.im (1.1)

5.2.3 Noms de valeur

syntaxe:

<*nom de valeur*> ::= (1)
<*nom de synonyme* > (1.1)
| <*nom d'énumération de valeur* > (1.2)
| <*nom de valeur faire-avec* > (1.3)
| <*nom de valeur reçue* > (1.4)
| <*nom de procédure générale* > (1.5)

sémantique: Un nom de valeur donne une valeur.

Un nom de valeur entre dans une des catégories suivantes:

- un nom **de synonyme**, c.-à-d. un nom défini dans un *énoncé de définition de synonymes*;
- un nom **d'énumération de valeur**, c.-à-d. un nom défini par un *compteur de boucle* dans une *énumération de valeur*;
- un nom **de valeur faire-avec**, c.-à-d. un nom de **champ** introduit comme nom de valeur dans *l'action faire* avec une *partie avec*;
- un nom **de valeur reçue**, c.-à-d. un nom introduit dans une *action recevoir et choisir*;

- un nom **de procédure générale** (voir section 8.4).

propriétés statiques: La classe d'un *nom de valeur* est respectivement la classe du *nom de synonyme*, du *nom d'énumération de valeur*, du *nom de valeur faire-avec*, du *nom de valeur reçue*, ou la classe dérivée de M, où M est le mode du *nom de procédure générale*.

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

Un *nom de valeur* est **constant** si c'est un *nom de synonyme* ou un *nom de procédure générale* indiquant un nom **de procédure** qui s'est attaché à une *définition de procédure* qui n'est pas entourée par un bloc.

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

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

- d'une valeur de mode structure **variable avec marqueurs** et que le(s) 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.

exemples:

10.12	*max*	*(1.1)*
8.8	*i*	*(1.2)*
15.54	*this_counter*	*(1.4)*

5.2.4 Littéraux

5.2.4.1 Généralités

syntaxe:

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

sémantique: Un littéral donne une valeur constante.

propriétés statiques: La classe du *littéral* est respectivement la classe du *littéral d'entier*, *littéral de booléen*, ..., etc. Un *littéral* est **discret** si c'est un *littéral d'entier*, un *littéral de booléen*, un *littéral d'ensemble*, un *littéral de chaîne de caractères* de longueur 1, ou un *littéral de chaîne de bits* de longueur 1.

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

5.2.4.2 Littéraux d'entier

syntaxe:

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

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

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

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

<chiffre octal> ::= (8)
0 | *1* | *2* | *3* | *4* | *5* | *6* | *7* (8.1)

Note: Une implémentation peut offrir des lettres minuscules pour les **qualifications de littéraux** (c.-à-d. *d'*, *b'*, *o'*, *h'*) et pour les lettres d'un *chiffre hexadécimal* (c.-à-d. a, b, c, d, e, f).

sémantique: Un littéral d'entier donne une valeur entière non négative. La notation décimale usuelle (base 10) est offerte, de même que les notations binaire (base 2), octale (base 8), hexadécimale et décimale (base 16). 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 soulignés.

exemples:

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

5.2.4.3 Littéraux de booléen

syntaxe:

<littéral de booléen> ::= (1)
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)*
11.78 *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**.

exemples:

10.43 *NULL* *(1.1)*

5.2.4.6 Littéraux de chaîne de caractères

syntaxe:

<littéral de chaîne de caractères> ::= *(1)*
' { *<caractère non apostrophe>* | *<apostrophe>*}*' *(1.1)*
| *C'* { *<chiffre octal>* *<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**. Une implémentation peut offrir des lettres minuscules pour les **qualifications de littéraux** (c.-à-d. *c'*).

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 de l'alphabet CCITT no. 5 dans un littéral de chaîne de caractères comme production terminale de *<caractère>* (voir Appendice A1). En plus de la représentation imprimable, on peut employer la représentation hexadécimale. Chaque paire de *chiffres octaux* ou *hexadécimaux* dénote cette valeur caractère dont la représentation est la valeur hexadécimale donnée (voir Appendice A1); le caractère souligné (_) n'est pas significatif.

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

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

exemples:

8.19 *'A-B<ZAA9K' '* (1.1)
8.19 *' '* (6.1)

5.2.4.7 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' { <chiffre octal> | _ } *'* (3.1)

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

Note: Une implémentation peut offrir des lettres minuscules pour les **qualifications de littéraux** (c.-à-d. *b'*, *o'*, *h'*).

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 *chiffre octal* après *O'*, soit quatre fois le nombre d'occurrences de *chiffre hexadécimal* après *H'*.

La classe d'un *littéral de chaîne de bits* est la *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)
| *<littéral de chaîne de caractères>* (1.2)
| *<littéral de chaîne de bits>* (1.3)

<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)

Note: Si le *multiplet* est un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits*, la construction syntaxique est ambiguë; elle sera interprétée comme un *multiplet* si et seulement si un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits* se produit dans un contexte où un *multiplet de rangée* sans *nom de mode* est autorisé.

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 10.1.3).

Si un multiplet rangée est **constant** et que le mode des éléments est **compatible** avec la classe dérivée de *CHAR(1)* (dérivée de *BIT(1)*), il est possible d'utiliser un *littéral de chaîne de caractères (de bits)* comme expression abrégée pour le multiplet rangée (par exemple, *(: 'a', 'b', 'c', 'd' :)* peut être écrit *'abcd'*).

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 de champ sont spécifiés dans la liste de noms de champ pour la valeur.

L'ordre d'évaluation des expressions et des valeurs dans un multiplet est indéfini et elles peuvent être vues comme étant évaluées dans un ordre mélangé.

propriétés statiques: La classe d'un *multiplet* est la M-classe par valeur où M est le *nom de mode*, s'il y en a un, sinon M dépend du contexte où le *multiplet* se trouve 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 *paramètre effectif* dans un *appel de procédure* ou dans une *expression de début*, 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;
- si le *multiplet* est la *valeur* dans un appel d'opération prédéfinie *GETSTACK* ou *ALLOCATE*, alors M est le mode dénoté par *argument*.

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 qu'un *multiplet ensembliste*, *multiplet de rangée* ou *multiplet de structure* est spécifié, les règles de compatibilité suivantes doivent être respectées:

a. *multiplet ensembliste*

1. Le mode du *multiplet* doit être un mode ensembliste.

2. La classe de chaque *expression* doit être **compatible** avec le mode **primitif** du mode du *multiplet*.

3. Pour un multiplet ensembliste constant, la valeur donnée par chaque *expression* doit être une des valeurs définies par ce mode **primitif**.

b. *multiplet de rangée*

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

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

3. Dans le cas d'un *multiplet de rangée sans indices*, il faut autant d'occurrences de *valeur* que d'éléments dans le mode rangée du *multiplet*.

4. Dans le cas d'un *multiplet de rangée avec indices*, les conditions de sélection de cas doivent être remplies pour la liste d'occurrences de *liste d'étiquettes de cas* (voir

section 10.1.3). La **classe résultante** de la liste doit être **compatible** avec le mode **d'indice** du mode du *multiplet*.

5. Dans le cas d'un *multiplet de rangée avec indices*, 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 une valeur définie par le mode **d'indice** du *multiplet*.

6. Dans un *multiplet de rangée sans indices*, au moins une occurrence de *valeur* doit être une *expression*.

7. Pour un *multiplet* (de rangée) **constant** où le mode **des éléments** du mode du *multiplet* est un mode discret, chaque *valeur* spécifiée doit donner une valeur définie par ce mode **des éléments**, sauf si c'est une valeur **indéfinie**.

8. Dans les contextes où le *nom de mode* facultatif peut être supprimé (comme spécifié ci-dessus), il est possible d'utiliser un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits*, à condition que:

1. le mode du *multiplet* soit un mode rangée;

2. le mode **des éléments** du mode du *multiplet* soit **compatible** avec la classe de *CHAR(1)* ou de *BIT(1)* si le *multiplet* est un *littéral de chaîne de caractères* ou un *littéral de chaîne de bits*, respectivement;

3. la **longueur de chaîne** du *littéral de chaîne de caractères* ou du *littéral de chaîne de bits* doive rendre la même valeur que le **nombre des éléments** du mode du *multiplet*.

c. *multiplet de structure*

1. Le mode de multiplet doit être un mode structure.

2. Ce mode ne doit pas être un mode structure qui a des **noms de champ** invisibles (voir section 10.2.5).

Dans le cas d'un *multiplet de structure sans noms de champ*:

- Si le mode du *multiplet* n'est ni un mode structure **variable** ni un mode structure **paramétré**, alors:

  3. Il doit y avoir autant d'occurrences de *valeur* qu'il y a de **noms de champ** dans la liste de **noms de champ** du mode du *multiplet*.

  4. La classe de chaque *valeur* doit être **compatible** avec le mode du **nom de champ** correspondant (par position) du mode du *multiplet*.

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

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

  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** ou un mode structure **paramétré 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 *champ à choisir* ou après **ELSE** . Tous les noms de **champ** d'un choix de champs sélectionnés ou définis après **ELSE** , doivent être mentionnés une 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*.

Aucun *multiplet* ne peut comporter deux occurrences de *valeur* telles que l'une soit **extrarégionale** et l'autre **intrarégionale** (voir section 9.2.2).

conditions dynamiques: Les conditions d'affectation de chaque valeur pour ce qui est du mode **primitif**, du mode **des éléments**, ou du mode **de champ** associé, dans le cas respectivement d'un *multiplet ensembliste*, *multiplet de rangée* ou *multiplet de structure* (voir section 6.2) s'appliquent (voir conditions a2, 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.

La valeur donnée par un *multiplet* ne doit pas être **indéfinie**.

exemples:

9.6	*number_list*[ ]	*(1.1)*
9.7	[*2:max*]	*(2.1)*
8.25	[*('A'):3,('B','K','Z'):1,(* **ELSE** *):0*]	*(6.1)*
17.5	[*(*):' '*]	*(6.1)*
12.35	*(: NULL , NULL ,536:)*	*(7.1)*
11.18	[*.status:occupied,.p:*[*white,rook*]]	*(9.1)*

5.2.6 Valeurs élément de chaîne

syntaxe:

<valeur élément de chaîne> ::= *(1)*
<valeur primitive chaîne>(<élément de début>) *(1.1)*

syntaxe dérivée: Une *valeur élément de chaîne* est une syntaxe dérivée pour une *valeur tranche de chaîne* de longueur 1 (voir section 5.2.7), c.-à-d.:
< valeur primitive chaîne> (<élément de début>)
est dérivé de:
<valeur primitive chaîne > (<élément de début> **UP** *1)*

5.2.7 Valeurs tranche de chaîne

syntaxe:

<valeur tranche de chaîne> ::= *(1)*
< valeur primitive chaîne>(<élément de gauche> : <élément de droite>) *(1.1)*
| *< valeur primitive chaîne>(<élément de début>* **UP** *<taille de chaîne>)* *(1.2)*

Note: Si la *valeur primitive chaîne* est un *locus chaîne*, la construction syntaxique est ambiguë et sera interprétée comme une *tranche de chaîne* (voir section 4.2.7).

sémantique: Une valeur tranche de chaîne donne une valeur chaîne (éventuellement dynamique) qui est la partie de la valeur chaîne spécifiée indiquée par *l'élément de gauche* et *l'élément de droite* ou par *l'élément de début* et *la taille de tranche*. La longueur (éventuellement dynamique) de la tranche de chaîne est déterminée à partir des expressions spécifiées.

propriétés statiques: La classe (éventuellement dynamique) d'une *valeur tranche de chaîne* est la valeur de tranche M, où M est un mode chaîne **paramétré** construit comme:
&nom (taille de chaîne)
où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) de *la valeur primitive de chaîne* et où *taille de chaîne* est soit

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

soit

NUM (taille de tranche).

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

conditions statiques: Si *l'élément de gauche* et *l'élément de droite* sont **littéraux**, ou que la *taille de tranche* est **littérale**, alors ils doivent donner des valeurs d'entier telles que la relation suivante se vérifie:

0 ≤ NUM (élément de gauche) ≤ NUM (élément de droite) ≤ L – 1

1 ≤ NUM (taille de tranche) ≤ L

où *L* est la **longueur de chaîne** du mode de la *valeur primitive chaîne*. (Si le mode de la *valeur primitive chaîne* est dynamique, ces relations ne peuvent se vérifier qu'à l'exécution; voir plus loin.)

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

L'exception *RANGEFAIL* est causée si l'une quelconque des relations ci-dessus ne se vérifie pas dans le cas d'une *valeur primitive chaîne* de classe dynamique, ou si l'une quelconque des relations suivantes ne se vérifie pas:

0 ≤ NUM (élément de gauche) ≤ NUM (élément de droite) ≤ L – 1

0 ≤ NUM (élément de début) < NUM (élément de début) +
NUM (taille de tranche) ≤ L

où *L* est la **longueur de chaîne** (éventuellement dynamique) du mode de la *valeur primitive chaîne*.

5.2.8 Valeurs élément de rangée

syntaxe:

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

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

syntaxe dérivée: Voir section 4.2.8.

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 la *valeur primitive rangée*.

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

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

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

L ≤ expression ≤ U

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

5.2.9 Valeurs tranche de rangée

syntaxe:

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

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

sémantique: Une valeur tranche de rangée donne une valeur rangée (éventuellement dynamique) qui est la partie de la valeur rangée spécifiée indiquée par *l'élément inférieur* et *l'élément supérieur*, ou par le

premier élément et la *taille de tranche*. La **borne inférieure** de la valeur tranche de rangée est égale à la **borne inférieure** de la valeur rangée spécifiée; la **borne supérieure** (éventuellement dynamique) est déterminée à partir des expressions spécifiées.

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

&nom (indice supérieur)

où *&nom* est un nom de **synmode** virtuel **synonyme** du mode (éventuellement dynamique) de la *valeur primitive rangée* et *l'indice supérieur* est soit une expression dont la classe est **compatible** avec les classes de *l'élément inférieur* et de *l'élément supérieur* et donne une valeur telle que:

NUM (indice supérieur) = NUM (L) + NUM (élément supérieur) – NUM (élément inférieur)

ou est une expression dont la classe est **compatible** avec la classe du *premier élément* et donne une valeur telle que:

NUM (indice supérieur) = NUM (L) + NUM (taille de tranche) – 1

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

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

conditions statiques: Les classes de *l'élément inférieur* et de *l'élément supérieur* ou la classe du *premier élément* doivent être **compatibles** avec le mode **d'indice** de la *valeur primitive rangée*.

Si *l'élément inférieur* et *l'élément supérieur* sont tous deux **littéraux** ou que la *taille de tranche* est **littérale**, ils doivent donner des valeurs telles que la relation suivante se vérifie:

$L \leq$ *élément inférieur* $\leq$ *élément supérieur* $\leq U$

$1 \leq$ *NUM (taille de tranche)* $\leq$ *NUM (U) – NUM (L) + 1*

où *L* et *U* sont respectivement la **borne inférieure** et la **borne supérieure** du mode de la *valeur primitive rangée*. Si le mode de la *valeur primitive rangée* est dynamique, ces relations ne peuvent se vérifier qu'à l'exécution; voir ci-dessous.

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

L'exception *RANGEFAIL* est causée si l'une quelconque des relations ci-dessus ne se vérifie pas pour une *valeur primitive rangée* qui a une classe dynamique ou si l'une quelconque des relations ci-après ne se vérifie pas:

$L \leq$ *élément inférieur* $\leq$ *élément supérieur* $\leq U$

NUM (L) $\leq$ *NUM (premier élément)* $\leq$ *NUM (premier élément) + NUM (taille de tranche) – 1* $\leq$ *NUM (U)*

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

5.2.10 Valeurs champ de structure

syntaxe:

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

Note: Si la *valeur primitive structure* est un *locus structure*, la construction syntaxique est ambiguë et sera interprétée comme un *champ de structure* (voir section 4.2.10).

sémantique: Une valeur champ de structure donne une valeur qui est un champ de la valeur structure spécifiée. Si la *valeur primitive structure* a un mode structure **variable sans marqueurs** et que le *nom de champ* est un nom de champ **récurrent**, la sémantique est définie par l'implémentation.

propriétés statiques: La classe d'une *valeur champ de structure* est la M-classe par valeur où M est le mode du *nom de champ*.

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

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

L'exception *TAGFAIL* est causée si la *valeur primitive 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:

16.51 (**RECEIVE** *user_buffer).allocator* *(1.1)*

5.2.11 Conversions d'expression

syntaxe:

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

Note: Si *l'expression* est un *locus mode statique*, la construction syntaxique est ambiguë et sera interprétée comme une *conversion de locus* (voir section 4.2.13).

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. Si le mode du nom de mode est un mode discret et que la classe de la valeur donnée par l'expression est discrète, alors, la valeur donnée par la conversion d'expression doit être telle que:

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

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

propriétés statiques: La classe de la *conversion d'expression* est la M-classe par valeur, où M est le *nom de mode*. Une *conversion d'expression* est **constante** si et seulement si *l'expression* est **constante**.

conditions statiques: Le *nom de mode* ne doit pas avoir la **propriété de non-valeur**. Une implémentation peut imposer des conditions statiques supplémentaires.

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

5.2.12 Appels de procédure rendant valeur

syntaxe:

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

sémantique: Un appel de procédure rendant valeur donne la valeur retournée par la procédure.

propriétés statiques: La classe d'un *appel de procédure rendant valeur* est la M-classe par valeur où M est le mode de la **spec de résultat** de *l'appel de procédure rendant valeur*.

conditions dynamiques: L'*appel de procédure rendant valeur* ne peut donner une valeur **indéfinie** (voir sections 5.3.1 et 6.8).

exemples:

6.50 *julian_day_number([10,dec,1979])* *(1.1)*
11.63 *ok_bishop(b,m)* *(1.1)*

5.2.13 Appels d'opération prédéfinie rendant valeur

syntaxe:

<appel d'opération prédéfinie rendant valeur> ::= *(1)*
< appel d'opération prédéfinie 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)*
| *SUCC (< expression discrète>)* *(2.3)*
| *ABS (< expression entière>)* *(2.4)*
| *CARD (< expression ensembliste>)* *(2.5)*
| *MAX (< expression ensembliste>)* *(2.6)*
| *MIN (< expression ensembliste>)* *(2.7)*
| *SIZE ({ < nom de mode> | < locus de mode statique>})* *(2.8)*
| *UPPER (<argument pour upper lower >)* *(2.9)*
| *LOWER (<argument pour upper lower >)* *(2.10)*
| *GETSTACK (<argument pour getstack > [, <valeur>])* *(2.11)*
| *ALLOCATE (<argument pour allocate > [, <valeur>])* *(2.12)*
| *<appel d'opération prédéfinie valeur io CHILL >* *(2.13)*

<argument pour getstack> ::= *(3)*
<argument> *(3.1)*

<argument pour allocate> ::= *(4)*
<argument> *(4.1)*

<argument> ::= *(5)*
< nom de mode> *(5.1)*
| *< nom de mode rangée>(<expression>)* *(5.2)*
| *< nom de mode chaîne>(< expression d'entier >)* *(5.3)*
| *<nom de mode structure récurrente >(<liste d'expressions >)* *(5.4)*

<argument pour upper lower> ::= *(6)*
< locus rangée> *(6.1)*
| *<valeur primitive rangée >* *(6.2)*
| *< nom de mode rangée>* *(6.3)*
| *< locus chaîne>* *(6.4)*
| *<valeur primitive chaîne >* *(6.5)*
| *< nom de mode chaîne>* *(6.6)*
| *< locus discret>* *(6.7)*
| *< expression discrète>* *(6.8)*
| *< nom de mode discret>* *(6.9)*

Note: Si *l'argument pour upper lower* est un *locus* (*rangée*, *chaîne*, *discret*), l'ambiguïté syntaxique est résolue comme suit: on interprète *argument pour upper lower* comme un *locus* plutôt que comme une *expression* ou une *valeur primitive*.

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. Les opérations prédéfinies par CHILL rendant valeur qui se rapportent à l'entrée-sortie sont définies dans le chapitre 7.

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* donnent respectivement la valeur discrète précédant ou suivant immédiatement. 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).

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

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 adressables 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 et *LOWER* sont définis dans les éventualités ci-après (éventuellement dynamiques):

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

respectivement.

GETSTACK et *ALLOCATE* créent un locus du mode spécifié et donnent une valeur repère pour le locus créé. *GETSTACK* crée ce locus sur la pile (voir section 8.9). Si *l'argument* est un *nom de mode*, un locus de mode statique de ce mode est créé et une valeur repère est donnée. Sinon, un locus de mode dynamique est créé, dont le mode est un mode **paramétré** avec paramètres non **littéraux** comme spécifié dans *l'argument*, et une valeur rangée qui se rapporte au locus est donnée. Le locus créé est initialisé par la valeur de *valeur*, si elle existe; sinon, par la valeur **non définie** (voir section 4.1.2).

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

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

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

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

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

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

La classe d'un appel d'opération prédéfinie *GETSTACK* ou *ALLOCATE* est la M-classe par repère, où M est le mode de *l'argument*, M soit le *nom de mode* soit un mode **paramétré** 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 d'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 d'opération prédéfinie *MAX* ou *MIN* est **constant**, il ne peut pas donner la valeur ensembliste vide.

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

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

Les conditions de compatibilité suivantes doivent être remplies pour un *argument de 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*.
- Le *mode structure variable* doit être **paramétrable** et il doit y avoir autant d'expressions dans la *liste d'expressions* qu'il y a de classes dans la liste de classes du *nom de mode structure variable* et la classe de chaque expression doit être **compatible** avec la classe correspondante de la liste de classes du *nom de mode structure variable*.

La classe de la *valeur*, si elle existe dans l'appel d'opération prédéfinie *GETSTACK* et *ALLOCATE*, doit être **compatible** avec le mode de *l'argument*; cette vérification est dynamique dans le cas où le mode de *l'argument* est un mode dynamique.

propriétés dynamiques: Une valeur repère est une valeur repère **attribuée** si et seulement si elle est renvoyée par un appel d'opération prédéfinie *ALLOCATE* .

conditions dynamiques: *PRED* et *SUCC* causent l'exception *OVERFLOW* s'ils sont appliqués à la plus petite ou à la plus grande valeur discrète définie par le mode **racine** de la classe de leur argument.

NUM et *CARD* causent l'exception *OVERFLOW* si la valeur résultante est en dehors de l'ensemble de valeurs définies par *INT* .

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

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.

GETSTACK et *ALLOCATE* causent l'exception *RANGEFAIL* si, dans *l'argument*:

- 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 de la chaîne** 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.

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

ALLOCATE cause l'exception *ALLOCATEFAIL* si les requêtes de mémoire ne peuvent être satisfaites.

Pour *GETSTACK* et *ALLOCATE* , les conditions d'attribution de la valeur données par *valeur* par rapport au mode de *l'argument* sont applicables.

exemples:

9.12	*MIN (sieve)*	*(2.10)*
11.47	*PRED (col_1)*	*(2.2)*
11.47	*SUCC (col_1)*	*(2.4)*

5.2.14 Expressions démarrer

syntaxe:

<expression démarrer> ::= *(1)*

START *<nom de processus>* ([*<liste de paramètres effectifs>*]) *(1.1)*

sémantique: L'évaluation de l'expression démarrer crée et active un nouveau processus dont la définition est identifiée par le nom de processus (voir chapitre 9). 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: 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.

exemples:

15.35 **START** *counter()* *(1.1)*

5.2.15 Opérateur nullaire

syntaxe:

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

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

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

5.2.16 Expressions parenthésées

syntaxe:

<expression parenthésée> ::= *(1)*
(<expression>) *(1.1)*

sémantique: Une expression parenthésée donne la valeur rendue par l'évaluation de l'expression.

propriétés statiques: La classe de *l'expression parenthésée* est la classe de *l'expression.*

Une *expression parenthésée* est **constante** (**littérale**) si et seulement si *l'expression* est **constante** (**littérale**).

exemples:

5.10 (a1 OR b1) *(1.1)*

5.3 VALEURS ET EXPRESSIONS

5.3.1 Généralités

syntaxe:

<valeur> ::= *(1)*
<expression> *(1.1)*
| *<valeur indéfinie>* *(1.2)*

<valeur indéfinie> ::= *(2)*
* *(2.1)*
| *< nom de synonyme indéfini>* *(2.2)*

sémantique: Une valeur est 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**.

propriétés 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 tranche de chaîne*, une *valeur élément de 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 le contexte exige une expression **constante** ou **littérale**, on admet que l'évaluation est faite avant l'achèvement et ne peut causer une exception. Une implémentation définira les rangées de valeurs autorisées pour des expressions littérales et constantes et peut rejeter un programme si cette évaluation préalable donne une valeur en dehors des bornes définies par l'implémentation.

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 *OR* exclusif;
- 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 bits 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:

10.31 *i<min* *(1.1)*
10.31 *i<min OR i>max* *(1.2)*

5.3.3 Opérande-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:

5.10 (a1 OR b1) *(1.1)*
5.10 NOT k2 AND (a1 OR b1) *(1.2)*

5.3.4 Opérande-2

syntaxe:

<opérande-2> ::= *(1)*
<opérande-3> *(1.1)*
| *<sous-opérande-2> <opérateur-3> <opérande-3>* *(1.2)*

<sous-opérande-2> ::= *(2)*
<opérande-2> *(2.1)*

<opérateur-3> ::= *(3)*
<opérateur relationnel> *(3.1)*
| *<opérateur d'appartenance>* *(3.2)*
| *<opérateur d'inclusion ensembliste>* *(3.3)*

<opérateur relationnel> ::= *(4)*
= | /= | > | >= | < | <= *(4.1)*

<opérateur d'appartenance> ::= *(5)*
IN *(5.1)*

<opérateur d'inclusion ensembliste> ::= *(6)*
<= | >= | < | > *(6.1)*

sémantique: L'opérateur d'égalité (=) et les opérateurs d'inégalité (/=) sont définis entre toutes les valeurs d'un mode donné. Les autres opérateurs relationnels (inférieur à: <, inférieur ou égal à: <=, supérieur à: >, supérieur ou égal à: >=) sont définis entre les valeurs d'un mode donné discret 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. Un opérateur d'inclusion ensembliste donne une valeur booléenne comme résultat.

propriétés statiques: Si un *opérande-2* est un *opérande-3*, la classe de *l'opérande-2* est la classe de *l'opérande-3*. Si un *opérateur-3* est spécifié, la classe de *l'opérande-2* est la *BOOL* -classe par dérivation.

Un *opérande-2* est **constant** (**littéral**) si et seulement s'il est soit un *opérande-3* qui est **constant** (**littéral**), soit construit d'un *sous-opérande-2* et d'un *opérande-3* qui sont tous deux **constants** (**littéraux**).

conditions statiques: Si un *opérateur-3* est spécifié, les conditions de compatibilité suivantes entre la classe de *sous-opérande-2* et celle de *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:
10.50 NULL *(1.1)*
10.50 last= NULL *(1.2)*

5.3.5 Opérande-3

syntaxe:

<opérande-3> ::= *(1)*
<opérande-4> *(1.1)*
| *<sous-opérande-3> <opérateur-4> <opérande-4>* *(1.2)*

<sous-opérande-3> ::= *(2)*
<opérande-3> *(2.1)*

<opérateur-4> ::= *(3)*
<opérateur arithmétique additif > *(3.1)*
| *<opérateur de concaténation de chaîne >* *(3.2)*
| *<opérateur de différence ensembliste >* *(3.3)*

<opérateur arithmétique additif> ::= *(4)*
\+ | - *(4.1)*

<opérateur de concaténation de chaîne> ::= *(5)*
// *(5.1)*

<opérateur de différence ensembliste> ::= *(6)*
– *(6.1)*

sémantique: Si l'opérateur-4 est un opérateur arithmétique additif, les deux opérandes donnent des valeurs entières et la valeur entière résultante est la somme (+) ou différence (–) des deux valeurs.

Si l'opérateur-4 est un opérateur de concaténation de chaîne, les deux opérandes donnent soit des valeurs chaîne de bits soit des valeurs chaîne de caractères; la valeur résultante consiste en la concaténation de ces valeurs.

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 de chaîne** des modes **racine** des deux classes,
  - sinon, la classe est la *&nom(n)*-classe par valeur, où *&nom* est un nom de **synmode** virtuel **synonyme** du mode de l'un des opérandes **forts** et où *n* dénote la somme des **longueurs de chaîne** des modes **racine** des deux classes.

  (Cette classe est dynamique si l'un ou les deux opérandes ont une classe dynamique.)
- Si l'*opérateur-4* est un *opérateur arithmétique additif* ou *opérateur de différence ensembliste*, la classe de l'*opérande-3* est la **classe résultante** des classes de *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**) et que l'*opérateur-4* est soit l'*opérateur arithmétique d'addition* soit l'*opérateur de différence ensembliste*.

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* est un *opérateur de concaténation de chaîne*, les modes **racine** des classes des deux opérandes doivent tous deux être **compatibles** avec le mode chaîne **de bits** ou avec le mode chaîne de **caractères** et, si les deux classes sont des classes par valeur, les modes **racine** doivent avoir la même **nouveauté**;
- si l'*opérateur-4* est un *opérateur de différence ensembliste*, les classes des deux opérandes doivent être **compatibles** et toutes deux doivent avoir un mode **racine** ensembliste.

conditions dynamiques: Dans le cas d'un *opérande-3* qui n'est pas **constant**, une exception *OVERFLOW* est causée si une addition (+) ou une soustraction (–) donne une valeur qui n'est pas comprise entre les bornes spécifiées par le mode **racine** de la classe de l'*opérande-3*.

exemples:

1.6 j *(1.2)*
1.6 i+j *(1.2)*

5.3.6 Opérande-4

syntaxe:

<opérande-4> ::= *(1)*
<opérande-5> *(1.1)*
| *<sous-opérande-4> <opérateur arithmétique multiplicatif> <opérande-5>* *(1.2)*

<sous-opérande-4> ::= *(2)*
<opérande-4> *(2.1)*

<opérateur arithmétique multiplicatif> ::= *(3)*
* | / | *MOD* | *REM* *(3.1)*

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

L'opération modulo est définie de telle manière que *I MOD J* donne l'entier unique K, $0 \leq K < J$ tel qu'il existe un entier N tel que $I = N * J + K$; J doit être supérieur à *0*.

L'opération quotient se définit de telle sorte que toutes les relations
ABS (X/Y) = ABS (X) / ABS (Y) et
signe (X/Y) = signe (X) / signe (Y) et
*ABS (X) - (ABS (X) / ABS (Y)) * ABS (Y) = ABS (X) MOD ABS (Y)*
donnent *TRUE* pour toutes les valeurs entières de X et Y, où le *signe (X) = -1* si $X < 0$, sinon le *signe (X) = 1*.

L'opération de reste est définie de telle manière que *X REM Y = X - (X/Y) * Y* donne *TRUE* pour toutes les valeurs entières de X et Y.

propriétés statiques: Si l'*opérande-4* est un *opérande-5*, la classe de l'*opérande-4* est la classe de l'*opérande-5*; sinon, la classe de l'*opérande-4* est la **classe résultante** des classes 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 de *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 l'une des valeurs définies 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 Opérande-5

syntaxe:

<opérande-5> ::= *(1)*
[*<opérateur unaire>*] *<opérande-6>* *(1.1)*

<opérateur unaire> ::= *(2)*
- | *NOT* *(2.1)*
| *<opérateur de répétition de chaîne>* *(2.2)*

<opérateur de répétition de chaîne> ::= *(3)*
(< expression littérale entière>) *(3.1)*

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

Si l'opérateur unaire est *NOT* , l'opérande-6 donne soit une valeur booléenne soit une valeur chaîne de bits, soit une valeur ensembliste. Dans les deux premiers cas, la négation logique de la valeur booléenne ou chaîne de bits est donnée, dans le dernier cas, la valeur ensembliste complémentaire, c.-à-d. que l'ensemble de ces valeurs primitives qui ne sont pas dans la valeur ensembliste opérande est donné.

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 de chaîne** du littéral chaîne.

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

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

conditions statiques: Si *l'opérateur unaire* est –, la classe de *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'opération changer-de-signe (–) donne une valeur qui n'est pas dans l'une des valeurs définies par le mode racine de la classe de *l'opérande-5*.

exemples:

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

5.3.8 Opérande-6

syntaxe:

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

<locus repéré> ::= *(2)*
-> <locus> *(2.1)*
| **ADDR** *(<locus>)* *(2.2)*

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

syntaxe dérivée: **ADDR** *(<locus>)* est la syntaxe dérivée de *-> <locus>*.

sémantique: Un opérande-6 est soit un locus repéré, soit une expression recevoir, soit une valeur primitive (voir section 5.2.1).

Un locus repéré donne un repère au locus spécifié.

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

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

conditions statiques: Le *locus* doit être **repérable**.

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

exemples:
8.24 *-> c* *(2.1)*
16.51 **RECEIVE** *user_buffer* *(3.1)*

6 ACTIONS

6.1 GÉNÉRALITÉS

syntaxe:

<énoncé d'action> ::= (1)
[<définition > :] <action> [<filet>] [<représentation textuelle de nom simple>]; (1.1)
| <module> (1.2)
| <module de spec> (1.3)

<action> ::= (2)
<action parenthésée> (2.1)
| <action d'affectation> (2.2)
| <action appeler> (2.3)
| <action sortir> (2.4)
| <action revenir> (2.5)
| <action résulter> (2.6)
| <action aller> (2.7)
| <action affirmer> (2.8)
| <action vide> (2.9)
| <action démarrer> (2.10)
| <action arrêter> (2.11)
| <action mettre en attente> (2.12)
| <action continuer> (2.13)
| <action envoyer> (2.14)
| <action causer> (2.15)

<action parenthésée> ::= (3)
<action conditionnelle> (3.1)
| <action de cas> (3.2)
| <action faire> (3.3)
| <bloc début-fin> (3.4)
| <action mettre en attente et choisir> (3.5)
| <action recevoir et choisir> (3.6)

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

propriétés statiques: Une *définition* apparaissant dans un *énoncé d'action* définit un nom **d'étiquette**.

conditions statiques: La *représentation textuelle de nom simple* ne peut être donnée qu'après une *action* qui est une *action parenthésée* ou si un *filet* est spécifié et seulement si une *définition* est spécifiée. La *représentation textuelle de nom simple* doit être la même représentation textuelle que la *définition*.

6.2 ACTION D'AFFECTATION

syntaxe:

<action d'affectation> ::= (1)
<action d'affectation simple> (1.1)
| <action d'affectation multiple> (1.2)

<action d'affectation simple> ::= (2)
<locus> { <symbole d'affectation> | <opérateur affectant>} <valeur> (2.1)

<action d'affectation multiple> ::= (3)
<locus> { ,<locus>}$^{+}$ <symbole d'affectation> <valeur> (3.1)

<opérateur affectant> ::= *(4)*

<opérateur binaire fermé> <symbole d'affectation> *(4.1)*

<opérateur binaire fermé> ::= *(5)*

OR | XOR

| AND *(5.1)*

| <opérateur de différence ensembliste> *(5.2)*

| <opérateur arithmétique additif> *(5.3)*

| <opérateur arithmétique multiplicatif> *(5.4)*

<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, la sémantique des champs récurrents qui en dépendent est définie par l'implémentation.

conditions statiques: Les modes de chaque occurrence de *locus* doivent être **équivalents** et ils ne peuvent avoir ni la **propriété de protection**, ni la **propriété de non-valeur**. Chaque mode doit être **compatible** avec la classe de la *valeur*. Les vérifications sont dynamiques dans les cas où il s'agit de locus de mode dynamique et/ou d'une valeur de classe dynamique.

La *valeur* doit être **régionalement sûre** pour chaque *locus* (voir section 9.2.2).

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* n'est ni l'une des valeurs définies par le mode intervalle ni la *valeur indéfinie*.

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:

4.12	*a := b+c*	*(1.1)*
10.25	*stackindex- := 1*	*(2.1)*
19.19	*x.prex, x.next := NULL*	*(3.1)*
10.25	*- :=*	*(4.1)*

6.3 ACTION CONDITIONNELLE

syntaxe:

<action conditionnelle> ::= *(1)*
IF *< expression booléenne> <clause alors> [<clause sinon>]* **FI** *(1.1)*

<clause alors> ::= *(2)*
THEN *<liste d'énoncés d'action>* *(2.1)*

<clause sinon> ::= *(3)*
ELSE *<liste d'énoncés d'action>* *(3.1)*
| **ELSIF** *< expression booléenne> <clause alors> [<clause sinon>]* *(3.2)*

syntaxe dérivée: La notation:
ELSIF *< expression booléenne> <clause alors> [<clause sinon>]*
est une syntaxe dérivée pour:
ELSE IF *< expression booléenne> <clause alors> [<clause sinon>]* **FI** *;*

sémantique: 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, c'est la liste d'énoncés d'action qui suit **ELSE** , s'il y en a une, qui est entamée.

exemples:

```
7.22    IF n >= 50 THEN rn(r) := 'L';
            n- := 50;
            r+ := 1;
        FI                                              (1.1)
10.50   IF last = NULL
            THEN first,last := p;
            ELSE last->.succ := p;
            p->.pred := last;
            last := p;
        FI                                              (1.1)
```

6.4 ACTION 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 sélecteur de cas> ::= *(2)*
< expression discrète> { *,< expression discrète>*} * *(2.1)*

<liste d'intervalles> ::= *(3)*
<mode discret> { *,< mode discret>*} * *(3.1)*

<cas à choisir> ::= *(4)*
<spécification d'étiquettes de cas> : <liste d'énoncés d'action> *(4.1)*

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). Cette liste d'énoncés d'action est étiquetée par une spécification d'étiquettes de cas qui consiste en une liste d'étiquettes de cas (une pour chaque sélecteur de cas). Chaque liste d'étiquettes de cas définit un ensemble de valeurs. L'emploi d'une liste d'expressions discrètes dans la liste de sélecteurs de cas permet de choisir un cas suivant plusieurs conditions.

L'action de cas entame la liste d'énoncés d'action pour laquelle les valeurs données dans la spécification d'étiquettes de cas correspondent aux valeurs 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 10.1.3).

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 10.1.3) 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.

La représentation textuelle de nom simple **réservée** optionnelle **ELSE** , suivie 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 10.1.3).

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:

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

6.5 ACTION FAIRE

6.5.1 Généralités

syntaxe:

```
<action faire> ::=                                        (1)
        DO [ <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:

```
4.17    DO FOR i := 1 TO c;
            op(a,b,d,order-1);
            d := a;
        OD                                                    (1.1)
15.58   DO WITH each;
            IF this_counter = counter
                THEN
                 status := idle;
                  EXIT find_counter;
            FI ;
        OD                                                    (1.1)
```

6.5.2 Commande pour

syntaxe:

```
<commande pour> ::=                                           (1)
        FOR { <itération> { ,<itération>} * | EVER }          (1.1)

<itération> ::=                                               (2)
        <énumération de valeur>                               (2.1)
      | <énumération de locus>                                (2.2)

<énumération de valeur> ::=                                   (3)
        <énumération par pas>                                 (3.1)
      | <énumération par intervalle>                          (3.2)
      | <énumération ensembliste>                             (3.3)

<énumération par pas> ::=                                     (4)
        <compteur de boucle> <symbole d'affectation>
        <valeur initiale> [ <valeur de pas>] [ DOWN ] <valeur finale>   (4.1)

<compteur de boucle> ::=                                      (5)
        <définition >                                         (5.1)

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

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

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

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

<locus composite> ::= *(12)*
< locus rangée> *(12.1)*
| *< locus chaîne>* *(12.1)*

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 avec 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, 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. **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 dont la représentation textuelle de nom est la même que celle du 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 une valeur **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 PRECEDENTE) (PRED (VALEUR PRECEDENTE)).

   La termaison normale se produit si la liste d'énoncés d'action a été exécutée pour la plus grande (plus petite) valeur. Lorsque l'action faire est exécutée, l'expression ensembliste est évaluée une seule fois.

   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 se termine normalement quand la liste d'énoncés d'action a été exécutée pour la plus grande (plus petite) valeur. Quand l'action faire est exécutée, l'expression **ensembliste** n'est évaluée qu'une fois.

énumération par pas:

Dans le cas d'une énumération par pas sans (avec) spécification de **DOWN** , l'ensemble de valeurs du compteur de boucle est déterminé par une valeur initiale, 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 PRECEDENTE + VALEUR DE PAS (VALEUR PRECEDENTE - VALEUR DE PAS)

dans le cas d'une spécification de *valeur de pas*, sinon comme:

SUCC (VALEUR PRECEDENTE) (PRED (VALEUR PRECEDENTE)).

La terminaison normale se produit si l'évaluation donne une valeur qui est plus grande (plus petite) que la valeur finale, ou aurait causé l'exception *OVERFLOW* .

3. **énumération de locus**:

Dans le cas d'une énumération de locus sans (avec) spécification de **DOWN** , la liste d'énoncés d'action est entamée de façon répétée pour un ensemble de locus spécifiés qui sont les éléments du locus rangée dénoté par le *locus rangée* ou les composants du locus chaîne désigné par le *locus chaîne*. La sémantique est comme si initialement la déclaration de loc-identité:

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

était rencontrée, où *mode* est le mode des éléments du mode du *locus rangée*, où *&nom(1)* tel que *&nom* est un nom de **synmode** virtuel **synonyme** du mode du *locus chaîne*, et où *l'indice* est réglé initialement sur la **borne inférieure** (**borne supérieure**) du mode du *locus rangée* ou du *locus chaîne* et *l'indice* précédent chaque exécution subséquente de la liste d'énoncés d'action est réglé sur *SUCC (indice) (PRED (indice))*. La liste d'énoncés d'action ne sera pas entamée si la **longueur de chaîne** du mode du *locus chaîne* = 0.
L'action faire se termine (terminaison normale) si *l'indice* qui suit immédiatement une exécution de la liste d'énoncés d'action est égal à la **borne supérieure** (**borne inférieure**) du mode du *locus rangée* ou du *locus chaîne*.
Quand l'action faire est terminée, le *locus composé* n'est évalué qu'une seule fois.

propriétés statiques: A un compteur de boucle est rattaché une chaîne de noms qui est la chaîne de noms de sa définition.

énumération de valeur:

Le nom défini par le *compteur de boucle* est un nom **d'énumération de valeur**. Une chaîne de noms est visible dans le domaine dans lequel se trouve *l'action faire* qui est égale au *compteur de boucle*, le *compteur de boucle* est **explicite**, sinon il est **implicite**.

énumération par pas:

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

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

énumération ensembliste:

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

énumération de locus:

Le nom défini par le *compteur de boucle* est un nom **d'énumération de locus**. Son mode est le mode **des éléments** du mode du *locus rangée* ou le mode chaîne *&nom(1)*, où *&nom* est un nom de **synmode** virtuel **synonyme** du mode du *locus chaîne*.

Un nom **d'énumération de locus** est **repérable** 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 nom défini par le *compteur de boucle*.

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:

4.17	**FOR** *i := 1* **TO** *c*	*(1.1)*
15.37	**FOR EVER**	*(1.1)*
4.17	*i := 1* **TO** *c*	*(3.1)*
9.12	*j := MIN (sieve)* **BY** *MIN (sieve)* **TO** *max*	*(3.1)*
14.28	*i* **IN** *INT (1:100)*	*(3.2)*

6.5.3 Commande tandis

syntaxe:

<commande tandis> ::= *(1)*

WHILE < *expression booléenne*> *(1.1)*

sémantique: L'expression booléenne est évaluée juste avant d'entamer la liste d'énoncés d'action (après l'évaluation de la commande pour, si présente). Si elle donne *TRUE* , la liste d'énoncés d'action est entamée, sinon l'action faire est terminée (terminaison anormale).

exemples:

7.35	**WHILE** *n >= 1*	*(1.1)*

6.5.4 Partie avec

syntaxe:

<partie avec> ::= *(1)*

WITH *<commande avec>* { ,*<commande avec>*} * *(1.1)*

<commande avec> ::= *(2)*

< locus structure> *(2.1)*

| *< valeur primitive structure>* *(2.2)*

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

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

Les règles de visibilité se présentent comme si une définition de nom de champ était introduite pour chaque nom de champ attaché au mode du locus ou de la valeur primitive et ayant la même chaîne de nom que le nom de champ.

Si un *locus structure* est spécifié, des noms d'accès ayant la même chaîne de noms que les noms de champ du mode du *locus structure* sont implicitement définis, dénotant les sous-locus du locus structure.

Si une *valeur primitive structure* est spécifiée, des noms de valeur ayant la même chaîne de noms que les noms de champ du mode de la *valeur primitive structure* (**forte**) sont implicitement définis, dénotant les sous-valeurs de la valeur structure.

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

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

valeur primitive structure: Une définition (virtuelle) dans une *partie avec* définit un nom de **valeur faire-avec**. Sa classe est la M-classe par valeur, où M est le mode de ce nom de **champ** du mode *structure* de la *valeur primitive structure*, qui est rendue disponible comme **valeur** de nom *faire-avec*.

locus structure: Une définition (virtuelle) dans une *partie avec* définit un nom de **locus faire-avec**. Son mode est le mode de ce nom de **champ** du mode du *locus structure*, qui est rendu disponible comme nom de **locus faire-avec**. Un nom de **locus faire-avec** est **repérable** si l'implantation de champ du nom de **champ** associé est **NOPACK** .

exemples:

15.58 **WITH** *each* *(1.1)*

6.6 ACTION SORTIR

syntax:

<action sortir> ::= *(1)*

EXIT *<représentation textuelle de nom simple >* *(1.1)*

sémantique: Une action sortir est employée pour quitter un énoncé d'action parenthésé ou un module. L'exécution reprend immédiatement après l'énoncé d'action parenthésé englobant le plus proche ou le module étiqueté par la représentation textuelle de nom simple.

conditions statiques: L'*action sortir* doit résider à l'intérieur de l'énoncé d'action parenthésé ou du module dont la *définition* devant a la même représentation textuelle de nom simple que la *représentation textuelle de nom simple.*

Si l'*action sortir* se trouve à l'intérieur d'une définition de procédure ou d'une définition de processus, l'énoncé d'action parenthésé ou le module dont on sort doivent aussi résider à l'intérieur de la même définition de procédure ou de processus (c.-à-d. l'action sortir ne peut s'employer pour quitter des procédures ou des processus).

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

exemples:

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

6.7 ACTION APPELER

syntaxe:

<action appeler> ::= *(1)*
[**CALL**] { *<appel de procédure>* *(1.1)*
| *<appel d'opération prédéfinie CHILL>* *(1.2)*
| *< appel d'opération prédéfinie d'implémentation>*} *(1.3)*

<appel procédure> ::= *(2)*
{ *< nom de procédure >* | *<valeur primitive procédure >*}
([*<liste de paramètres effectifs>*]) *(2.1)*

<liste de paramètres effectifs> ::= *(3)*
<paramètre effectif> { ,*<paramètre effectif>*} * *(3.1)*

<paramètre effectif> ::= *(4)*
<valeur> *(4.1)*
| *<locus>* *(4.2)*

<appel d'opération prédéfinie CHILL> ::= *(5)*
<appel d'opération prédéfinie de valeur CHILL > *(5.1)*
| *<appel d'opération prédéfinie de locus CHILL >* *(5.2)*
| *<appel d'opération prédéfinie simple CHILL >* *(5.3)*

<appel d'opération prédéfinie simple CHILL> ::= *(6)*
TERMINATE (<expression repère>) *(6.1)*
| *<appel d'opération prédéfinie simple io CHILL >* *(6.2)*

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 l'appel soit d'une procédure soit d'une opération prédéfinie. Un appel de procédure cause un appel de la procédure générale indiquée par la valeur donnée par la valeur primitive procédure ou la procédure indiquée par le nom de la procédure. Les valeurs et les locus effectifs spécifiés dans la liste des paramètres effectifs sont transmis à la procédure.

Un appel d'opération prédéfinie CHILL est soit un appel d'opération prédéfinie de locus CHILL, qui donne un locus (voir section 4.2.12), soit un appel d'opération prédéfinie de valeur CHILL, qui donne une valeur (voir section 5.2.13) soit un appel d'opération prédéfinie simple CHILL, qui ne donne ni valeur ni locus. Les opérations prédéfinies simples d'entrée-sortie sont décrites dans le chapitre 7.

TERMINATE termine la durée de vie du locus repéré par la valeur donnée par l'*expression repère*. Une implémentation peut en conséquence libérer la mémoire occupée par ce locus. Si la durée de vie du locus s'est déjà terminée avant l'appel *TERMINATE* , aucune action n'est exécutée.

propriétés statiques: Un *appel de procédure* a les propriétés suivantes: il a une liste de **specs de paramètre**, éventuellement une **spec de résultat**, un ensemble éventuellement vide de noms d'exception, une **généralité**, une **récursivité**, et il peut être **intrarégional** (cette dernière propriété n'est possible que pour un *nom de procédure*, voir section 9.2.2). Ces propriétés sont héritées du *nom de procédure* ou d'un mode **compatible** avec la classe de la *valeur primitive procédure* (dans le dernier cas, la généralité est toujours **générale**).

Un *appel de procédure* qui a une **spec de résultat** est un *appel de procédure rendant locus* si et seulement si **LOC** est spécifié dans la spec de résultat, sinon c'est un *appel de procédure rendant valeur*.

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 non-valeur**. Le *paramètre effectif* est une *valeur* qui doit être **régionalement sûre** pour *l'appel de procédure.*

- Si la spec de paramètre a l'attribut **INOUT** ou **OUT** , le *paramètre effectif* doit être un *locus*, dont le mode doit être **compatible** avec la M-classe par valeur, où M est le mode dans la spec de paramètre correspondante. Le mode du *locus* (effectif) doit être statique et ne peut avoir la **propriété de protection** ni la **propriété de non-valeur**. Le *paramètre effectif* est un *locus*. Il peut être considéré comme une *valeur* qui doit être **régionalement sûre** pour *l'appel de procédure*.

- Si la spec de paramètre a l'attribut **INOUT** , le mode dans la spec de paramètre doit être **compatible** avec la M-classe par valeur, où M est le mode du *locus*.

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

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

- Si la spec de paramètre a l'attribut **LOC** , alors:

  - si le *paramètre effectif* est un *locus*, il doit avoir la même **régionalité** que *l'appel de procédure*;

  - si le *paramètre effectif* est une *valeur*, il doit être **régionalement sûr** pour *l'appel de procédure.*

conditions dynamiques: Un appel de procédure peut causer toute exception de l'ensemble de noms d'exception qui lui est attaché. Il cause l'exception *EMPTY* si la *valeur primitive 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 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 après le retour de la procédure.

- Si la spec de paramètre a l'attribut **LOC** et que le *paramètre effectif* est une *valeur* qui n'est pas un *locus*, les conditions d'affectation de la *valeur* (effective) en tenant compte du mode de la spec de paramètre doivent être respectées à l'endroit de l'appel et les exceptions possibles sont causées avant que la procédure ne soit appelée (voir section 6.2).

La *valeur primitive procédure* ne peut pas donner une procédure définie dans une définition de processus dont l'activation n'est pas la même que l'activation du processus exécutant l'appel de

procédure (autre que le processus imaginaire le plus extérieur) et la durée de vie de la procédure désignée ne doit pas être terminée.

TERMINATE cause l'exception *EMPTY* si *l'expression repère* donne la valeur *NULL* .

TERMINATE cause l'exception *TERMINATEFAIL* si *l'expression repère* ne donne pas une valeur repère **assignée**.

exemples:

4.18 op(a,b,d,order-1) *(1.1)*

6.8 ACTION RÉSULTER ET ACTION REVENIR

syntaxe:

<*action revenir*> ::= *(1)*
RETURN [<*résultat*>] *(1.1)*

<*action résulter*> ::= *(2)*
RESULT <*résultat*> *(2.1)*

<*résultat*> ::= *(3)*
<*valeur*> *(3.1)*
| <*locus*> *(3.2)*

syntaxe dérivée: L'*action revenir* avec *résultat* est dérivée de **RESULT** <*résultat*> ; **RETURN** . Si un *filet* termine une telle *action revenir*, il est considéré 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 une 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** (**LOC DYNAMIC**) est spécifié dans la **spec de résultat** du nom de **procédure** de l'*action résulter*, le *résultat* doit être un *locus*, tel que le mode dans la **spec de résultat** soit **compatible en lecture** (**compatible en lecture dynamique**) avec le mode du *locus*. Le *locus* doit être **repérable** si **NONREF** n'est pas spécifié dans **la spec de résultat**. Le *résultat* est un *locus* qui doit avoir la même **régionalité** que le *nom de procédure* attaché à l'*action résulter*.

Si **LOC** n'est pas spécifié dans la **spec de résultat** du nom **de procédure** de l'*action résulter*, le *résultat* doit être une *valeur* dont la classe est **compatible** avec le mode dans la **spec de résultat**. Le *résultat* est une *valeur* qui doit être **régionalement sûre** pour le *nom de procédure* attaché à l'*action résulter*.

conditions dynamiques: Si **LOC** n'est pas spécifié dans la **spec de résultat** du nom de **procédure**, les conditions d'affectation de la *valeur* dans l'*action résulter* en tenant compte du mode dans la **spec de résultat** de son nom de **procédure** doivent être respectées.

exemples:

4.21 **RETURN** *(1.1)*
1.6 **RESULT** *i+j* *(2.1)*
5.19 *c* *(3.1)*

6.9 ACTION ALLER

syntaxe:

<action aller> ::= *(1)*
GOTO *<représentation textuelle de nom simple>* *(1.1)*

sémantique: L'action aller cause un déplacement du point d'exécution. L'exécution continue à l'énoncé d'action étiqueté par la représentation textuelle de nom simple.

conditions statiques: Si l'*action aller* se trouve à l'intérieur d'une définition de procédure ou d'une définition de processus, l'étiquette indiquée par la *représentation textuelle de nom simple* 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:

4.7 **ASSERT** *b>0 AND c>0 AND order>0* *(1.1)*

6.11 ACTION VIDE

syntaxe:

<action vide> ::= *(1)*
<vide> *(1.1)*

<vide> ::= *(2)*

sémantique: 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 l'exception dont le nom est indiqué par *nom d'exception.*

conditions statiques: Un *filet* ne peut pas terminer une *action causer.*

exemples:

4.9 **CAUSE** *wrong_input* *(1.1)*

6.13 ACTION DÉMARRER

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

exemples:

14.45 **START** *call_distributor ( )* *(1.1)*

6.14 ACTION ARRÊTER

syntaxe:

<action arrêter> ::= *(1)*
STOP *(1.1)*

sémantique: L'action arrêter termine le processus qui exécute l'action arrêter (voir section 9.1).

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 9 pour plus de détails).

exemples:

13.25 **CONTINUE** *resource_freed* *(1.1)*

6.16 ACTION METTRE EN ATTENTE

syntaxe:

<action mettre en attente> ::= *(1)*
DELAY *< locus événement>* [*<priorité>*] *(1.1)*

<priorité> ::= *(2)*
PRIORITY *< expression littérale entière>* *(2.1)*

sémantique: 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 9 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:

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

6.17 ACTION METTRE EN ATTENTE ET CHOISIR

syntaxe:

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

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

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

sémantique: L'action mettre en attente et choisir cause la mise en attente du processus qui l'exécute. Il peut être réactivé 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 9 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 événement 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:

14.26
```
DELAY CASE
   (operator_is_ready): /* some actions */
   (switch_is_closed): DO FOR i IN INT (1:100);
                          CONTINUE operator_is_ready;
                       /* empty the queue */
                       OD ;
ESAC                                                    (1.1)
```

6.18 ACTION ENVOYER

6.18.1 Généralités

syntaxe:

<action envoyer> ::= *(1)*
<action envoyer signal> *(1.1)*
| *<action envoyer tampon>* *(1.2)*

sémantique: 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** < *valeur primitive 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 le signal. Cette identification de processus ne peut être en contradiction avec le nom de processus éventuellement attaché au nom de signal.

conditions statiques: Le nombre d'occurrences de *valeur* doit être égal au nombre de modes du *nom de signal*. La classe de chaque *valeur* doit être **compatible** avec le mode correspondant du *nom de signal*. Aucune occurrence de *valeur* ne peut être **intrarégionale** (voir section 9.2.2). 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 la *valeur primitive exemplaire* est terminée au point d'exécution de l'action envoyer signal.

L'exception *SENDFAIL* est causée si le *nom de signal* a un nom **de processus** qui n'est pas le nom du processus indiqué par la valeur donnée par la *valeur primitive exemplaire*.

exemples:

15.78 **SEND** *ready* **TO** *received_user* *(1.1)*
15.86 **SEND** *readout(count)* **TO** *user*

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 9 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 **intrarégionale** (voir section 9.2.2). 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:

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

6.19 ACTION RECEVOIR ET CHOISIR

6.19.1 Généralités

syntaxe:

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

sémantique: 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 9 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 définitions>*]) : *<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, le *locus exemplaire* est évalué 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.

Lorsqu'un signal est reçu, on entame une liste d'énoncés d'action portant le nom de **signal** du signal reçu. Si plus d'un signal est reçu, un signal de la priorité la plus élevée sera choisi conformément à un algorithme de programmation défini par l'implémentation. Si une liste de modes est attachée au nom du **signal**, c.-à-d. qu'une liste de valeurs est envoyée avec le signal, une liste de définitions doit être spécifiée après **IN** .
Elles définissent les noms de **valeur recevoir** désignant les valeurs reçues. Si, dans le domaine dans lequel se trouve l'action recevoir signal et choisir, un nom d'accès est visible, qui est égal à un nom introduit, la valeur reçue sera mise dans le locus désigné, immédiatement après réception du signal et avant exécution de la liste d'énoncés d'action.

Si l'option **SET** est spécifiée et que le signal est reçu, la valeur **exemplaire** qui dénote le processus ayant envoyé le signal reçu sera mise dans le *locus exemplaire* immédiatement après réception du signal et avant *introduction du signal à choisir.*

propriétés statiques: Une *définition* de la *liste de définitions* d'un *signal à choisir* définit un nom de **valeur reçue**. Sa classe est la M-classe par valeur, où M est le mode correspondant 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 c'est un nom de valeur reçue (signal) **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 définitions* 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 définitions* 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**. Le mode du *nom d'accès* ne peut pas avoir la **propriété de protection**; s'il a la **propriété de repérabilité**, le *nom d'accès* doit être **extérieur à la région**.

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:

15.83
```
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** <*définition*>) : <*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 tampons 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 les deux points. La définition après **IN** définit 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 et que la valeur est reçue, la valeur **exemplaire** dénotant le processus qui a envoyé la valeur reçue sera mise dans le *locus exemplaire* immédiatement après réception de la valeur et avant introduction de *tampon à choisir*.

propriétés statiques: Une *définition* dans un *tampon à choisir* définit un nom de **valeur reçue**. Sa classe est la M-classe par valeur, où M est le mode de **l'élément tampon** du mode du *locus tampon* qualifiant le *tampon à choisir*.

Si un nom est visible dans le domaine dans lequel est placée *l'action recevoir tampon* est choisi, et égal au *nom* introduit après **IN** , le nom de **valeur reçue** est dit **explicite**, sinon il est **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**; s'il a la **propriété de repérage**, alors, le *nom d'accès* doit être **extrarégional**.

conditions dynamiques: Si le nom de la **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.

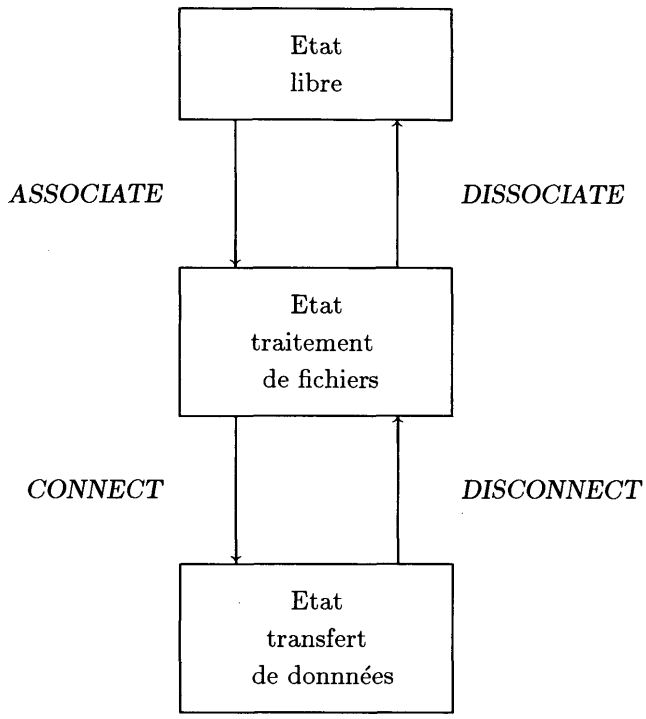
La durée de vie d'aucun des locus **tampon** donnés ne peut se terminer pendant que le processus qui exécute l'action recevoir et choisir est en attente sur lui.

7 ENTRÉE ET SORTIE

7.1 MODÈLE DE RÉFÉRENCE I/O

Un modèle est utilisé pour la description des facilités d'entrée-sortie, d'une manière indépendante de l'implémentation; on distingue trois états pour un locus association donné: un état libre, un état de traitement des fichiers et un état de transfert de données.

Le diagramme ci-après représente ces trois états et les transitions possibles entre ceux-ci.



Le locus association ne contient pas de valeur.
Aucune relation avec un objet du monde extérieur.

Le locus association contient une association.
Des opérations créent et suppriment un
fichier, ou en modifient les propriétés.

Un locus accès est relié au locus association.
Transfert de données vers un fichier ou à partir
de celui-ci: opérations lire et écrire.

Le modèle est fondé sur l'hypothèse que des objets, qui dans des implémentations, sont souvent appelés ensembles de données, fichiers, ou dispositifs, existent dans le **monde extérieur**, c.-à-d. dans un milieu extérieur à un programme CHILL. Dans le modèle, cet objet du monde extérieur est appelé un **fichier**. Un fichier peut être un dispositif matériel, une ligne de communication ou simplement un fichier d'un système de gestion de fichiers. En général, un fichier est un objet qui peut produire et/ou utiliser des données.

En CHILL, l'utilisation d'un fichier nécessite une **association**; une association est créée par l'opération associate et elle identifie un fichier. Une association a des **attributs**; ces attributs décrivent les propriétés d'un fichier qui est ou peut être lié à l'association.

Dans **l'état libre**, il n'y a ni interaction ni relation entre le programme CHILL et des objets du monde extérieur. L'opération associate modifie l'état du modèle, qui passe de l'état libre à **l'état traitement de fichiers**. Cette opération prend pour argument un locus association et une dénotation définie par l'implémentation pour un objet du monde extérieur pour lequel une association doit être créée; des arguments supplémentaires peuvent être utilisés pour indiquer le type d'association de l'objet et les valeurs initiales des attributs de l'association. En outre, une association particulière implique un ensemble d'opérations (dépendant de l'implémentation) qui peut être appliqué au fichier attaché à cette association.

Dans l'état traitement de fichiers, il est possible de manipuler un fichier et ses propriétés par l'intermédiaire d'une association, à condition que l'association permette cette opération particulière; pour des opérations qui modifient les propriétés d'un fichier, une association réservée exclusivement au fichier sera normalement nécessaire.

Dans le modèle, on admet que les associations sont généralement exclusives, c.-à-d. qu'une seule association existe au même moment pour un objet donné du monde extérieur. Toutefois, des implémentations peuvent admettre la création de plusieurs associations pour le même objet, à condition que cet objet puisse être partagé

par différents utilisateurs (programmes) et/ou différentes associations dans le même programme. Toutes les opérations effectuées dans l'état traitement de fichiers prennent une association pour argument.

L'opération **dissociate** est utilisée pour mettre fin à une association relative à un objet du monde extérieur; cette opération fait que le locus revient de l'état traitement de fichiers à l'état libre.

Le transfert de données vers un fichier ou à partir de celui-ci n'est possible que dans **l'état transfert de données**; les opérations de transfert exigent qu'un locus **accès** soit **connecté** à une association relative à ce fichier. L'opération de connexion relie un locus accès à une association et modifie l'état du modèle, qui passe à l'état transfert de données. L'opération prend pour arguments un locus association et un locus accès; le locus association contient une association avec un fichier dans lequel ou à partir duquel les données peuvent être transférées par l'intermédiaire du locus accès. Des arguments supplémentaires de l'opération de connexion indiquent pour quel type d'opération de transfert le locus accès doit être connecté et dans quel registre le fichier doit être classé. Un locus accès au plus peut être connecté avec un locus association à un moment donné.

L'opération **disconnect** prend pour argument un locus accès et le déconnecte de l'association à laquelle il est relié; elle modifie l'état du modèle, qui revient à l'état traitement de fichiers.

Dans l'état transfert de données, un locus accès doit être utilisé comme argument d'une opération de transfert; deux opérations de transfert sont possibles, à savoir une opération **lire**, pour transférer des données d'un fichier au programme, et une opération **écrire**, pour transférer des données du programme à un fichier. Les opérations de transfert utilisent le mode enregistrement du locus accès pour transformer des valeurs CHILL en enregistrement de fichiers, et vice-versa.

Dans le modèle, un fichier se présente comme une **rangée de valeurs**; chaque élément de cette rangée se rapporte à un enregistrement du fichier. Le mode des éléments de cette rangée est déterminée par l'opération de connexion qui est le mode enregistrement du locus accès qui est connecté. Une valeur d'indice est affectée à chaque enregistrement du fichier; cette valeur identifie de manière unique chaque enregistrement du fichier. Dans la description des opérations de connexion et de transfert, trois valeurs d'indice spéciales seront utilisées, à savoir un indice de **base**, un indice **courant** et un indice de **transfert**. L'indice de base est fixé par l'opération de connexion et reste inchangé jusqu'à une opération de connexion suivante. Il est utilisé pour calculer l'indice de transfert dans des opérations de transfert et l'indice courant dans une opération de connexion. L'indice de transfert indique dans le fichier la position où un transfert aura lieu; l'indice courant désigne l'enregistrement sur lequel le fichier est actuellement placé.

7.2 VALEURS D'ASSOCIATION

7.2.1 Généralités

Une valeur d'association reflète les propriétés d'un fichier qui est ou qui peut lui être rattaché. Une valeur d'association déterminée applique en outre un ensemble d'opérations (dépendant de l'implémentation) sur le fichier qui lui est éventuellement rattaché.

Les valeurs d'association ne possèdent pas de dénotation mais elles sont contenues dans des locus de mode association; il n'existe pas d'expression désignant une valeur de mode association. Les valeurs d'association ne peuvent être manipulées que par des opérations prédéfinies qui prennent un locus d'association pour paramètre.

7.2.2 Attributs des valeurs d'association

Une valeur d'association a des attributs, qui décrivent les propriétés de l'association et le fichier qui peut ou qui pourrait y être rattachés.

Les attributs suivants sont définis par le langage:

- **existant** : un fichier (éventuellement vide) est rattaché à l'association;
- **lisible** : les opérations lire sont possibles pour le fichier lorsqu'il est rattaché à l'association;
- **écrivable** : les opérations écrire sont possibles pour le fichier lorsqu'il est rattaché à l'association;
- **indexable** : lorsqu'il est rattaché à l'association, le fichier permet l'accès aléatoire à ses enregistrements;

- **séquençable** : lorsqu'il est rattaché à l'association, le fichier permet l'accès séquentiel à ses enregistrements;

- **variable** : la taille des enregistrements du fichier, lorsque celui-ci est rattaché à l'association, peut varier à l'intérieur du fichier.

Ces attributs ont une valeur booléenne; les attributs sont initialisés lorsque l'association est créée et peuvent être mis à jour à la suite d'opérations particulières sur l'association. Cette liste ne comprend que des attributs définis par le langage; des implémentations peuvent ajouter des attributs selon leurs propres besoins.

7.3 VALEURS D'ACCÈS

7.3.1 Généralités

Des valeurs d'accès sont contenues dans des locus de mode accès. Il faut un locus accès pour transférer des données d'un fichier au monde extérieur ou viceversa.

Les valeurs d'accès n'ont pas de dénotation mais sont contenues dans des locus de mode accès; il n'existe pas d'expression désignant une valeur de mode accès. Les valeurs d'accès ne peuvent être manipulées que par des opérations prédéfinies qui prennent pour paramètre un locus d'accès.

7.3.2 Attributs des valeurs d'accès

Les valeurs ont des attributs qui décrivent leurs propriétés dynamiques, la sémantique des opérations de transfert et les conditions dans lesquelles des exceptions peuvent se produire.

CHILL définit les attributs suivants:

- **usage** : indiquant pour quelle(s) opération(s) de transfert le locus accès est connecté à une association; l'attribut est fixé par l'opération de connexion;

- **hors du fichier** : indiquant si l'indice de transfert calculé par la dernière opération lire était ou non dans le fichier; l'attribut est initialisé sur *FALSE* par l'opération de connexion et fixé par chaque opération lire.

7.4 OPÉRATIONS PRÉDÉFINIES POUR ENTRÉE-SORTIE

7.4.1 Généralités

Les opérations prédéfinies par le langage sont définies pour des opérations sur des locus association et des locus accès ainsi que pour examiner et modifier les attributs de leurs valeurs.

syntaxe:

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

<appel d'opération prédéfinie d'e/s simple de CHILL> ::= *(2)*
< appel d'opération prédéfinie d'e/s simple de CHILL désassocier> *(2.1)*
| < appel d'opération prédéfinie d'e/s simple de CHILL modification> *(2.2)*
| < appel d'opération prédéfinie d'e/s simple de CHILL connecter> *(2.3)*
| < appel d'opération prédéfinie d'e/s simple de CHILL déconnecter> *(2.4)*
| < appel d'opération prédéfinie d'e/s simple de CHILL écrire article> *(2.5)*

<appel prédéfini d'e/s rendant locus de CHILL> ::= *(3)*
< appel d'opération prédéfinie d'e/s rendant locus de CHILL > *(3.1)*

Les opérations prédéfinies seront décrites dans les sections qui suivent.

7.4.2 Association avec un objet du monde extérieur

syntaxe:

<appel d'opération prédéfinie d'e/s rendant locus de CHILL associer> ::= (1)
ASSOCIATE (< locus association>[,<liste de paramètres pour associer>]) (1.1)

<appel d'opération prédéfinie d'e/s rendant valeur de CHILL est associé> ::= (2)
ISASSOCIATED (< locus association>) (2.1)

<liste de paramètres pour associer> ::= (3)
<paramètre pour associer> { ,*<paramètre pour associer>* } * (3.1)

<paramètre pour associer> ::= (4)
<locus> (4.1)
| *<valeur>* (4.2)

sémantique: *ASSOCIATE* crée une association avec un objet du monde extérieur. Il initialise le *locus association* avec l'association créée. Il initialise les attributs de l'association créée. En outre, le locus association est renvoyé comme résultat de l'appel. L'association particulière qui est créée est déterminée par les locus et/ou les valeurs qui apparaissent dans la *liste de paramètres pour associer*; les modes (classes) et la sémantique de ces locus (valeurs) sont définis par l'implémentation.

ISASSOCIATED renvoie *TRUE* si le *locus association* contient une association et, sinon, *FALSE* .

propriétés statiques: La classe d'un appel d'opération prédéfinie *ISASSOCIATED* est la *BOOL* -classe par dérivation. Le mode de l'appel d'opération prédéfinie *ASSOCIATE* est le mode du *locus association*.

conditions statiques: Le mode et la classe de chaque *paramètre pour associer* sont définis par l'implémentation.

conditions dynamiques: *ASSOCIATE* cause l'exception *ASSOCIATEFAIL* si le *locus association* contient déjà une association ou si l'association peut être créée pour des raisons définies par l'implémentation.

exemple:

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

7.4.3 Dissociation d'un objet du monde extérieur

syntaxe:

< appel d'opération prédéfinie d'e/s simple de CHILL désassocier > ::= (1)
DISSOCIATE (< locus association>) (1.1)

sémantique: *DISSOCIATE* met fin à une association avec un objet du monde extérieur. Si un locus accès est encore relié à l'association contenue dans un locus association, il est déconnecté avant que l'association ne soit terminée.

conditions dynamiques: *DISSOCIATE* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

exemple:

22.38 DISSOCIATE (association); (1.1)

7.4.4 Accès aux attributs association

syntaxe:

< appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association> ::= *(1)*
EXISTING (< locus association>) *(1.1)*
| *READABLE (< locus association>)* *(1.2)*
| *WRITEABLE (< locus association>)* *(1.3)*
| *INDEXABLE (< locus association>)* *(1.4)*
| *SEQUENCIBLE (< locus association>)* *(1.5)*
| *VARYING (< locus association>)* *(1.6)*

sémantique: *EXISTING* , *READABLE* , *WRITEABLE* , *INDEXABLE* , *SEQUENCIBLE* et *VARYING* rendent respectivement la valeur de l'attribut **existant**, soit **lisible**, **écrivable**, **indexable**, **séquençable** et **variable**, de l'association contenue dans le *locus association*.

propriétés statiques: La classe de *l'appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association* est la *BOOL* -classe par dérivation.

conditions dynamiques: *L'appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

7.4.5 Modification des attributs association

syntaxe:

< appel d'opération prédéfinie d'e/s simple de CHILL modification> ::= *(1)*
CREATE (< locus association>) *(1.1)*
| *DELETE (< locus association>)* *(1.2)*
| *MODIFY (< locus association>[,<liste de paramètres pour modifier>])* *(1.3)*

<liste de paramètres pour modifier> ::= *(2)*
<paramètre pour modifier> { , *<paramètre pour modifier>* } * *(2.1)*
<paramètre pour modifier> ::= *(3)*
<valeur> *(3.1)*
| *<locus>* *(3.2)*

sémantique: *CREATE* crée un fichier vide et le rattache à l'association désignée par le *locus association*. L'attribut **existant** de l'association indiquée donne *TRUE* si l'opération réussit.

DELETE détache un fichier de l'association désignée par le *locus association* et supprime le fichier. L'attribut **existant** de l'association indiquée donne *FALSE* si l'opération réussit.

MODIFY fournit les moyens de changer les propriétés d'un objet du monde extérieur pour lequel il existe une association et qui est désigné par *locus association*; les locus et/ou les valeurs qui apparaissent dans la *liste de paramètres pour modifier* indiquent comment modifier les propriétés. Les modes (classes) et la sémantique de ces locus (valeurs) sont définis par l'implémentation.

conditions dynamiques: *CREATE* , *DELETE* et *MODIFY* causent l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

CREATE cause l'exception *CREATEFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *TRUE*;
- la création du fichier échoue (définie par l'implémentation).

DELETE cause l'exception *DELETEFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *FALSE*;
- la suppression du fichier échoue (définie par l'implémentation).

MODIFY cause l'exception *MODIFYFAIL* si les propriétés, définies par la *liste de paramètres pour modifier* peuvent ou ne peuvent pas être modifiées; les conditions dans lesquelles cette exception peut se produire sont définies par l'implémentation.

exemple:

```
21.39   CREATE (outassoc);                                   (1.1)
21.69   DELETE (curassoc);                                   (1.2)
```

7.4.6 Connection d'un locus accès

syntaxe:

```
< appel d'opération prédéfinie d'e/s simple de CHILLconnecter> ::=          (1)
        CONNECT (< locus accès>,< locus association>, <expression usage>
        [,<expression positionnement >[,<expression indice > ]])            (1.1)

<expression usage> ::=                                                      (2)
        <expression>                                                        (2.1)

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

<expression indice> ::=                                                     (4)
        <expression>                                                        (4.1)
```

modes ensemble prédéfinis: Pour commander l'opération de connexion exécutée par l'opération prédéfinie *CONNECT* , deux noms de mode ensemble sont prédéfinis dans le langage, à savoir *USAGE* et *WHERE*; les modes ensemble correspondants sont: **SET** (*READONLY* , *WRITEONLY* , *READ - WRITE*) et **SET** (*FIRST* , *SAME* , *LAST*), respectivement.

sémantique: *CONNECT* rattache le *locus accès* à l'association qui est contenue dans le *locus association*; il doit y avoir un fichier rattaché à l'association désignée c.-à-d. que l'attribut **existant** doit être *TRUE*.

La valeur donnée par l'*expression usage* indique pour quel type d'opérations de transfert le locus accès doit être relié à un fichier. Si l'expression donne *READONLY* , la connexion n'est établie que pour des opérations lire; si elle donne *WRITEONLY* , la connexion n'est établie que pour des opérations écrire; si elle donne *READWRITE* , la connexion est établie pour des opérations lire et écrire.

L'attribut **indexable** de l'association désignée doit être *TRUE* si le locus accès a un mode d'indice, tandis que l'attribut **séquençable** doit être *TRUE* si le locus n'a pas de mode d'indice.

CONNECT (re)place le fichier qui est rattaché à l'association désignée c.-à-d. qu'il établit un indice de **base** et un indice **courant** (nouveaux) dans le fichier. Le (nouvel) indice de **base** dépend de la valeur donnée par *l'expression positionnement*:

- si *l'expression positionnement* donne *FIRST* ou n'est pas spécifiée, l'indice de **base** est réglé sur 0, c.-à-d. que le fichier est positionné avant le premier enregistrement;

- si *l'expression positionnement* donne *SAME* , l'indice de **base** est réglé sur l'indice **courant** du fichier, c.-à-d. que la position du fichier n'est pas modifiée;

- si *l'expression positionnement* donne *LAST* , l'indice de **base** est réglé sur N, où N désigne le nombre d'enregistrements dans le fichier, c.-à-d. que le fichier est positionné après le dernier enregistrement.

Une fois fixé l'indice de **base**, un indice **courant** sera établi par *CONNECT* . Cet indice **courant** dépend de la spécification facultative d'une *expression d'indice*:

- si aucune *expression d'indice* n'est spécifiée, l'indice **courant** est fixé sur le (nouvel) indice de **base**;

- si une *expression d'indice* est spécifiée, l'indice courant est fixé sur
  indice de **base** + *NUM (v)* – *NUM (l)*
  où *l* désigne la **borne inférieure** du mode d'indice du locus accès et *v* désigne la valeur donnée par *l'expression d'indice.*

Si le locus accès est connecté pour les opérations écrire séquentielles (c.-à-d. le locus accès n'a pas de mode d'indice et *l'expression usage* donne *WRITEONLY*), alors les enregistrements du fichier qui ont un indice supérieur à l'indice **courant** (nouveau) sont supprimés du fichier, c.-à-d. que le fichier peut être tronqué ou vidé par *CONNECT* .

Un locus accès qui n'a pas de mode d'indice ne peut être connecté simultanément à une association pour des opérations lire et écrire.

Tout locus accès auquel l'association désignée peut être connectée sera déconnecté implicitement avant la connexion de l'association au locus désigné par le *locus accès*.

CONNECT initialise l'attribut **outoffile** du locus d'accès sur *FALSE* et fixe l'attribut **usage** conformément à la valeur donnée par *l'expression usage.*

conditions statiques: Le mode du *locus accès* doit avoir un mode **d'indice** si une *expression d'indice* est spécifiée; la classe de la valeur donnée par *l'expression d'indice* doit être **compatible** avec ce mode **d'indice**.

La classe de la valeur donnée par *l'expression usage* doit être **compatible** avec la *USAGE*-classe par dérivation.

La classe de la valeur donnée par *l'expression positionnement* doit être **compatible** avec la *WHERE*-classe par dérivation.

conditions dynamiques: *CONNECT* cause l'exception *NOTASSOCIATED* si le *locus association* ne contient pas d'association.

CONNECT cause l'exception *CONNECTFAIL* si l'une des conditions suivantes se vérifie:

- l'attribut **existant** de l'association est *FALSE*;

- l'attribut **lisible** de l'association est *FALSE* et *l'expression usage* donne *READONLY* ou *READWRITE* ;

- l'attribut **écrivable** de l'association est *FALSE* et *l'expression usage* donne *WRITEONLY* ou *READWRITE* ;

- l'attribut **indexable** de l'association est *FALSE* et le *locus accès* a un mode **d'indice**;

- l'attribut **séquençable** de l'association est *FALSE* et le *locus accès* n'a pas de mode **d'indice**;

- *l'expression positionnement* donne *SAME* , tandis que l'association contenue dans le *locus association* n'est pas connectée à un locus accès;

- l'attribut **variable** de l'association est *FALSE* et le *locus* accès a un mode d'enregistrement dynamique, tandis que *l'expression usage* donne *WRITEONLY* ou *READWRITE* ;
- l'attribut **variable** de l'association est *TRUE* et le *locus* accès a un mode d'enregistrement statique, tandis que *l'expression usage* donne *READONLY* ou *READWRITE* ;
- le *locus* accès n'a pas de mode **d'indice**, tandis que *l'expression usage* donne *READWRITE*;
- l'association contenue dans le *locus association* ne peut être connectée au *locus* accès, en raison de conditions définies dans l'implémentation.

CONNECT cause l'exception *RANGEFAIL* si le mode **d'indice** du *locus* accès est un mode rangée et que *l'expression d'indice* donne une valeur extérieure aux bornes de ce mode rangée.

exemple:

20.22 *CONNECT (record_file, file_association, READWRITE);* *(1.1)*
20.24 *READONLY* *(2.1)*

7.4.7 Déconnexion d'un locus accès

syntaxe:

< appel d'opération prédéfinie d'e/s simple de CHILL déconnecter> ::= *(1)*
DISCONNECT (< locus accès>) *(1.1)*

sémantique: *DISCONNECT* déconnecte le *locus* accès dénoté par le *locus* accès de l'association à laquelle il était connecté.

conditions dynamiques: *DISCONNECT* cause l'exception *NOTCONNECTED* si le *locus* accès n'est pas connecté à une association.

7.4.8 Attributs d'accès de locus accès

syntaxe:

< appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association> ::= *(1)*
GETASSOCIATION (< locus accès>) *(1.1)*
| *GETUSAGE (< locus accès>)* *(1.2)*
| *OUTOFFILE (< locus accès>)* *(1.3)*

sémantique: *GETASSOCIATION* renvoie une valeur de référence au locus association auquel le *locus* accès est connecté; il renvoie *NULL* si le *locus* accès n'est pas connecté à une association.

GETUSAGE renvoie la valeur de l'attribut **usage**, c.-à-d. *READONLY* (*WRITEONLY*) si le *locus* accès n'est connecté que pour des opérations lire (écrire), ou *READWRITE* si le *locus* accès est connecté pour des opérations lire et écrire.

OUTOFFILE renvoie la valeur de l'attribut **outoffile** du *locus* accès, c.-à-d. *TRUE* si la dernière opération lire a calculé un indice de transfert qui n'était pas dans le fichier, sinon *FALSE*.

propriétés statiques: La classe d'un appel d'opération prédéfinie *GETASSOCIATION* est la *ASSOCIATION*-classe par repère.

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

La classe d'un appel d'opération prédéfinie *GETUSAGE* est la *USAGE*-classe par dérivation.

conditions dynamiques: *GETUSAGE* et *OUTOFFILE* causent l'exception *NOTCONNECTED* si le *locus* accès n'est pas connecté à une association.

exemple:

21.47 OUTOFFILE (infiles (FALSE)) (1.3)

7.4.9 Opérations de transfert de données

syntaxe:

< appel d'opération prédéfinie d'e/s rendant valeur de CHILL lire article> ::= (1)
READRECORD (< locus accès>[,<expression d'indice>] [,<locus de lecture>]) (1.1)

< appel d'opération prédéfinie d'e/s simple de CHILL écrire article> ::= (2)
WRITERECORD (< locus accès>[,<expression d'indice>], <expression écrire>) (2.1)

<locus de lecture> ::= (3)
< locus de mode statique> (3.1)

<expression écrire> ::= (4)
<expression> (4.1)

Note: La syntaxe de l'opération prédéfinie *READRECORD* est ambiguë; la solution consiste à utiliser le mode *locus accès*.

sémantique: Pour le transfert de données à un fichier ou à partir de celui-ci, les opérations prédéfinies *WRITERECORD* et *READRECORD* sont définies. Le *locus accès* doit avoir un mode enregistrement et il doit être connecté à une association pour transférer des données à ou à partir d'un fichier rattaché à cette association. La direction du transfert ne doit pas être en contradiction avec la valeur effective de l'attribut **usage** du *locus accès*.

La position dans le fichier de l'enregistrement à transférer est calculée avant que le **transfert** soit effectué. Si le *locus accès* n'a pas de mode **d'indice**, l'indice de **transfert** est l'indice **actuel** augmenté de 1; si le *locus accès* a un mode **d'indice**, l'indice de **transfert** est calculé comme suit:

indice de **transfert** := indice de **base** + *NUM (v)* − *NUM (l)* + *1*

où *l* est la **borne inférieure** du mode **d'indice** du *locus accès* et *v* dénote la valeur donnée par *l'expression d'indice*. Si le transfert de l'enregistrement portant l'indice de **transfert** calculé a été exécuté avec succès, l'indice **actuel** devient l'indice de **transfert**.

Les opérations lire:

READRECORD transfère au programme CHILL des données d'un fichier du monde extérieur.

Si l'indice de **transfert** calculé n'est pas dans le fichier, l'attribut **outoffile** est fixé sur *TRUE*, sinon, le fichier est positionné et l'enregistrement mis en lecture.

L'enregistrement en lecture ne doit pas donner une valeur **indéfinie**; l'effet de l'opération lire est défini par l'implémentation si l'enregistrement du fichier mis en lecture n'a pas une valeur correcte conformément au mode enregistrement du *locus accès*.

Si le *locus de lecture* est spécifié, alors, la valeur de l'enregistrement qui a été lu est affectée à ce locus. Si aucun *locus de lecture* n'est spécifié, la valeur sera affectée à un locus créé implicitement; la durée de vie de ce locus se termine lorsque le *locus accès* est déconnecté ou reconnecté. On ne précise pas si le locus repéré est créé une seule fois par l'opération de connexion ou chaque fois qu'une opération lire est exécutée.

READRECORD renvoie dans les deux cas une valeur repère qui se rapporte au locus (de mode éventuellement dynamique) auquel la valeur est affectée.

Les opérations écrire:

WRITERECORD transfère des données d'un programme CHILL à un fichier du monde extérieur. Le fichier est positionné sur l'enregistrement portant l'indice calculé et l'enregistrement est écrit.

Une fois l'enregistrement écrit, le nombre d'enregistrements est fixé sur l'indice de transfert, si ce dernier est supérieur au nombre effectif d'enregistrements.

L'enregistrement écrit par *WRITERECORD* est la valeur donnée par *l'expression écrire*; la classe de cette valeur ne peut être dynamique que si le *locus* accès a un mode d'enregistrement dynamique.

propriétés statiques: La classe de l'appel d'opération prédéfinie *READRECORD* est la M-classe par repère, où M est le mode enregistrement du *locus* accès, s'il a un mode d'enregistrement statique, ou une version paramétrée dynamiquement, si le locus a un mode d'enregistrement dynamique; les paramètres de cet enregistrement paramétré dynamiquement sont les suivants:

- la **longueur de chaîne** dynamique de la valeur chaîne lue dans le cas d'un mode chaîne;
- la **borne supérieure** dynamique de la valeur rangée lue dans le cas d'un mode rangée;
- la liste de valeurs (étiquette) associées au mode de la valeur structure lue dans le cas d'un mode structure variable.

conditions statiques: Le *locus* accès doit avoir un mode enregistrement.

Une *expression d'indice* peut ne pas être spécifiée si le *locus* accès n'a pas de mode d'indice et doit être spécifiée si le *locus* accès a un mode d'indice. La classe de la valeur donnée par *l'expression d'indice* doit être **compatible** avec ce mode d'indice.

Si le *locus enregistrement* est spécifié, alors, le mode du *locus enregistrement* et le mode d'enregistrement du *locus* accès doivent être **équivalents**. Le *locus de lecture* doit être **repérable**.

Le mode du *locus de lecture* ne peut pas avoir la **propriété d'être protégé**.

La classe de la valeur donnée par *l'expression écrire* doit être **compatible** avec le mode enre - istrement du *locus* accès; la classe peut être dynamique si et seulement si le *locus* accès a un mode d'enregistrement dynamique.

conditions dynamiques: Les appels d'opération prédéfinie *READRECORD* et *WRITERECORD* causent l'exception *NOTCONNECTED* si le *locus* accès n'est pas connecté à une association.
Les appels d'opération prédéfinie *READRECORD* ou *WRITERECORD* causent l'exception *RANGEFAIL* si le mode d'indice du *locus* accès est un mode intervalle et *l'expression d'indice* donne une valeur extérieure aux bornes de ce mode intervalle.

L'appel d'opération prédéfinie *READRECORD* cause l'exception *READFAIL* si l'une des conditions suivantes se vérifie:

- la valeur de l'attribut **usage** est *WRITEONLY* ;
- la valeur de l'attribut **outoffile** est *TRUE*;
- la lecture de l'enregistrement ayant l'indice calculé échoue en raison de conditions du monde extérieur.

L'appel d'opération prédéfinie *WRITERECORD* cause l'exception *WRITEFAIL* si et seulement si l'une des conditions suivantes se vérifie:

- la valeur de l'attribut **usage** est *READONLY* ;
- l'opération d'écriture de l'enregistrement portant l'indice calculé échoue, en raison de conditions du monde extérieur.

L'appel d'opération prédéfinie *WRITERECORD* cause l'exception *RANGEFAIL* si le locus accès a un mode enregistrement qui est un mode intervalle et que *l'expression écrire* donne une valeur extérieure aux bornes de ce mode intervalle.

Si l'exception *RANGEFAIL* ou l'exception *NOTCONNECTED* se produit, alors, elle se présente avant que la valeur de tout attribut ne soit modifiée et avant que le fichier soit positionné.

exemple:

20.24	*READRECORD (record_file, curindex, record_buffer);*	*(1.1)*
22.25	*READRECORD (fileaccess);*	*(1.1)*
20.32	*WRITERECORD (record_file, curindex, record_buffer);*	*(2.1)*
21.61	*WRITERECORD (outfile, buffers(flag));*	*(2.1)*
20.24	*record_buffer*	*(3.1)*
21.61	*buffers(flag)*	*(4.1)*

8 STRUCTURE DE PROGRAMME

8.1 GÉNÉRALITÉS

Les *actions faire*, *bloc début-fin*, *module*, *région*, *spec de module*, *spec de région*, *quasi module*, *quasi région*, *contexte*, *action recevoir*, *définition de procédure* et *définition de processus* déterminent la structure du programme, c-à-d. qu'elles déterminent la portée des noms et la durée de vie des locus qui y sont créés.

- Le mot **bloc** sera employé pour dénoter:
  - la *liste d'énoncés d'action* dans 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* et la *spec de paramètre* de tous les *paramètres formels* de la *liste de paramètres formels*;
  - la *définition de processus* en excluant la *spec de paramètre* de tous les *paramètres formels* de la *liste de paramètres formels*;
  - la *liste d'énoncés d'action* dans un *tampon à choisir* ou dans un *signal à choisir*, y compris la *définition* ou *liste de définitions* 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:
  - un *module* ou une *région*, en excluant des *contextes*, les *définitions* et/ou les *filets*, s'il en existe;
  - un *quasi module* ou une *quasi région*;
  - une *spec de module* ou une *spec de région*, en excluant les *contextes*, s'il en existe;
  - un *contexte*.
- Le mot **groupe** sera employé pour dénoter soit un **bloc** soit un **modulion**.
- Le mot **domaine** ou **domaine d'un groupe** sera employé pour dénoter cette partie du groupe qui n'est pas englobée (voir section 8.2) par un groupe interne au groupe (c.-à-d. la partie qui consiste en le niveau d'imbrication le plus externe du groupe).

Un groupe influence la portée de chacun des *noms* **créés** dans son domaine.
Des noms peuvent être créés par des *définitions* :

- Un *nom* qui apparaît dans la *liste de définitions* d'une *déclaration*, d'une *définition de mode*, ou d'une *définition de synonyme*, ou qui apparaît dans une *définition de signal* crée un *nom* dans le domaine dans lequel respectivement la *déclaration*, la *définition de mode*, la *définition de synonyme* ou la *définition du signal* apparaît.
- Une *définition* qui apparaît dans un *mode ensemble* créée un nom dans le domaine qui englobe immédiatement le *mode ensemble*.
- Une *définition* qui apparaît dans la *liste de définitions* dans une *liste de paramètres formels* crée un nom dans le domaine de la *définition de procédure* ou de la *définition de processus* correspondante.
- Une *définition*, devant un deux points, suivie par une *action*, par une *région*, par un *quasi module*, par une *quasi région*, par une *définition de procédure*, par une *définition d'entrée* ou par une *définition de processus* crée un nom dans le domaine dans lequel respectivement l' *action*, la *région*, le *quasi module*, la *quasi région*, la *définition de procédure*, la *définition de procédure* qui contient la *définition d'entrée*, la *définition de processus* apparaît.

- Une *définition* (virtuelle) introduite par une *partie-avec* ou dans un *compteur de boucle* crée un nom dans le domaine du bloc de l'*action faire* correspondante.

- Une *définition* de la *liste de définitions* d'un *tampon à choisir* ou d'un *signal à choisir* crée un nom, respectivement dans le domaine du bloc du *tampon à choisir* ou du *signal à choisir* correspondants.

- Une *définition* (virtuelle) relative à un nom prédéfini par le langage ou à un nom défini par l'implémentation crée un nom dans le domaine du processus imaginaire ou le plus externe (voir section 8.8).

Les endroits où le *nom* est employé sont appelés **occurrences d'utilisation** du *nom*. Les **règles d'identification** associent une *définition unique* à chaque occurrence d'utilisation du *nom* (voir section 10.2.2).

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 et les procédures 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.

8.2 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>* | *<spec de région>* } *
<liste d'énoncés d'action> *(4.1)*

<corps de région> ::= *(5)*
{ *<énoncé informatif>* | *<énoncé de visibilité>*} * *(5.1)*

<corps de spec de module> ::= *(6)*
{ *<quasi énoncé informatif>* | *<énoncé de visibilité>* | *<quasi module>*
| *<spec de module>* | *<quasi région>* | *<spec de région>*
| *<quasi action causer>* } * *(6.1)*

<corps de spec de région> ::= *(7)*
{ *<quasi énoncé informatif>* | *<énoncé de visibilité>* | *<quasi action causer>* }* *(7.1)*

<corps de contexte> ::= *(8)*
{ *<quasi énoncé informatif>* | *<énoncé de visibilité>* | *<quasi module>* |
<spec de module> | *<quasi région>* | *<spec de région>* } * *(8.1)*

<corps de quasi module> ::= *(9)*
{ *<quasi énoncé informatif>* | *<énoncé de visibilité>* | *<quasi module>* |
<spec de module> | *<quasi région>* | *<spec de région>* }* *(9.1)*

<quasi corps de région> ::= *(10)*
{ *<quasi énoncé informatif>* | *<énoncé de visibilité>* } * *(10.1)*

<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 statistiques: Tout domaine est immédiatement englobé comme suit par zéro, un ou plusieurs groupes:

- Si le domaine est le domaine d'une *action faire*, d'un *bloc début-fin*, d'une *définition de procédure*, d'une *définition de processus*, alors il est immédiatement englobé respectivement par le groupe dans le domaine duquel l'*action faire*, le *bloc début-fin*, la *définition de procédure*, la *définition de processus* se trouvent.

- Si le domaine est la *liste d'énoncés d'action* ou un *tampon à choisir* ou un *signal à choisir*, ou la *liste d'énoncés d'action* suivant **ELSE** dans une *action recevoir tampon et choisir* ou une *action recevoir signal et choisir*, il est immédiatement englobé par le groupe dans le domaine duquel l'*action recevoir tampon et choisir* ou *action recevoir signal et choisir*, se trouve, et seulement par ce groupe.

- Si, dans un *filet* qui ne termine **pas** un groupe, le domaine est la *liste d'énoncés d'action* d'une liste de *choix d'exceptions* ou la *liste d'énoncés d'action* qui suit **ELSE** , alors, il est directement englobé par le groupe dans le domaine duquel est placé l'énoncé terminé par le *filet*, et seulement par ce groupe.

- Si le domaine est un *choix d'exceptions* ou une *liste d'énoncés d'action* qui suit **ELSE** d'un *filet* qui termine un groupe, alors il est immédiatement englobé par le groupe terminé par le *filet*, et seulement par ce groupe.

- Si le domaine est un *module*, une *région*, une *spec de module* ou une *spec de région*, alors, il est immédiatement englobé par le groupe dans le domaine duquel il se trouve, et aussi englobé immédiatement dans le contexte qui précède immédiatement le *module*, la *région*, la *spec de module* ou la *spec de région*, s'il en existe. C'est le seul cas où un domaine a plus d'un groupe immédiatement englobant.

- Si le domaine est un *contexte*, alors, il est immédiatement englobé dans le *contexte* qui le précède immédiatement. Si un tel *contexte* n'existe pas, il n'a pas de groupe immédiatement englobant.

Un domaine a des **domaines englobant immédiatement** qui sont les domaines des groupes 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 première action, si présente, reçoit le contrôle depuis l'extérieur de la liste d'énoncés d'action.

8.3 BLOCS DÉBUT-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 8.9 et 10.2).

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 15.73 - 15.90

8.4 DÉFINITIONS DE PROCÉDURE

syntaxe:

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

<définition de procédure> ::= *(2)*
PROC ([*<liste de paramètres formels>*]) [*<spec de résultat>*]
[**EXCEPTIONS** *(<liste d'exceptions>)*] *<attributs de procédure>*;
<corps de procédure> **END** *(2.1)*

<liste de paramètres formels> ::= *(3)*
<paramètre formel> { ,*<paramètre formel>*} * *(3.1)*

<*paramètre formel*> ::= (4)
<*liste de définitions*> <*spec de paramètre*> (4.1)

<*attributs de procédure*> ::= (5)
<*généralité*>] [**RECURSIVE**] (5.1)

<*généralité*> ::= (6)
GENERAL (6.1)
| **SIMPLE** (6.2)
| **INLINE** (6.3)

<*énoncé d'entrée*> ::= (7)
<*définition* > : <*définition d'entrée*>; (7.1)

<*définition d'entrée*> ::= (8)
ENTRY (8.1)

syntaxe dérivée: Un *paramètre formel* où la *liste de définitions* 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 elle 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).

La définition devant la définition de procédure définit le nom de la procédure.

Une procédure peut avoir des points d'entrée multiples par l'emploi d'énoncés d'entrée. Ces énoncés sont considérés comme des définitions de procédure additionnelles. La définition 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** <*définition de paramètre formel* ><*mode*> :=

<*paramètre effectif*>; était rencontrée. 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 obtenu 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 (de mode éventuellement dynamique) est passé comme paramètre au corps de procédure. Seuls des locus **repérables** peuvent être passés de cette manière. Tout se passe comme si, au point d'entrée de la procédure, la déclaration de loc-identité: **DCL** <*définition de paramètre formel* ><*mode*> **LOC** [**DYNAMIC**] := <*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*, un locus contenant la valeur spécifiée sera un locus créé implicite et passé à l'endroit de l'appel. La durée de vie du locus créé est celle de l'appel de procédure. Le mode du locus créé est dynamique si la valeur a une classe dynamique.

transmission de résultat:

Une valeur ou un locus peuvent être retournés par la procédure. Dans le premier cas, une *valeur* est spécifiée toute *action résulter*; dans le dernier cas, un *locus* (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 appel de procédure rendant locus (voir section 4.2.11) ni comme un appel de procédure rendant valeur (voir section 5.2.12) 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: Une *définition* dans un *énoncé de définition de procédure* ou un *énoncé d'entrée* définit un nom de **procédure**.

Un nom de **procédure** possède une *définition de procédure* qui est définie comme suit:

- 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 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 résulter 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 9.2.1) la récursivité est **non-récursive**.

Un nom de **procédure** qui est **général**, est un nom de **procédure général**. Un nom de **procédure général** a un mode procédure qui est construit comme:
PROC *([<liste de paramètres>])* [*<spec de résultat>*]
[**EXCEPTIONS** *(<liste d'exceptions>)*] [**RECURSIVE**]
où *<spec de résultat>*, si présent, et *<liste d'exceptions>* sont les mêmes que dans sa *définition de procédure* et *<liste de paramètres>* est la séquence d'occurrences de *<spec de paramètre>* dans la *liste de paramètres formels*, séparées par des virgules.

Un nom défini dans une *liste de définitions* dans le *paramètre formel* est un nom de **locus** si et seulement si la *spec de paramètre* dans le *paramètre formel* ne contient pas l'attribut **LOC** . S'il le contient, c'est un nom de **loc-identité**. De tels noms de **locus** ou nom de **loc-identité** sont **repérables**.

conditions statiques: Si un nom de **procédure** est **intrarégional** (voir section 9.2.2), sa définition de procédure ne peut pas spécifier **GENERAL** .

Si un nom de **procédure** est une procédure **critique** (voir section 9.2.1), sa définition ne peut spécifier ni **GENERAL** ni **RECURSIVE** .

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

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

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

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

exemples:

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

8.5 DÉFINITIONS DE PROCESSUS

syntaxe:

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

<définition de processus> ::= (2)
PROCESS ([*<liste de paramètres formels>*]); *<corps de processus>* **END** (2.1)

sémantique: 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 9).

propriétés statiques: Une *définition* dans un *énoncé de définition de processus* définit un nom de **processus**.

conditions statiques: Si spécifié, le *nom de chaîne simple* doit être égal au *nom de chaîne de la définition* 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 le plus externe (voir la section 8.8).

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 non-valeur**.

exemples:

14.13

```
PROCESS ();
 wait:
  PROC (x INT );
     /*some wait action*/
  END wait;
  DO FOR EVER ;
     wait(10 /* seconds */ );
      CONTINUE operator_is_ready;
  OD ;
END                                                    (2.1)
```

8.6 MODULES

syntaxe:

```
<module> ::=                                           (1)
        [ <contexte> ] [ <définition>:]
        MODULE <corps de module> END [ <filet> ]
        [ <représentation textuelle de nom simple> ];  (1.1)
      | [ <contextes> ] <module distant>               (1.2)
```

sémantique: Un module est un énoncé d'action qui peut éventuellement contenir des déclarations et des définitions locales. Un module est un moyen de restreindre la visibilité de représentation textuelle de nom; il n'influence pas la durée de vie des locus créés localement.

Les règles de visibilité détaillées pour les modules sont données dans la section 10.2.

propriétés statiques: Une *définition* dans un *module* définit un nom de **module** ainsi qu'un nom d'**étiquette**. Ce nom a un *module* (considéré comme un **modulion** c.-à-d. en excluant les *contextes*, la *définition*, le *filet*, s'il en existe).

conditions statiques: Si spécifiée, la *représentation textuelle de nom simple* doit être égale à la représentation textuelle de *nom de la définition*.

exemples:

7.48

```
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)
```

8.7 RÉGIONS

syntaxe:

<région> ::= *(1)*

[*<contextes>*] [*<définition>* :] **REGION** *<corps de région>* **END**
[*<filet>*] [*<représentation textuelle de nom simple>*]; *(1.1)*

| [*<contextes>*] *<région distante>*; *(1.2)*

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 9). Elle détermine la visibilité des noms créés localement de la même manière qu'un module.

propriétés statiques: Une *définition* dans une *région* définit un nom de **région**. Elle est rattachée à la région (considérée comme un **modulion** c.-à-d. en excluant les *contextes*, la *définition*, le *filet*, s'il en existe).

conditions statiques: Si elle est spécifiée, la *représentation textuelle de nom simple* doit être égale à la représentation textuelle de nom de la *définition*.

Une *région* ne peut pas être englobée par un bloc autre que la *définition de processus* imaginaire le plus externe.

exemples: voir 13.1 - 13.28

8.8 PROGRAMMES

syntaxe:

<programme> ::= *(1)*

{ *<module>* | *<spec de module>* | *<région>* | *<spec de région>*} $^{+}$ *(1.1)*

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

Les définitions de noms prédéfinis par CHILL (voir Appendice C2) et les opérations prédéfinies par l'implémentation, les modes définis par l'implémentation et les noms de registre définis par l'implémentation sont considérés, pendant leur durée de vie, comme étant définis dans le domaine de la définition d'un processus imaginaire le plus externe. Pour leur visibilité, voir section 10.2.

8.9 ALLOCATION DE MÉMOIRE ET DURÉE DE VIE

Le temps durant lequel un locus ou une procédure existent dans un programme est appelé sa **durée de vie**.

Un locus est créé par une déclaration ou par l'exécution d'un appel à l'opération prédéfinie *GETSTACK* ou *ALLOCATE* .

La durée de vie d'un locus déclaré dans le domaine d'un bloc est le temps pendant lequel le contrôle est dans le bloc, sauf s'il est déclaré avec l'attribut **STATIC** . La durée de vie d'un locus déclaré dans le domaine d'un modulion est la même que s'il était déclaré dans le domaine du bloc englobant du plus près le modulion. La durée de vie d'un locus déclaré avec l'attribut **STATIC** est la même que s'il était déclaré dans le domaine de la définition de processus imaginaire le plus externe. Ceci implique que pour une déclaration de locus avec l'attribut **STATIC** , l'allocation de mémoire ne se fait qu'une fois, lorsque le processus imaginaire le plus externe démarre. Si une telle déclaration apparaît dans une définition de procédure ou une définition de processus, un locus seulement existera pour toutes les invocations ou activations.

La durée de vie d'un locus créé en exécutant l'appel d'opération prédéfinie *GETSTACK* est le temps pendant lequel le contrôle se trouve dans le bloc immédiatement englobant ou dans une procédure dont l'appel provient de ce bloc.

La durée de vie d'un locus créé par un appel d'opération prédéfinie *ALLOCATE* est le temps qui s'écoule à partir de l'appel *ALLOCATE* jusqu'au moment où aucun programme CHILL ne peut plus accéder au locus. Tel est toujours le cas si un appel d'opération prédéfinie *TERMINATE* est exécuté sur une valeur **repère** affectée repérant le locus.

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

La durée de vie d'une procédure est le bloc englobant du plus près la définition de procédure.

propriétés statiques: Un *locus* est dit **statique** si et seulement si c'est un *locus de mode statique* d'une des sortes suivantes:

- Un *nom de locus* déclaré avec l'attribut **STATIC** ou dont la définition n'est pas englobée par un bloc autre que la définition de processus imaginaire le plus externe.
- Un *élément de chaîne* ou une *tranche de chaîne* où le *locus chaîne* est **statique**, et soit l'*élément de gauche* et l'*élément de droite*, soit l'*élément de début* et la *tranche de chaîne* sont **constants**.
- Un *élément rangée* ou *tranche rangée* où le *locus rangée* est **statique** et soit l'*élément inférieur* et l'*élément supérieur*, soit le *premier élément* et la *taille de tranche* sont **constants**.
- Un *champ de structure* où le *locus structure* est **statique**. 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**.

8.10 CONSTRUCTIONS POUR LA PROGRAMMATION PAR FRAGMENTS

8.10.1 Fragments distants

syntaxe:

<*module distant*> ::= (1)
[<*représentation textuelle de nom simple*>:]
MODULE REMOTE <*indicateur de texte d'origine*>; (1.1)

<*région distante*> ::= (2)
[<*représentation textuelle de nom simple*>:]
REGION REMOTE <*indicateur de texte d'origine*>; (2.1)

<*spec de module distante*> ::= (3)
[<*représentation textuelle de nom simple*>:]
SPEC MODULE REMOTE <*indicateur de texte d'origine*>; (3.1)

<*spec de région distante*> ::= (4)
[<*représentation textuelle de nom simple*>:]
SPEC REGION REMOTE <*indicateur de texte d'origine*>; (4.1)

<*contexte distant*> ::= (5)
CONTEXT REMOTE <*indicateur de texte d'origine*> **FOR** (5.1)

<*indicateur de texte d'origine*> ::= (6)
<*littéral de chaîne de caractères*> (6.1)
| <*nom repère de texte*> (6.2)
| <*vide*> (6.3)

sémantique: Les *modules distants*, *régions distantes*, *spec de modules distants*, *spec de régions distantes* et *contextes distants* sont des moyens utilisés pour représenter le texte d'origine d'un programme sous la forme d'un ensemble de fichiers (interconnectés).

Un *indicateur de texte d'origine* se rapporte d'une manière définie par l'implémentation et comme indiqué ci-après, à un fragment de texte d'origine CHILL:

- Si l' *indicateur de texte d'origine* est vide, le texte d'origine est extrait de la structure du programme dans lequel il se trouve.

- Si l' *indicateur de texte d'origine* contient un *littéral de chaîne de caractères*, celui-ci est utilisé pour extraire le texte.

- Si l' *indicateur de texte d'origine* contient un *nom de repère de texte*, celui-ci est interprété d'une manière définie par l'implémentation pour extraire le texte d'origine.

Un programme avec *modules distants* (*régions distantes*, *spec de modules distantes*, *spec de régions distantes*, *contextes distants*) est équivalent au programme formé en remplaçant chaque *module distant* (*région distante*, *spec de module distante*, *spec de région distante*, *contexte distant*) par le fragment de texte CHILL auquel se réfère l'indicateur de texte d'origine.

conditions statiques: L' *indicateur de texte d'origine* (1. d'un *module distant*, 2. d'une *région distante*, 3. d'une *spec de module distante*, 4. d'une *spec de région distante*, 5. d'un *contexte distant*) doit se référer à un fragment du texte d'origine qui est une production terminale (1. d'un *module* qui n'est pas un *module distant*, 2. d'une *région* qui n'est pas une *région distante*, 3. d'une *spec de module* qui n'est pas une *spec de module distante*, 4. d'une *spec de région* qui n'est pas une *spec de région distante*, 5. d'un *contexte* qui n'est pas un *contexte distant*).

Lorsque le texte d'origine mentionné par l'*indicateur de texte d'origine* dans (1. un *module distant* 2. une *région distante*) commence par une *définition*, alors, (1. le *module distant*, 2. la *région distante*) doit commencer par une *représentation textuelle de nom simple* qui est la représentation textuelle du nom de cette *définition*.

Lorsque le texte d'origine mentionné par l' *indicateur de texte d'origine* dans (1. une *spec de module distante*, 2. une *spec de région distante*) commence par une *représentation textuelle de nom simple*, alors, (1. la *spec de module distante*, 2. la *spec de région distante*) doit commencer par la *représentation textuelle de nom simple*.

exemples:

25.9 **MODULE REMOTE** *stack_code* *(1.1)*
25.9 *stack_code* *(6.2)*

8.10.2 Spec de modules, spec de régions et contextes

syntaxe:

<*spec de module*> ::= *(1)*
[<*contextes*>] [<*représentation textuelle de nom simple*> :] **SPEC MODULE**
<*corps de spec de module*> **END** [<*représentation textuelle de nom simple*>]; *(1.1)*
| <*spec de module distante*> *(1.2)*

<*spec de région*> ::= *(2)*
[<*contextes*>] [<*représentation textuelle de nom simple*> :] **SPEC REGION**
<*corps de spec de région*> **END** [<*représentation textuelle de nom simple*>]; *(2.1)*
| <*spec région distante*> *(2.2)*

<*contextes*> ::= *(3)*
<*contexte*> { <*contexte*> } * *(3.1)*

<*contexte*> ::= *(4)*
CONTEXT <*corps de contexte*> **END** [<*quasi filet*>] **FOR** *(4.1)*
| <*contexte distant*> *(4.2)*

sémantique: Les spec de modules, les spec de régions et les contextes servent à spécifier des propriétés statiques de noms. Elles sont redondantes mais elles peuvent être utilisées pour la programmation par fragments.

Des représentations textuelles de nom simple apparaissant dans des spec de modules et des spec de régions ne sont pas des noms; elles n'ont ni bornes ni règles de visibilité.

conditions statiques: Dans une *spec de module* ou une *spec de région*, la *représentation textuelle de nom simple facultative* qui suit **END** ne peut être présente que si la *représentation textuelle de nom simple facultative* qui précède **SPEC** est présente. Lorsque toutes deux sont présentes, elles ont des représentations textuelles de nom égales.

Un *contexte* qui n'a pas de groupe immédiatement englobant ne peut contenir des énoncés de visibilité.

exemples:

23.1 *letter_count:*
SPEC MODULE
SEIZE *max ;*
count: **PROC** *(* **ROW** *CHAR (max)* **IN** , **ARRAY** *('A':'Z') INT* **OUT** *);* **END** *;*
GRANT *count ;*
END *letter_count;* *(1.1)*

24.1 **CONTEXT**
count: **PROC** *(* **ROW** *CHAR (max)* **IN** , **ARRAY** *('A':'Z') INT* **OUT** *);* **END** *;*
END FOR *(4.1)*

8.10.3 Quasi énoncés

syntaxe:

<quasi énoncé informatif> ::= *(1)*
<quasi énoncé déclaratif> *(1.1)*
| *<quasi énoncé définissant>* *(1.2)*

<quasi énoncé déclaratif> ::= *(2)*
DCL *<quasi déclaration>* {, *<quasi déclaration>* } *; *(2.1)*

<quasi déclaration> ::= *(3)*
<définition> <mode>
[**STATIC**] [**NONREF**] [**DYNAMIC**] *(3.1)*

<quasi énoncé définissant> ::= *(4)*
<énoncé de définition de synmode> *(4.1)*
| *<énoncé de définition de neumode>* *(4.2)*
| *<énoncé de définition de synonyme>* *(4.3)*
| *<quasi énoncé de définition de procédure>* *(4.4)*
| *<quasi énoncé de définition de processus>* *(4.5)*
| *<énoncé de définition de signal>* *(4.6)*
| *<vide>;* *(4.7)*

<quasi énoncé de définition de procédure> ::= *(5)*
<définition> : **PROC** *(* [*<quasi liste de paramètres formels>*] *)*
[*<spec de résultat>*] [**EXCEPTIONS** *(<liste d'exceptions>)*]
<attributs de procédure> { *<quasi énoncé d'entrée>* } *
END [*<représentation textuelle de nom simple>*]; *(5.1)*

<quasi énoncé d'entrée> ::= *(6)*
<définition> : **ENTRY** ; *(6.1)*

<quasi liste de paramètres formels> ::= *(7)*
<quasi paramètre formel> {, *<quasi paramètre formel>* } * *(7.1)*

<quasi paramètre formel> ::= *(8)*
[*<représentation textuelle de nom simple>*
{, *<représentation textuelle de nom simple>* } *] *<spec de paramètre>* *(8.1)*

<quasi énoncé de définition de processus> ::= *(9)*
<définition> : **PROCESS** *(* [*<quasi liste de paramètre formel>*] *)*
END [*<représentation textuelle de nom simple>*]; *(9.1)*

<quasi région> ::= *(10)*
[*<définition>* :] **REGION** *<quasi corps de région>*
END [*<représentation textuelle de nom simple>*]; *(10.1)*

<quasi module> ::= *(11)*
[*<définition>* :] **MODULE** *<quasi corps de module>*
END [*<représentation textuelle de nom simple>*]; *(11.1)*

<quasi action causer> ::= *(12)*
CAUSE *<liste d'exceptions>*; *(12.1)*

<quasi filet> ::= *(13)*
ON ELSE END
| **ON** *<liste d'exceptions>* [**ELSE**] **END** *(13.1)*

sémantique: Des quasi énoncés sont utilisés dans des spec de modules, des spec de régions et des contextes pour spécifier les propriétés statiques de noms. Ces spécifications sont redondantes mais des quasi énoncés peuvent être utilisés pour la programmation par fragments.

Des quasi actions causer indiquent la présence d'énoncés causer dans des modules distants ou des régions distantes immédiatement englobés par le domaine immédiatement englobant le domaine de la spec de module ou de la spec de région dans laquelle se trouve la quasi action causer.

Des quasi filets indiquent la présence d'un filet dans le programme, atteignable à partir de la région du module, ou d'un contexte immédiatement englobé dans le contexte terminé par le quasi filet.

propriétés statiques: Les *quasi énoncés* sont des formes limitées des *énoncés* correspondants et ont les mêmes propriétés statiques.

Le nom défini par une *définition* dans un *quasi énoncé* est **repérable** si **NONREF** n'est pas spécifié.

Une *définition* devant **REGION** dans une *quasi région* (devant **MODULE** dans un *quasi module*) définit un nom de **quasi région** (de **quasi module**) rattaché à la *quasi région* (au *quasi module*).

Toutes les *définitions* englobées par un *contexte*, une *spec de module* ou une *spec de région* sont des **quasi** *définitions*.

conditions statiques: Les quasi énoncés sont des formes limitées des énoncés correspondants et sont soumis aux mêmes conditions statiques.

8.10.4 Correspondance entre quasi définitions et définitions

Les règles suivantes s'appliquent:

- Si une *représentation textuelle de nom* dans un domaine qui n'est pas le domaine d'un *quasi module*, ou d'une *quasi région* d'une *spec de module*, d'une *spec de région* ou d'un *contexte* est rattachée à une **quasi** *définition*, elle doit aussi être rattachée à une *définition* qui n'est pas une **quasi** *définition*. Les deux *définitions* doivent avoir des propriétés statiques identiques (catégorie sémantique, repérabilité, régionalité, staticité, champs interdits, etc., selon le cas). Si les deux *définitions* ont un mode ou si leurs propriétés statiques impliquent un mode, alors, ces deux modes doivent être **semblables**. Si la **quasi** *définition* est englobée par le domaine, alors, la *définition* qui n'est pas une **quasi** *définition* doit être entourée, mais non immédiatement englobée dans ce domaine.

- Dans chaque domaine, si des *représentations textuelles de nom* N1 et N2 sont rattachées respectivement à des *définitions* D1 et D2 et des **quasi** *définitions* Q1 et Q2 et si les modes de Q1 et Q2 sont **N-semblables**, alors, les modes de D1 et D2 doivent aussi être **N-semblables**.

9 EXÉCUTION CONCURRENTE

9.1 LES PROCESSUS ET LEURS DÉFINITIONS

Un **processus** est l'exécution séquentielle d'une série d'énoncés. Il peut être exécuté en parallèle avec d'autres processus. Le comportement d'un processus est décrit par une **définition de processus** (voir section 8.5), 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 5.2.14). 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 parallèle. Chaque processus est identifié univoquement par une valeur **exemplaire**, donnée comme résultat de l'expression démarrer, ou l'évaluation de l'opérateur *THIS* . La création d'un processus cause la création de ses locus déclarés localement, sauf ceux qui sont déclarés avec l'attribut **STATIC** (voir section 8.9), et de ses valeurs et de ses procédures définies localement. Les locus déclarés localement ainsi que les valeurs et procédures sont dits avoir la même activation que le processus créé auquel ils appartiennent. Le processus imaginaire le plus externe (voir section 8.8) 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 des deux états: il est soit **actif** (c.-à-d. en exécution) ou **en attente** (c.-à-d. attendant une condition à satisfaire). La transmission d'actif à en attente est appelée la **mise en attente** du processus, la transition de en attente à actif est appelé la **réactivation** du processus.

9.2 EXCLUSION MUTUELLE ET RÉGIONS

9.2.1 Généralités

Les régions (voir section 8.7) 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 9.2.2) 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 8.4) 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 9.3). Dans le cas d'appels à des procédures critiques imbriquées dynamiquement, seule la région verrouillée le plus tard sera libérée.
- Le processus exécutant la procédure **critique** est terminé. Dans le cas d'appels de procédure **critiques** emboîtés dynamiquement, toutes les régions verrouillées par le processus seront libérées.

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

Quand une région est libérée et que plus d'un processus a été suspendu en essayant d'appeler une de ses procédures **critiques** ou d'être réactivé dans une de ses procédures **critiques**, un processus seulement sera sélectionné pour verrouiller la région suivant un algorithme de sélection défini par l'implémentation.

9.2.2 Régionalité

Pour permettre une vérification statique du fait qu'un locus déclaré dans une région ne peut être accédé qu'en appelant une procédure **critique** ou en entamant la région pour effectuer les initialisations domaniales, les conditions de contexte statiques suivantes doivent être respectées:

- les conditions de régionalité mentionnées dans les sections adéquates (action d'affectation, appel de procédure, action envoyer, action résulter);
- les procédures **régionales** ne sont pas **générales** (voir section 8.4);
- les procédures **critiques** ne sont ni **générales**, ni **récursives** (voir section 8.4).

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

1. **Locus**

   Un *locus* est **intrarégional** si et seulement si une des conditions suivantes est remplie:

   - C'est un *nom d'accès* qui est:
     - soit un *nom de locus* déclaré textuellement à l'intérieur d'une *région* et qui n'est pas défini dans un *paramètre formel* d'une procédure **critique**,
     - soit un *nom de loc-identité*, dans la déclaration duquel le *locus* est **régional** ou qui est défini dans le *paramètre formel* d'une procédure **intrarégionale**,
     - soit un *nom basé* dans la déclaration duquel le *nom de locus repère lié ou libre* est **intrarégional**,
     - soit un *nom d'énumération de locus*, dont le *locus rangée* ou le *locus chaîne* dans *l'action faire* associée est **intrarégional**,
     - soit un *nom de locus faire-avec* dont le *locus structure* dans *l'action faire* associée est **intrarégional**.
   - C'est un *repère lié dérepéré*, contenant une *expression repère lié* qui est **intrarégionale**.
   - C'est un *repère libre dérepéré*, contenant une *expression repère libre* qui est **intrarégionale**.
   - C'est un *descripteur dérepéré*, contenant une *expression descripteur* qui est **intrarégionale**.

- C'est un *élément de rangée* ou une *tranche de rangée* contenant un *locus rangée* qui est **intrarégional**.
- C'est un *élément de chaîne* ou une *tranche de chaîne* contenant un *locus chaîne* qui est **intrarégional**.
- C'est un *champ de structure* contenant un *locus structure* qui est **intrarégional**.
- C'est un *appel de procédure rendant locus*, tel que dans *l'appel de procédure rendant locus* on spécifie un *nom de procédure* qui est **intrarégional**.
- C'est un *appel d'opération prédéfinie rendant locus*, que l'implémentation spécifie être **intrarégional**.
- C'est une *conversion de locus*, contenant un *locus de mode statique* qui est **intrarégional**.

Un *locus* qui n'est pas **intrarégional** est **extrarégional**.

2. **Valeur**

Une *valeur* a une **régionalité** qui dépend de sa classe. Si elle a la M-classe par dérivation, la classe **toute** ou la classe **nulle** alors elle a une **régionalité** nulle. Sinon elle a la M-classe par valeur ou la M-classe par repère et qu'elle a une **régionalité** qui dépend du mode M, dans les conditions suivantes:

Si M n'a pas la **propriété de repérage**, alors la **régionalité** est nulle; sinon, la *valeur* est un *opérande-6* (et a la **propriété de repérage**) :

Si c'est une *valeur primitive*, alors

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

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

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

3. **Nom de procédure**

   Un *nom de procédure* est **intrarégional** si et seulement si il est défini à l'intérieur d'une *région* et qu'il n'est pas **critique** (c.-à-d. qu'il n'est pas octroyé par la région). Sinon, il est **extrarégional**.

4. **Appel de procédure**

   Un *appel de procédure* est **intrarégional** s'il contient un *nom de procédure* qui est **intrarégional**; sinon, il est **extrarégional**.

Une *valeur* est **régionalement sûre** pour un non-terminal (utilisée seulement pour *locus*, *appel de procédure* et *nom de procédure*) si et seulement si:

- le non-terminal est **extrarégional** et la *valeur* n'est pas **intrarégionale**;
- le non-terminal est **intrarégional** et la *valeur* n'est pas **extrarégionale**.

9.3 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 l'ensemble de processus mis en attente qui est attaché à chaque locus événement spécifié dans un événement à choisir de l'action mettre en attente et choisir.

Expression recevoir (voir section 5.3.8). 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 de l'ensemble de processus recevants mis en attente attaché au locus tampon (vide) spécifié.

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 l'ensemble de processus mis en attente attaché à chaque nom de signal spécifié dans un signal à choisir.

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 tampon spécifiés, et si **ELSE** n'est pas spécifié. Il devient un membre de l'ensemble de processus recevants mis en attente attaché à chaque locus tampon spécifié dans un tampon à choisir de l'action recevoir tampon et 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.

9.4 RÉACTIVATION 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 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 envoyer tampon (voir section 6.18.3). Si un processus exécute un 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).

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 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 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 l'ensemble des processus envoyants en attente et sa valeur est reçue.

Action recevoir tampon et choisir (voir section 6.19.3). Quand un processus exécute une expression 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 des 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.

9.5 ÉNONCÉ DE DÉFINITION DE SIGNAL

syntaxe:

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

<définition de signal> ::= (2)
<définition> [= (*<mode>* { ,*<mode>*} *)] [**TO** *<nom de processus>*] (2.1)

sémantique: Une définition de signal définit une fonction de composition et décomposition pour des valeurs à transmettre entre processus. Si un signal est envoyé, la liste des valeurs spécifiée est transmise. Si aucun processus n'est en attente du signal dans une action recevoir et signal et choisir, les valeurs sont gardées jusqu'à ce qu'un processus les reçoive.

propriétés statiques: Une *définition* dans un *nom de signal* définit 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 non-valeur**.

exemples:

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

10 PROPRIÉTÉS SÉMANTIQUES GÉNÉRALES

10.1 VÉRIFICATION DE MODES

10.1.1 Propriétés des modes et des classes

10.1.1.1 Propriété de protection

Informel

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

Définition

Un mode a la **propriété de protection** si et seulement si, il est:

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

10.1.1.2 Modes paramétrables

Informel

Un mode est **paramétrable** s'il peut être paramétré.

Définition

Un mode est **paramétrable** si et seulement si c'est:

- un mode structure **variable paramétrable**;
- un mode rangée;
- un mode chaîne.

10.1.1.3 Propriété de repérer

Informel

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

Définition

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

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

10.1.1.4 Propriété de marquage et de paramétrage

Informel

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

Définition

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

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

10.1.1.5 Propriété de non-valeur

Informel

Un mode a la **propriété de non-valeur** s'il n'existe pour ce mode aucune expression ni dénotation de valeur primitive.

Définition

Un mode a la **propriété de non-valeur** si et seulement si c'est:

- un mode rangée ayant un mode des **éléments** qui a la **propriété de non-valeur**;
- un mode structure dont au moins un des modes de **champ** a la **propriété de non-valeur**;
- un mode événement, un mode tampon, un mode accès ou un mode association.

10.1.1.6 Mode racine

Toute M-classe par valeur ou M-classe par dérivation où M n'est pas un mode discret ou un mode chaîne, a un mode **racine** défini de la façon suivante:

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

10.1.1.7 Classe résultante

Etant donné deux classes **compatibles** (voir section 10.1.2.7), 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 valeur et de la N-classe par valeur, de la N-classe par dérivation ou de la classe **toute**, 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 dérivation et de la N-classe par dérivation ou de la classe **toute** est la M-classe par dérivation;
- la **classe résultante** de la classe **toute** et de la classe **toute** est la classe **toute**.

Etant donnée une liste C_i de classes **compatibles** deux à deux (i=1,...,n), la **classe résultante** de la liste de classes est définie récursivement comme, si n > 1 la **classe résultante** de la **classe résultante** de la liste de classes C_i (i=1,...,n–1) et de la classe C_n , sinon la **classe résultante** de C_1 et de C_1 .

(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**.)

10.1.2 Relations entre modes et classes

10.1.2.1 Généralités

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 d'autres relations, qui sont employées principalement dans le présent chapitre dans ce but.

10.1.2.2 Relations d'équivalence sur les modes

Informel

Les relations d'équivalence suivantes jouent un rôle dans la formulation des relations de compatibilité:

- Deux modes sont **similaires** s'ils sont de la même sorte, c.-à-d. s'ils ont les mêmes propriétés héréditaires.
- Deux modes sont **v-équivalents** (équivalents en valeur) s'ils sont **similaires** et ont aussi la même **nouveauté**.
- Deux modes sont **équivalents** s'ils sont **v-équivalents** et si on prend aussi en considération les différences possibles dans la représentation des valeurs en mémoire ou dans la taille minimale de mémoire.
- Deux modes sont **l-équivalents** (équivalents en locus) s'ils sont **équivalents** et ont la même spécification de **protection**.
- Deux modes sont **semblables** s'ils sont impossibles à distinguer, c.-à-d. si toutes les opérations qui peuvent être appliquées aux objets de l'un de ces modes peuvent être appliquées à celles de l'autre, à condition de ne pas tenir compte de la **nouveauté**.
- Deux modes sont dits **N-semblables** s'ils sont **semblables** et qu'ils ont une spécification de **nouveauté** égale.

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:

- 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** et, pour chaque nom **d'élément d'ensemble** défini par un mode, il existe un nom **d'élément d'ensemble** défini par l'autre mode, qui a la même représentation textuelle et la même valeur de représentation;
- 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 modes **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 **spec de paramètre** et les **spec de paramètre** correspondantes (par position) ont des modes **l-équivalents**, les mêmes attributs de paramètre et, si présentes, les mêmes spécifications de **registres**;
    2. tous deux ont ou n'ont pas une **spec de résultat**. Si présentes, les **spec 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 même **récursivité**;
- ce sont des modes exemplaire;
- ce sont des modes événement tels qu'ils ont soit la même **longueur**, soit n'en ont pas;
- ce sont des modes tampon tels que:
    1. tous 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 association;
- ce sont des modes accès tels que:
    1. tous deux ont des modes **d'indice équivalents** ou n'en ont pas;
    2. au moins un n'a pas de mode **enregistrement**, ou tous deux ont des modes **enregistrement** qui sont **l-équivalents** et qui sont tous deux soit des modes **enregistrement statiques** soit des modes **enregistrement dynamiques**;
- ce sont des modes chaîne tels que:
    1. ce sont tous les deux des modes chaînes **de bits** ou des modes 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**;
    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**;

    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 donnera lieu à 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 doit l'être aussi;
- si l'un est un mode caractère, l'autre doit l'être 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 le mode de **protection**, l'autre doit l'avoir aussi, et:

- si les deux sont des modes repère lié, leurs modes **repérés** doivent être **l-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 ce sont des modes structure qui ne sont pas des modes structure **paramétrés**, les champs correspondants (par position) doivent être **l-équivalents**; si ce sont des modes structure **paramétrés**, leurs modes structure **variables originels** doivent être **l-équivalents**.

Les relations équivalent et l-équivalent pour les champs

Deux champs (tous les deux pris dans le contexte de deux modes structure donnés), sont 1. **équivalents**, 2. **l-équivalents** si et seulement s'ils sont tous les deux des champs fixes qui sont 1. **équivalents**, 2. **l-équivalents** ou s'ils sont tous les deux des choix de champs qui sont 1. **équivalents**, 2. **l-équivalents**.

Les relations **équivalent** et **l-équivalent** sont définies récursivement pour respectivement des champs fixes, des champs récurrents, des choix de champs et des champs à choisir correspondants, respectivement de la manière suivante:

- Champs fixes et champs récurrents

    1. Les deux champs doivent avoir des **implantations de champ équivalentes**.

    2. Les modes de deux **champs** doivent être 1. **équivalents**, 2. **l-équivalents**.

- Choix de champs

    1. Les deux choix de champs ont des marqueurs ou les deux n'en ont pas. Dans le premier cas, les marqueurs doivent avoir le même nombre de noms de **champs marqueurs** et les noms de

champs marqueurs correspondants (par position) doivent dénoter des champs fixes correspondants.

2. Les deux doivent avoir le même nombre de champs à choisir et les champs à choisir correspondants (par position) doivent être 1. **équivalents**, 2. **l-équivalents**.

3. Les deux doivent ne pas avoir de spécification de **ELSE** ou les deux doivent l'avoir. Dans le deuxième cas, le même nombre de champs récurrents doit suivre et les champs récurrents correspondants (par position) doivent être 1. **équivalents**, 2. **l-équivalents**.

- Champs à choisir

1. Les deux champs à choisir doivent avoir le même nombre de listes d'étiquettes de cas et les listes d'étiquettes de cas correspondantes (par position) doivent être toutes les deux *indifférent*, ou toutes les deux définir le même ensemble de valeurs. Si une liste d'étiquettes de cas contient un **ELSE** , alors, l'autre doit aussi le contenir.

2. 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. **l-équivalents**.

La relation équivalent pour les implantations

Dans la suite, on admettra 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>)*

La section 3.11.6 donne les règles appropriées pour donner à *pos* ou *pas* la forme désirée.

- Implantation de champ

  Deux **implantations de champ** sont **équivalentes** si elles sont toutes les deux **NOPACK** , ou toutes les deux **PACK** , ou toutes les deux *pos*. Dans le dernier cas, le premier *pos* doit être **équivalent** au second (voir plus loin).

- Implantation d'élément

  Deux **implantations d'élément** sont **équivalentes** si elles sont toutes les deux **NOPACK** , ou toutes les deux **PACK** , ou toutes les deux *pas*. Dans ce dernier cas, le *pos* dans le premier *pas* doit être **équivalent** au *pos* dans le second *pas* (voir plus loin) et la *taille de pas* doit donner la même valeur pour les deux **implantations d'élément**.

- Pos

  Un *pos* est équivalent à un autre *pos* si et seulement si les deux occurrences de *mot* donnent la même valeur, les deux occurrences de *bit initial* donnent la même valeur, et les deux occurrences de *longueur* donnent la même valeur.

La relation semblable

Deux modes sont **semblables** si et seulement si tous deux sont ou ne sont pas des modes **protégés** et s'ils ont tous deux la **nouveauté nulle** ou si tous deux ont la **nouveauté** non-**nulle** et que:

- ce sont des modes entier;
- ce sont des modes booléen;
- ce sont des modes caractère;
- ce sont des modes ensemble **similaires**;
- ce sont des modes intervalle ayant des **bornes supérieures** égales et des **bornes inférieures** égales;
- ce sont des modes ensembliste tels que leurs modes **primitifs** sont **semblables**;

- ce sont des modes repère lié tels que leurs modes **repère** sont **semblables**;
- ce sont des modes repère libre;
- ce sont des modes descripteur tels que leurs modes **repérés originels** sont **semblables**;
- ce sont des modes procédure tels que:
    1. ils ont le même nombre de **spec de paramètre** et les **spec de paramètre** correspondantes (par position) ont des modes **semblables**, les mêmes attributs de paramètre et le même nom de **registre**, s'il existe;
    2. ils ont ou n'ont pas de **spec de résultat**. Si présentes, les **spec de résultat** doivent avoir des modes **semblables**, les mêmes attributs et, si présents, les mêmes **registres**;
    3. ils ont le même ensemble de noms **d'exception**;
    4. ils ont la même **récursivité**;
- ce sont des modes exemplaire;
- ce sont des modes événement tels que tous deux ont la même **longueur** d'événement ou n'en ont pas;
- ce sont des modes tampon tels que:
    1. tous deux ont la même **longueur tampon** ou n'en ont pas;
    2. ils ont des modes **des éléments de tampon** qui sont **semblables**;
- ce sont des modes association;
- ce sont des modes accès tels que:
    1. tous deux ont des modes **d'indice semblables** ou n'en ont pas;
    2. un au moins n'a pas de mode **enregistrement** ou tous deux ont des modes **enregistrement** qui sont **semblables** et qui sont tous deux des modes **d'enregistrement statiques** ou des modes **d'enregistrement dynamiques**;
- ce sont des modes chaîne tels que:
    1. tous deux sont des modes chaîne **de bits** ou des modes chaîne **de caractères**;
    2. ils ont la même **longueur de chaîne**;
- ce sont des modes rangée tels que:
    1. leurs modes **d'indice** sont **semblables**;
    2. leurs modes des **éléments** sont **semblables**;
    3. leurs **implantations des éléments** sont **équivalentes**;
    4. ils ont le même **nombre d'éléments**;
- ce sont des modes structure qui ne sont pas des modes structure **paramétrés** tels que:
    1. ils ont le même nombre de champs et les champs correspondants (par position) sont **semblables**;
    2. s'ils sont tous deux des modes structure **paramétrés variables**, leurs listes de classes doivent être **compatibles**;
- ce sont des modes structure **paramétrés** tels que:
    1. leurs modes structure **variables originels** sont **semblables**;

2. leurs valeurs correspondantes (par position) sont les mêmes.

Les relations égales pour des champs

Deux champs (tous deux dans le contexte de deux modes structure donnés) sont **semblables** si et seulement si, ils sont tous deux des champs fixes qui sont **semblables** ou tous deux des champs à choisir qui sont **semblables**.

La relation **semblable** est définie récursivement pour des champs fixes, des champs récurrents, des choix de champs et des champs à choisir (correspondants), respectivement de la manière suivante:

- champs fixes et champs récurrents

  1. Les deux champs doivent avoir une **implantation de champ équivalente**.
  2. Les deux modes de **champs** doivent être **semblables**.
  3. Les deux champs doivent avoir la même représentation textuelle de noms.

- choix de champs

  1. Les choix de champs doivent avoir tous deux des marqueurs ou ne pas en avoir. Dans le premier cas, les marqueurs doivent avoir le même nombre de noms de **champ marqueur** et les noms de **champ marqueur** correspondants (par position) doivent dénoter des champs fixes correspondants.
  2. Tous deux doivent avoir le même nombre de champs à choisir et les champs à choisir correspondants (par position) doivent être **semblables**.
  3. Tous deux peuvent ne pas avoir de spécification **ELSE** ou tous deux doivent avoir la spécification **ELSE** . Dans ce dernier cas, le même nombre de champs récurrents doit suivre et les champs récurrents correspondants (par position) doivent être **semblables**.

- champs à choisir

  1. Les deux champs à choisir doivent avoir le même nombre de listes d'étiquettes de cas et les listes d'étiquettes de cas correspondantes (par position) doivent soit être toutes deux *indifférentes* soit définir toutes deux le même ensemble de valeurs. Si une liste d'étiquettes de cas contient un **ELSE** , l'autre doit aussi en contenir un.
  2. Deux champs à choisir doivent avoir le même nombre de champs récurrents et des champs récurrents correspondants (par position) doivent être **semblables**.

La relation "N-semblable"

Deux modes qui sont **semblables** sont **N-semblables** si et seulement si, ils ont la même **nouveauté** et, si cette **nouveauté** est **nulle**, alors:

- si ce sont des modes ensembliste, leurs modes **primitifs** doivent être **N-semblables**;
- si ce sont des modes rangée, leurs modes **d'indice** respectifs et leurs modes des **éléments** doivent être **N-semblables**;
- si ce sont des modes structure, leurs modes de **champ** correspondants (par position) doivent être **N-semblables**;
- si ce sont des modes repère, leurs modes **repérés** doivent être **N-semblables**;
- si ce sont des modes descripteur, leurs modes **repérés originels** doivent être **N-semblables**;
- si ce sont des modes procédure, leur **spec de résultat**, si présente, et les **spec de paramètre** correspondantes (par position), si présentes, doivent avoir des modes **N-semblables**;
- si ce sont des modes tampon, leurs modes des **éléments tampon** doivent être **N-semblables**;

- si ce sont des modes accès, leurs modes **d'indice**, si présents, doivent être **N-semblables** et leurs modes **enregistrement**, si présents, doivent être **N-semblables**.

10.1.2.3 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 éventuels sont égaux ou ont des spécifications de **protection** plus restrictives. De ce fait, la relation est asymétrique.

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 modes rangée, le mode des **éléments** de M doit être **compatible en lecture** avec le mode des **éléments** de N;
- si M et N sont des modes structure qui ne sont pas des modes structure **paramétrés**, tout mode de **champ** de M doit être **compatible en lecture** avec le mode de **champ** correspondant de N. Si M et N sont des modes structure **paramétrés**, le mode structure **variable originel** de M doit être **compatible en lecture** avec le mode structure **variable originel** de N.

10.1.2.4 La relation compatible en lecture dynamique

Informel

La relation **compatible en lecture dynamique** n'intéresse que les modes qui peuvent être dynamiques, c.-à-d. des modes chaîne, rangée, et structure **variable**. Un mode **paramétrable** M est dit être **compatible en lecture dynamique** avec un mode N (éventuellement dynamique) s'il existe une version dynamiquement paramétrée de M qui est **compatible en lecture** dynamique avec N.

Définition

Un mode M est **compatible en lecture dynamique** avec un mode N (une relation asymétrique) si et seulement si l'une des conditions ci-après se vérifie:

- M et N sont des modes chaîne et il existe une longueur p telle que M(p) est **compatible en lecture** avec N. Cette vérification est dynamique si N est dynamique. Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL* .
- M et N sont des modes rangée et il existe une valeur p telle que M(p) est **compatible en lecture** avec N. Cette vérification est dynamique si N est dynamique. Une détection d'anomalie donnera lieu à l'exception *RANGEFAIL* .
- M est un mode structure **paramétrable variable** et N est un mode structure **paramétré** et il existe une liste de valeurs p_1 , ..., p_n telle que M(p_1 , ..., p_n) est **compatible en lecture** avec N. Cette vérification est dynamique si N est dynamique. Une détection d'anomalie donnera lieu à l'exception *TAGFAIL* .
- M et N sont des modes structure **paramétrables variables** et M est **compatible en lecture** avec N.

10.1.2.5 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 au moins restrictives que ceux repérés par N. De ce fait cette relation est asymétrique. La relation est employée pour les affectations.

Exemple:
REF *INT* est **limitable à REF READ** *INT*
STRUCT *(Q* **REF** *BOOL)* est **limitable à STRUCT** *(Q* **REF READ** *BOOL)*

Définition

Un mode M est **limitable à** un mode N (une relation asymétrique) si et seulement si M et N sont **équivalents** et que, de plus:

- si M et N sont des modes repère lié, le mode **repéré** de N doit être **compatible en lecture** avec le mode **repéré** de M;
- si M et N sont des modes descripteur, le mode **repéré originel** de N doit être **compatible en lecture** avec le mode **repéré originel** de M;
- si M et N sont des modes rangée, le mode des **éléments** de M doit être **limitable au** mode des **éléments** de N;
- si M et N sont des modes structure, chaque mode de **champ** de M doit être **limitable au** mode de **champ** correspondant de N.

10.1.2.6 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 satisfaite:
  1. N est un mode statique et M est un mode repère lié dont le mode **repéré** est **compatible en lecture** avec N;
  2. N est un mode statique et M est un mode repère libre;
  3. M est un mode 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. La valeur de p ne doit pas être supérieure à la **longueur de chaîne** de V. Cette vérification est dynamique si N a un mode dynamique. Une détection d'anomalie causera l'exception *RANGEFAIL* ;
     - 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** (éventuellement dynamique) de N. La valeur de p ne doit pas être supérieure à la **borne supérieure** de V. Cette vérification est dynamique si N est un mode dynamique. Une détection d'anomalie causera une exception *RANGEFAIL* ;
     - si V est un mode structure **variable**, M doit être un mode structure **paramétré** tel que V(p_1 ,...p_n) est **équivalent** à M, où p_1 ,...p_n dénote la liste de valeurs de M;
- 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.

10.1.2.7 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 originel** 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. La valeur de p ne doit pas être supérieure à la **longueur de chaîne** de V. Cette vérification est dynamique si M a un mode dynamique. Une détection d'anomalie causera l'exception *RANGEFAIL* .

        - 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. La valeur de p ne doit pas être supérieure à la **borne supérieure** de V. Cette vérification est dynamique si M est un mode dynamique. Une détection d'anomalie causera l'exception *RANGEFAIL* .

        - si V est un mode structure **variable**, M doit être un mode structure **paramétré** tel que V(p_1 ,...p_n) est **équivalent** à M, où p_1 ,...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**.

10.1.3 Sélection de cas

syntaxe:

<spécification d'étiquettes de cas> ::= *(1)*
<liste d'étiquettes de cas> { ,*<liste d'étiquettes de cas>*} * *(1.1)*

<liste d'étiquettes de cas> ::= *(2)*
(*<étiquette de cas>* { ,*<étiquette de cas>*} *) *(2.1)*
| *<indifférent>* *(2.2)*

<étiquette de cas> ::= *(3)*

< expression littérale discrète> *(3.1)*

| *<intervalle littéral>* *(3.2)*

| *< nom de mode discret>* *(3.3)*

| **ELSE** *(3.4)*

<indifférent> ::= *(4)*

(*) *(4.1)*

sémantique: La sélection de cas est un moyen de sélectionner une alternative d'une liste d'alternatives. La sélection se base sur la spécification d'une liste de valeurs de sélecteurs. La sélection de cas s'applique à:

- des choix de champs (voir section 3.11.4), 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 une seule 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, ou indiquées implicitement dans le cas de **ELSE** ;
- la liste d'étiquettes de cas consiste en *indifférent*.

Les valeurs indiquées **explicitement** par une étiquette de cas sont les valeurs de toute *expression 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*; si c'est **ELSE** , la classe **toute**.
- A une *liste d'étiquettes de cas* on attache une classe qui est, s'il s'agit de *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**. 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 de 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 3.11.4).

- 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, c'est l'ensemble des valeurs défini par M, où la classe du sélecteur correspondant est la M-classe par valeur ou la M-classe par dérivation.

conditions statiques: Pour chaque *spécification d'étiquettes de cas*, le nombre d'occurrences de *liste d'étiquettes de cas* doit être égal.

Pour tout couple de *spécification d'étiquettes de cas* leurs listes de classes doivent être **compatibles**.

La liste d'occurrences de *spécification d'étiquettes de cas* doit être **cohérente**, c.-à-d. que chaque liste de valeurs des sélecteurs possible ne correspond qu'à une spécification d'étiquettes de cas.

exemples:

11.9	*(occupied)*	*(3.1)*
11.58	*(rook),(*)*	*(1.1)*
8.25	*(**ELSE**)*	*(2.2)*

10.1.4 Définition et résumé des catégories sémantiques

Cette section donne un résumé de toutes les catégories sémantiques qui sont indiquées dans la description syntaxique au moyen d'une partie soulignée. Si ces catégories ne sont pas définies dans la section appropriée, la définition est donnée ici, sinon la section appropriée est référencée.

10.1.4.1 Noms

Noms de mode

nom de mode:	voir section 3.2.1.
nom de mode accès:	un *nom* défini par un mode accès.
nom de mode association:	un *nom* défini par un mode association.
nom de mode booléen:	un *nom* défini par un mode booléen.
nom de mode caractère:	un *nom* défini par un mode caractère.
nom de mode chaîne:	un *nom* défini par un mode chaîne.
nom de mode chaîne paramétré:	un *nom* défini par un mode chaîne paramétré.
nom de mode descripteur:	un *nom* défini par un mode descripteur.
nom de mode discret:	un *nom* défini par un mode discret.
nom de mode 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 une 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 section 4.1.3
nom de locus: voir section 4.1.2
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'énumération de valeur: voir section 6.5.2
nom de synonyme: voir section 5.1
nom de valeur faire-avec: voir section 6.5.4
nom de valeur reçue: voir sections 6.19.2, 6.19.3

Noms divers

liste de représentations textuelles de noms simples réservées: une *liste de représentations textuelles de noms simples* ne contenant que des représentations textuelles de noms simples **réservées** (voir Appendice C1).
nom de champ: voir section 3.10.4
nom de champ marqueur: voir section 3.11.4
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 8.6
nom de procédure: voir section 8.4
nom de procédure générale: un *nom de procédure* dont la généralité est un nom de procedure **générale**.
nom de processus: voir section 8.5
nom de région: voir section 8.7
nom de registre: un *nom* défini par l'implémentation dénotant un registre de la machine.
nom de signal: voir section 9.5
nom de synonyme indéfini: voir section 5.1
nom d'étiquette: voir sections 6.1, 8.6
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.

10.1.4.2 Locus

locus accès: un *locus* qui a un mode accès.
locus association: un *locus* qui a un mode association.
locus chaîne: un *locus* qui a un mode chaîne.
locus discret: un *locus* qui a un mode discret.

locus événement:	un *locus* qui a un mode événement.
locus exemplaire:	un *locus* qui a un mode exemplaire.
locus statique:	un *locus* qui a un mode statique.
locus rangée:	un *locus* qui a un mode rangée.
locus structure:	un *locus* qui a un mode structure.
locus tampon:	un *locus* qui a un mode tampon.

10.1.4.3 Expressions et valeurs

expression booléenne:	une *expression* dont la classe est **compatible** avec un mode booléen.
valeur primitive chaîne:	une *valeur primitive* dont la classe est **compatible** avec un mode chaîne.
valeur constante:	une *valeur* qui est constante.
valeur primitive descripteur:	une *valeur primitive* dont la classe est **compatible** avec un mode descripteur.
expression discrète:	une *expression* dont la classe est **compatible** avec un mode discret.
expression ensembliste:	une *expression* dont la classe est **compatible** avec un mode ensembliste.
expression entière:	une *expression* dont la classe est **compatible** avec un mode entier.
valeur primitive exemplaire:	une *valeur primitive* 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**.
valeur primitive procédure:	une *valeur primitive* dont la classe est **compatible** avec un mode procédure.
valeur primitive rangée:	une *valeur primitive* dont la classe est **compatible** avec un mode rangée.
valeur primitive repère:	une *valeur primitive* dont la classe est **compatible** avec un mode repère lié, un mode repère libre ou un mode descripteur.
valeur primitive repère libre:	une *valeur primitive* dont la classe est **compatible** avec un mode repère libre.
valeur primitive repère lié:	une *valeur primitive* dont la classe est **compatible** avec un mode repère lié.
valeur primitive structure:	une *valeur primitive* 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.

10.1.4.4 Appels d'opération prédéfinie

appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association	voir section 7.4.8
appel d'opération prédéfinie d'e/s rendant locus de CHILL associer	voir section 7.4.2
appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association	voir section 7.4.4
appel d'opération prédéfinie d'e/s simple de CHILL connecter	voir section 7.4.6
appel d'opération prédéfinie d'e/s simple de CHILL déconnecter	voir section 7.4.7
appel d'opération prédéfinie d'e/s simple de CHILL désassocier	voir section 7.4.3
appel d'opération prédéfinie par l'implémentation	voir section 6.7
appel d'opération prédéfinie par l'implémentation rendant locus	voir section 4.2.12
appel d'opération prédéfinie par l'implémentation rendant valeur	voir section 5.2.13
appel d'opération prédéfinied'e/s rendant locus de CHILL associer	voir section 7.4.2
appel d'opération prédéfinie d'e/s simple de CHILL modification	voir section 7.4.5

appel d'opération prédéfinie d'e/s rendant valeur de CHILL lire article voir section 7.4.9
appel d'opération prédéfinie d'e/s simple de CHILL écrire article voir section 7.4.9

10.1.4.5 Catégories sémantiques diverses

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

mode discret: un *mode* dans lequel le *mode simple* est un mode discret.

appel de procédure rendant locus : voir section 4.2.11

définition de modulion : une *définition* qui définit un *nom de module* ou un *nom de région*.

caractère différent d'apostrophe : un *caractère* qui n'est pas une apostrophe.

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

appel de procédure rendant valeur: voir section 5.2.12

10.2 VISIBILITÉ ET IDENTIFICATION

10.2.1 Degrés de visibilité

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

Tableau 1. Degrés de visibilité

Visibilité	**Propriétés** (informel)
Directement fortement visible	La représentation textuelle de nom est visible par création, octroi ou envahissement direct
Indirectement fortement visible	La représentation textuelle de nom est héritée via une imbrication de bloc ou son attribut d'envahissement
Faiblement visible	La représentation textuelle de nom est impliquée par un nom fortement visible
Invisible	La représentation textuelle de nom ne peut pas être utilisée

Une *représentation textuelle de nom* est dite être **fortement visible** dans un domaine si elle est soit **directement fortement visible** soit **indirectement fortement visible** dans ce domaine. Une *représentation textuelle de nom* est dite être **visible** si elle est soit **faiblement**, soit **fortement visible** dans ce domaine. Autrement, le *nom* est dit être **invisible** dans ce domaine. Les énoncés de structuration du programme et les énoncés de visibilité déterminent d'une façon univoque à quelle classe de visibilité chaque *représentation textuelle de nom* appartient.

Lorsqu'une *représentation textuelle de nom* est visible dans un domaine, elle peut être **liée directement** à une autre *représentation textuelle de nom* dans un autre domaine, ou **liée directement** à une *définition* dans le programme. Les règles des liaisons directes sont énoncées dans la section 10.2.4.

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

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

- N1 dans R1 est directement lié à N2 en R2 ou à D,
- N1 dans R1 est directement lié à un N dans un R, et, dans R, N est lié à N2 en R2, ou à D.

Il est à noter que la liaison doit être interprétée par rapport au degré de visibilité: une *représentation textuelle de nom* peut être fortement visible dans un domaine avec certaines liaisons, et faiblement visibles dans le même domaine avec d'autres liaisons.

10.2.2 Conditions de visibilité et identification

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

- Chaque *représentation textuelle de nom* visible dans ce domaine doit être liée à au moins une *définition*.
- Si une *représentation textuelle de nom* est fortement visible dans un domaine, et qu'elle est liée à plus d'une *définition*, alors, toutes ces *définitions* (qui ne sont pas des quasi *définitions*, auxquelles sont applicables les règles de la section 8.10) doivent être des *définitions* de classes **compatibles** (en d'autres termes: doivent définir le même élément d'ensemble) et toutes ces définitions doivent être directement englobées dans un même et unique domaine.

Une *représentation textuelle de nom* faiblement visible dans un domaine et liée comme une *représentation textuelle de nom* faiblement visible dans ce domaine à des *définitions* qui ne sont pas dans des classes **compatibles** est dite avoir une faible discordance dans ce domaine.

Une *représentation textuelle de nom* NS, visible dans un domaine R, est dite être liée dans R à plusieurs *définitions* conformément aux règles suivantes:

- Si NS est fortement visible dans R, NS est lié aux *définitions* auxquelles il est lié en R (comme une *représentation textuelle de nom* fortement visible).
- Autrement, si NS est faiblement visible en R, il est lié aux *définitions* auxquelles il est lié en R (comme une représentation textuelle de nom faiblement visible), à condition que NS n'ait pas de faible discordance en R. (De faibles discordances sont autorisées dans un domaine s'il n'existe pas dans le domaine de *nom* avec une *représentation textuelle de nom* ayant une faible discordance).
- Autrement, NS n'est pas lié en R.

condition statique: La *représentation textuelle de nom* attachée à chaque nom directement englobé dans un domaine doit être visible et liée à ce domaine.

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

10.2.3 Représentations textuelles de nom impliquées

Chaque *représentation textuelle de nom* fortement visible dans un domaine R a un ensemble de *représentations textuelles de nom impliquées*, qui sont faiblement visibles en R, avec les liaisons spécifiées ci-après.

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

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

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

- Si D définit un nom de **procédure**, les *définitions* impliquées de NS en R sont les définitions impliquées en R par la *liste de paramètres* et celle de la *spec de résultat* de la procédure, si elle existe.
- Si D définit un nom de **signal**, les *définitions* impliquées de NS en R sont toutes les *définitions* impliquées dans R par tous les modes attachés au signal.
- Si D définit un nom de **processus**, les *définitions* impliquées de NS en R sont les définitions impliquées en R par la *liste de paramètres*, si elle existe.

Tableau 2. Définitions impliquées de modes dans le domaine R

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

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

- NS est une *représentation textuelle de nom simple*. Alors Ni, directement lié en R à D, est une *représentation textuelle de nom* impliquée de NS.
- NS a la forme P ! S, où S une *représentation textuelle de nom simple*. Alors, P ! Ni directement lié en R à D est une *représentation textuelle de nom* impliquée de NS.

exemples:

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

10.2.4 Visibilité dans les domaines

10.2.4.1 Généralités

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

- cette *représentation textuelle de nom* est saisie dans le domaine (voir 10.2.4.5);
- cette *représentation textuelle de nom* est octroyée dans le domaine (voir 10.2.4.4);
- il existe une *définition* ayant cette *représentation textuelle de nom* dans le domaine. Dans ce cas, la *représentation textuelle de nom* dans le domaine est directement liée à la *définition*. (A noter que la *représentation textuelle de nom* peut être directement liée à plusieurs *définitions* dans le domaine.)
- cette *représentation textuelle de nom* est fortement visible dans un domaine immédiatement englobant et a la propriété d'envahissement dans ce domaine. Dans ce cas, la *représentation textuelle de nom* a aussi la propriété d'envahissement direct dans le domaine, et est directement liée à la même *représentation textuelle de nom* dans le domaine immédiatement englobant.

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

- le domaine est un bloc dans lequel la *représentation textuelle de nom* n'est pas directement fortement visible, et la *représentation textuelle de nom* est fortement visible dans le domaine immédiatement englobant. La *représentation textuelle de nom* est dite être héritée par le bloc et est directement liée à la même *représentation textuelle de nom* dans le domaine immédiatement englobant.
- la *représentation textuelle de nom* n'est pas directement fortement visible dans le domaine, et la *représentation textuelle de nom* est fortement visible dans un domaine immédiatement englobant, où elle a la propriété d'envahissement. La *représentation textuelle de nom* dans le domaine est directement liée à la *représentation textuelle de nom* dans le domaine immédiatement englobant. La *représentation textuelle de nom* a la propriété d'envahissement dans le domaine.
- la *représentation textuelle de nom* est une *représentation textuelle de nom* définie par le langage ou par l'implémentation, et le domaine est un *contexte* qui n'est pas précédé par un *contexte* ou la *définition de processus* imaginaire la plus externe. La *représentation textuelle de nom* est considérée comme étant directement liée à une *définition* pour sa signification prédéfinie. La *représentation textuelle de nom* a la propriété d'envahissement.

Une *représentation textuelle de nom* est faiblement visible dans un domaine si elle est impliquée par une *représentation textuelle de nom* qui est fortement visible dans le domaine; les règles de liaison sont définies en 10.2.3.

10.2.4.2 Enoncés de visibilité

syntaxe:

```
<énoncé de visibilité> ::=                    (1)
        <énoncé d'octroi>                     (1.1)
      | <énoncé de saisie>                    (1.2)
```

sémantique: Les énoncés de visibilité ne sont autorisés que dans le domaine des modulions et contrôlent la visibilité des représentations textuelles de nom qui y sont mentionnées et, implicitement, de leurs représentations textuelles de nom impliquées.

propriétés statiques: Un *énoncé de visibilité* a un ou deux domaines originels (voir section 8.2) et un ou deux domaines de destination, définis comme suit:

- Si l'*énoncé de visibilité* est un *énoncé de saisie*, son domaine de destination est le domaine du modulion immédiatement englobant l'*énoncé de saisie*, et ses domaines originels sont les domaines directement englobant ce domaine de modulion.

- Si l'*énoncé de visibilité* est un *énoncé d'octroi*, alors, son domaine originel est le domaine de modulion immédiatement englobant l'*énoncé d'octroi*, et ses domaines de destination sont les domaines immédiatement englobant ce domaine de modulion.

10.2.4.3 Clause renommer préfixe

syntaxe:

<clause renommer préfixe> ::= *(1)*
(<ancien préfixe> -> <nouveau préfixe>) ! <postfixe> *(1.1)*

<ancien préfixe> ::= *(2)*
<préfixe> *(2.1)*
| *<vide>* *(2.2)*

<nouveau préfixe> ::= *(3)*
<préfixe> *(3.1)*
| *<vide>* *(3.2)*

<postfixe> ::= *(4)*
<postfixe de saisie> {,*<postfixe de saisie>*}* *(4.1)*
| *<postfixe d'octroi>* {,*<postfixe d'octroi>*}* *(4.2)*

syntaxe dérivée: Une *clause renommer préfixe* dans laquelle le *postfixe* consiste en plus d'un *postfixe de saisie* (*postfixe d'octroi*) est la syntaxe dérivée de plusieurs *clauses renommer préfixe*, une pour chaque *postfixe de saisie* (*postfixe d'octroi*), séparées par des virgules, avec le même *ancien préfixe* et le même *nouveau préfixe*.

Par exemple :

GRANT *(p -> q) ! a , b ;*

est la syntaxe dérivée de

GRANT *(p -> q) ! a , (p -> q) ! b ;*

sémantique: Les clauses renommer préfixe sont utilisées dans des énoncés de visibilité pour exprimer le changement de préfixe dans des représentations textuelles de nom préfixées qui sont octroyées ou saisies. (Etant donné que les clauses renommer préfixe peuvent être utilisées sans changement de préfixe – lorsque l'ancien et le nouveau préfixes sont vides – elles sont prises pour base sémantique des énoncés de visibilité.)

propriétés statiques: Une *clause renommer préfixe* a un ou deux domaines originels, qui sont les domaines originels de l'*énoncé de visibilité* dans lequel elle est écrite.

Une *clause renommer préfixe* a un ou deux domaines de destination, qui sont les domaines de destination de l'*énoncé de visibilité* dans lequel elle est écrite.

Un *postfixe* a un ensemble de *représentations textuelles de nom* qui est l'ensemble de *représentation textuelle de nom* attaché à son *postfixe de saisie* ou à l'ensemble de représentation textuelle de nom attaché à son *postfixe d'octroi*. Ces *représentations textuelles de nom* sont les *représentations textuelles de nom* de postfixe de la *clause renommer préfixe*.

Une *clause renommer préfixe* a un ensemble d'anciennes *représentations textuelles de nom* et un ensemble de nouvelles *représentations textuelles de nom*. Chaque *représentation textuelle de nom* de postfixe attachée à une *clause renommer préfixe* donne à la fois une ancienne *représentation textuelle de nom* et une nouvelle *représentation textuelle de nom* attachées à cette *clause*, comme

suit: on obtient la nouvelle *représentation textuelle de nom* en préfixant la *représentation textuelle de nom* de postfixe avec le *nouveau préfixe*; on obtient l'ancienne *représentation textuelle de nom* en préfixant la représentation textuelle de nom de postfixe avec l'*ancien préfixe*.

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

règles de visibilité: Les nouvelles *représentations textuelles de nom* attachées à une *clause renommer préfixe* sont fortement visibles dans leurs domaines de destination et sont liées dans ces domaines à leurs sources dans les domaines originels. Si la *clause renommer préfixe* fait partie d'un *énoncé de saisie (d'octroi)*, ces *représentations textuelles de nom* sont saisies (octroyées) dans leurs domaines de destination.

Une *représentation textuelle de nom* NS fortement visible dans le domaine R est dite être saisissable par le modulion M immédiatement englobé dans R si NS n'est pas lié en R à une quelconque *représentation textuelle de nom* dans le domaine de M.

Une *représentation textuelle de nom* NS faiblement visible dans le domaine R est dite être saisissable par le modulion M immédiatement englobé dans R si NS est lié en R à une *définition* non englobée par le domaine de M.

Une *représentation textuelle de nom* NS fortement visible dans le domaine R du modulion M est dite être octroyable par M si NS n'est pas lié en R à NS dans le domaine immédiatement englobant M.

Une *représentation textuelle de nom* NS faiblement visible dans le domaine R du modulion M est dite être octroyable par M si NS est lié en R à une *définition* entourée par R.

conditions statiques: Si une *clause renommer préfixe* est dans un *énoncé de saisie* immédiatement englobé dans le domaine du modulion M, alors, chacune de ses anciennes *représentations textuelles de nom* doit être:

- visible et liée dans le domaine immédiatement englobant le domaine de M et
- saisissable par M.

Si une *clause renommer préfixe* est dans un *énoncé d'octroi* immédiatement englobé dans le domaine du modulion M, alors chacune de ses anciennes *représentations textuelles de nom* doit être:

- visible et liée dans le domaine de M, et
- octroyable par M.

Si un domaine de modulion contient plusieurs *clauses renommer préfixe* dans un (des) *énoncé(s) de saisie (d'octroi)*, alors leurs ensembles de nouvelles *représentations textuelles de nom* doivent être disjointes deux par deux.

exemples:

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

10.2.4.4 Enoncé d'octroi

syntaxe:

<énoncé d'octroi> ::= *(1)*

GRANT *<clause renommer préfixe>* {,*<clause renommer préfixe>*}*
[[**DIRECTLY**] **PERVASIVE**] ; *(1.1)*

| **GRANT** *<fenêtre d'octroi>* [*<clause préfixe>*]
[[**DIRECTLY**] **PERVASIVE**] ; *(1.2)*

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

<postfixe d'octroi> ::= (3)
<représentation textuelle de nom> (3.1)
| *<représentation textuelle de nom de neumode >* *<clause d'interdiction >* (3.2)
| [*<préfixe>* !] **ALL** (3.3)

<clause préfixe> ::= (4)
PREFIXED [*<préfixe>*] (4.1)

<clause d'interdiction> ::= (5)
FORBID { *<liste de noms d'interdiction>* | **ALL** } (5.1)

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

sémantique: Les énoncés d'octroi sont un moyen d'étendre aux domaines directement englobants la visibilité de représentation textuelle de nom d'un domaine de modulion. **FORBID** ne peut être spécifié que pour des noms de neumode qui sont des modes structure. Cela signifie que tous les locus et toutes les valeurs de ce mode ont des champs qui ne peuvent être choisis qu'à l'intérieur du modulion d'octroi, et non à l'extérieur.

Les règles de visibilité suivantes sont applicables:

- Si l'*énoncé d'octroi* contient une (des) *clause(s) renommer préfixe*, l'*énoncé d'octroi* a l'effet de sa (ses) *clause(s) renommer préfixe* (voir section 10.2.4.3).
- Si l'*énoncé d'octroi* contient des *fenêtres d'octroi*, c'est la notation abrégée pour un ensemble d'*énoncés d'octroi* avec *clause renommer préfixe* formée comme suit:
  - Il existe un *énoncé d'octroi* pour chaque *postfixe d'octroi* dans la *fenêtre d'octroi.*
  - L'*ancien préfixe* dans leur *clause renommer préfixe* est vide.
  - Le *nouveau préfixe* dans leur *clause renommer préfixe* est le *préfixe* attaché à la *clause préfixe* dans l'*énoncé d'octroi*, ou il est vide s'il n'existe pas de *clause préfixe* dans l'*énoncé d'octroi originel.*
  - Le *postfixe* dans la *clause renommer préfixe* est le *postfixe* correspondant dans la *fenêtre d'octroi.*
- La notation **FORBID ALL** est un abrégé syntaxique interdisant tous les *noms de champ* du nom de **neumode** (voir section 10.2.5).
- Si une *clause renommer préfixe* dans un *énoncé d'octroi* a un *postfixe d'octroi* qui contient un *préfixe* et **ALL** , alors elle est de la forme

  (OP->NP) ! P ! **ALL**

  où *OP* et *NP* sont respectivement l'*ancien préfixe* et le *nouveau préfixe* éventuellement vides et *P* le *préfixe* dans le *postfixe d'octroi.* La *clause renommer préfixe* est alors équivalente à une clause de la forme

  (OP ! P->NP ! P) ! **ALL**

propriétés statiques: Quand un *énoncé d'octroi* contient (**DIRECTLY**) **PERVASIVE** , alors, toutes les *représentations textuelles de nom* octroyées par cet énoncé auront la propriété d'envahissement (direct) dans les domaines englobants du modulion dans lequel est contenu l'*énoncé d'octroi.*

Une *clause préfixe* a un *préfixe* qui lui est attaché, défini comme suit:

- Si la *clause préfixe* contient un *préfixe*, alors, ce *préfixe* lui est attaché.

- Sinon, le *préfixe* qui lui est attaché est un *préfixe simple* dont la représentation textuelle de nom est déterminée comme suit:

  - Si le domaine englobant directement le *préfixe* est un *module*, une *région*, un *quasi module* ou une *quasi région*, alors, la représentation textuelle de nom est la même que celle du nom de modulion de ce modulion.

  - Si le domaine englobant directement le *préfixe* est une *spec de région* ou une *spec de module*, alors, la représentation textuelle de nom est la représentation textuelle de nom qui précède **SPEC**

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

- Si c'est une *représentation textuelle de nom*, ou s'il contient une *représentation textuelle de nom* de **neumode**, alors, l'ensemble ne contient que cette *représentation textuelle de nom*.

- Sinon, soit *OP* l'*ancien préfixe* (éventuellement vide) de la *clause renommer préfixe* dans laquelle est placé le *postfixe d'octroi*, l'ensemble contient toutes les *représentations textuelles de nom* de la forme *OP ! N* (c.-à-d. obtenues en préfixant *N* par *OP*) pour toute *représentation textuelle de nom N* telle que *OP ! N* est fortement visible dans le domaine du modulion dans lequel le *postfixe d'octroi* est placé et octroyable par ce modulion.

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

Si l'*énoncé d'octroi* se trouve dans le domaine d'une *région* ou d'une *spec de région*, il ne doit pas octroyer une *représentation textuelle de nom* qui est liée dans le domaine de la *région* ou de la *spec de région* à des *définitions* **intra-régionales**.

La *clause renommer préfixe* dans un *énoncé d'octroi* doit avoir un *postfixe d'octroi*.

Une *représentation textuelle de nom* octroyée par un *énoncé d'octroi* dans le domaine d'une *spec de module* ou d'une *spec de région* doit être liée dans ce domaine à une *définition* entourée par ce domaine. Cette *définition* ne peut pas avoir un *quasi module* ou une *quasi région* qui lui soit attaché.

Si un *énoncé d'octroi* contient une *clause de préfixe* qui ne contient pas de *préfixe*, alors son modulion immédiatement englobant ne doit pas être un *contexte* et,

- si son modulion immédiatement englobant est un *module*, une *région*, un *quasi module* ou une *quasi région*, alors, il doit être nommé (c.-à-d. qu'il doit être précédé par une *définition* suivie par un deux points);

- si son modulion immédiatement englobant est une *spec de module* ou une *spec de région*, alors, il doit être précédé par une *représentation textuelle de nom simple*.

exemples:

25.10	**GRANT ALL PREFIXED** *stack ! char ;*	*(1.1)*
6.44	*gregorian_date, julian_day_number*	*(2.1)*

10.2.4.5 Enoncé de saisie

syntaxe:

<énoncé de saisie> ::= *(1)*
SEIZE *<clause renommer préfixe>* { , *<clause renommer préfixe>*}*; *(1.1)*
| **SEIZE** *<fenêtre de saisie>* [*<clause de préfixe>*] ; *(1.2)*

<fenêtre de saisie> ::= *(2)*
<postfixe de saisie> { , *<postfixe de saisie>* }* *(2.1)*

<postfixe de saisie> ::= *(3)*
<représentation textuelle de nom> *(3.1)*
| *<représentation textuelle de nom de modulion>* **ALL** *(3.2)*
| [*<préfixe> !*] **ALL** *(3.3)*

<représentation textuelle de nom de modulion> ::= *(4)*
< représentation textuelle de nom de modulion> *(4.1)*

sémantique: Les énoncés de saisie sont un moyen d'étendre la visibilité des représentations textuelles de nom dans le domaine de groupes vers les domaines de modulions immédiatement englobés.

Les règles de visibilité suivantes s'appliquent:

- Si une *représentation textuelle de nom* qui a la propriété d'envahissement (directement) dans les domaines immédiatement englobants est saisie, elle sera **directement fortement visible** dans le domaine du modulion saisissant et conserve la propriété d'envahissement (directement).

- Si l'*énoncé de saisie* contient une (des) *clause(s) renommer préfixe*, il a l'effet de sa (ses) *clause(s) renommer préfixe* (voir 10.2.4.3).

- Si l'*énoncé de saisie* contient une *fenêtre de saisie*, c'est la notation abrégée pour un ensemble d'*énoncés de saisie* ayant des *clauses renommer préfixe* formées comme suit:

  - Pour chaque *postfixe de saisie* dans la *fenêtre de saisie*, il existe un *énoncé de saisie* correspondant.

  - L'*ancien préfixe* dans la *clause renommer préfixe* est le *préfixe* attaché à la *clause de préfixe* dans l'*énoncé de saisie*, ou il est vide s'il n'existe pas de *clause préfixe* dans l'*énoncé de saisie* originel.

  - Le *nouveau préfixe* de leur *clause renommer préfixe* est vide.

  - Le *postfixe* dans leur *clause renommer préfixe* est le *postfixe* correspondant de la *fenêtre de saisie.*

- Si une *clause renommer préfixe* dans un *énoncé de saisie* à un *postfixe de saisie* qui contient un *préfixe* et **ALL** , alors il a la forme

  (OP–>NP) ! P ! **ALL**

  où OP et NP sont respectivement l'*ancien préfixe* et le *nouveau préfixe* éventuellement vides et P le *préfixe* dans le *postfixe de saisie.* La *clause renommer préfixe* est alors équivalente à une clause de la forme

  (OP ! P–>NP ! P) ! **ALL**

- Si une *clause renommer préfixe* dans un *énoncé de saisie* a un *postfixe de saisie* qui contient une *représentation textuelle de nom* de modulion et **ALL** , alors il prend la forme

  (OP–>NP) ! MN **ALL**

  où *OP* et *NP* sont respectivement l'*ancien préfixe* et le *nouveau préfixe* éventuellement vides et *MN* est la *représentation de nom* de modulion dans le *postfixe de saisie.* La *clause renommer préfixe* est alors équivalente à un ensemble d'*énoncés de saisie*, dont chacun a une *clause renommer préfixe* de la forme

  (OP–>NP) ! NS

  avec un *énoncé de saisie* pour chaque *représentation textuelle* de *NS*, tel que:

  - *NS* est fortement visible dans le domaine qui englobe immédiatement le modulion immédiatement englobant l'*énoncé de saisie* et peut être saisi par ce modulion;

– *NS* est octroyé par le modulion attaché à la *définition* à laquelle *MN* est lié dans le domaine englobant immédiatement le modulion dans lequel se trouve l'*énoncé de saisie*.

propriété statique: Un *postfixe de saisie* a un ensemble de *représentations textuelles de nom*, défini comme suit:

- Si le *postfixe de saisie* est une *représentation textuelle de nom*, l'ensemble ne contient que la *représentation textuelle de nom*.
- Sinon, si le *postfixe de saisie* est **ALL** , soit *OP* l'*ancien préfixe* (éventuellement vide) de la *clause renommer préfixe* dont fait partie le *postfixe de saisie*, alors, l'ensemble contient toutes les *représentations textuelles de nom* de la forme *OP ! S*, pour toutes les *représentations textuelles de nom S*, telles que *OP ! S* est fortement visible dans le domaine englobant immédiatement le modulion dans lequel se trouve l'*énoncé de saisie* et peut être saisi par ce modulion.

conditions statiques: Une *représentation textuelle de nom* dans l'ensemble d'anciennes *représentations textuelles de nom* attaché à une *clause renommer préfixe* dans un *énoncé de saisie* ne doit pas être liée à une *définition* de **valeur faire-avec** ni à une *définition* de **locus faire-avec** dans un domaine qui englobe immédiatement le modulion dans lequel se trouve l'*énoncé de visibilité*.

La *clause renommer préfixe* dans un *énoncé de saisie* doit avoir un *postfixe de saisie*.

Si un *énoncé de saisie* contient une *clause de préfixe* qui ne contient pas de *préfixe*, alors, son modulion immédiatement englobant ne doit pas être un *contexte* et

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

La *représentation textuelle de nom* de **modulion** dans une *représentation textuelle de nom de modulion* doit être liée, dans des domaines englobant directement le domaine dans lequel se trouve la *représentation textuelle de nom de modulion*, à une *définition* de **modulion**. Il ne doit pas être lié à une *définition* à laquelle est attaché une *quasi région* ou un *quasi module*.

exemples:

25.35 **SEIZE** *(stack ! int -> stack) !* **ALL** *;* *(1.1)*
25.26 **SEIZE ALL PREFIXED** *stack ;* *(1.2)*

10.2.5 Visibilité de noms de champ

Des *noms de champ* ne peuvent apparaître que dans les contextes suivants:

- La *sélection d'un champ* d'un *locus structure* ou d'un *champ de valeur structure*.
- Les *multiplets de structure avec indices*.
- Les *clauses d'interdiction* d'un *énoncé d'octroi*.

Dans ces cas, la *représentation textuelle de nom* du *nom de champ* peut être limitée à une *définition de nom de champ* dans le mode M ou dans le mode définissant de M, obtenue comme suit:

- M est le mode du *locus* **structure** ou de la *valeur* **structure** (**forte**).
- M est le mode du *multiplet de structure*.
- M est le mode de la *définition* à laquelle la *représentation textuelle de nom* de **neumode** est liée dans le domaine dans lequel se trouve la *clause d'interdiction*.

Toutefois, si la **nouveauté** de M est un *nom* de **neumode** qui a été octroyé par un modulion avec une *clause d'interdiction*, alors, en dehors du modulion octroyant, les *noms de champ* mentionnés dans la *liste de noms d'interdiction* sont invisibles et ne peuvent être utilisés.

11 FILETS D'EXCEPTION

11.1 GÉNÉRALITÉS

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.

11.2 FILETS

syntaxe:

<filet> ::= *(1)*

ON { *<choix d'exceptions>*}* [**ELSE** *<liste d'énoncés d'action>*] **END** *(1.1)*

<choix d'exceptions> ::= *(2)*

(<liste d'exceptions>) : <liste d'énoncés d'action> *(2.1)*

sémantique: 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 d'exception 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 d'exception 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 d'**exception** dans toutes les occurrences de *liste d'exceptions* doivent être différents.

conditions dynamiques: L'exception *SPACEFAIL* arrive si on entame une liste d'énoncés d'action et que les requêtes de mémoire ne peuvent être satisfaites.

exemples:

10.47
```
ON
  ( ALLOCATEFAIL ): CAUSE overflow;
END
```
(1.1)

11.3 IDENTIFICATION DE FILET

Quand une exception E arrive dans une action A, ou un énoncé decriptif 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, un module ou une région parenthésés, le filet qui convient (si présent) est le filet qui convient pour l'action, le module ou la région parenthésés par rapport à E;

3. sinon, si A ou D 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.

12 OPTIONS POUR L'IMPLÉMENTATION

12.1 OPÉRATIONS PRÉDÉFINIES

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 une représentation textuelle de nom **réservé** peut être passé comme paramètre. L'appel d'opération prédéfinie peut rendre une valeur ou un locus. Le 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 domaine de la définition du processus imaginaire le plus externe ou dans un contexte quelconque (voir section 8.8). 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).

12.2 MODES ENTIER DÉFINIS PAR L'IMPLÉMENTATION

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.

12.3 NOMS DE REGISTRE DÉFINIS PAR L'IMPLÉMENTATION

Une implémentation peut définir un ensemble de noms de **registre** prédéfinis (voir sections 2.7 et 3.7).

12.4 NOMS D'EXCEPTION ET DE PROCESSUS DÉFINIS PAR L'IMPLÉMENTATION

Une implémentation peut définir un ensemble de noms de **processus** définis par l'implémentation, c.-à-d. des noms de **processus** dont la définition n'est pas spécifiée en CHILL. La définition est considérée être placée dans le domaine du processus imaginaire le plus externe ou dans un contexte quelconque. Les processus de ce nom peuvent être démarrés et des valeurs exemplaire les dénotant peuvent être manipulées.

Une implémentation peut définir un ensemble de noms **d'exception**.

12.5 FILETS DÉFINIS PAR L'IMPLÉMENTATION

Une implémentation peut spécifier qu'un filet défini par l'implémentation termine la définition de processus imaginaire la plus externe (voir section 8.8). Les noms **d'exceptions** et les actions dans un filet défini par l'implémentation peuvent spécifier 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.

12.6 REPÉRABILITÉ DÉFINIE PAR L'IMPLÉMENTATION

Une implémentation peut définir d'autres (sous-)locus comme **repérables**, en plus des locus qui sont définis comme étant **repérables** par le langage (voir section 4.2.1).

12.7 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

La représentation textuelle de nom **réservé ARRAY** devrait être soit obligatoire soit interdite.

RETURNS

Dans des définitions de procédure avec une **spec de résultat**, la représentation textuelle de nom **réservé RETURNS** devrait être soit obligatoire soit interdite.

Modes structure

Les modes structures doivent se noter soit dans la notation de structure emboîtée soit dans la notation de structure étagée.

Parenthèses de littéral et parenthèses de multiplet

Si des crochets carrés sont disponibles dans l'alphabet de représentation, les crochets [et] peuvent s'employer au lieu de respectivement *(:* et *:)*.

APPENDICE A: ENSEMBLES DE CARACTÈRES POUR LE LANGAGE CHILL

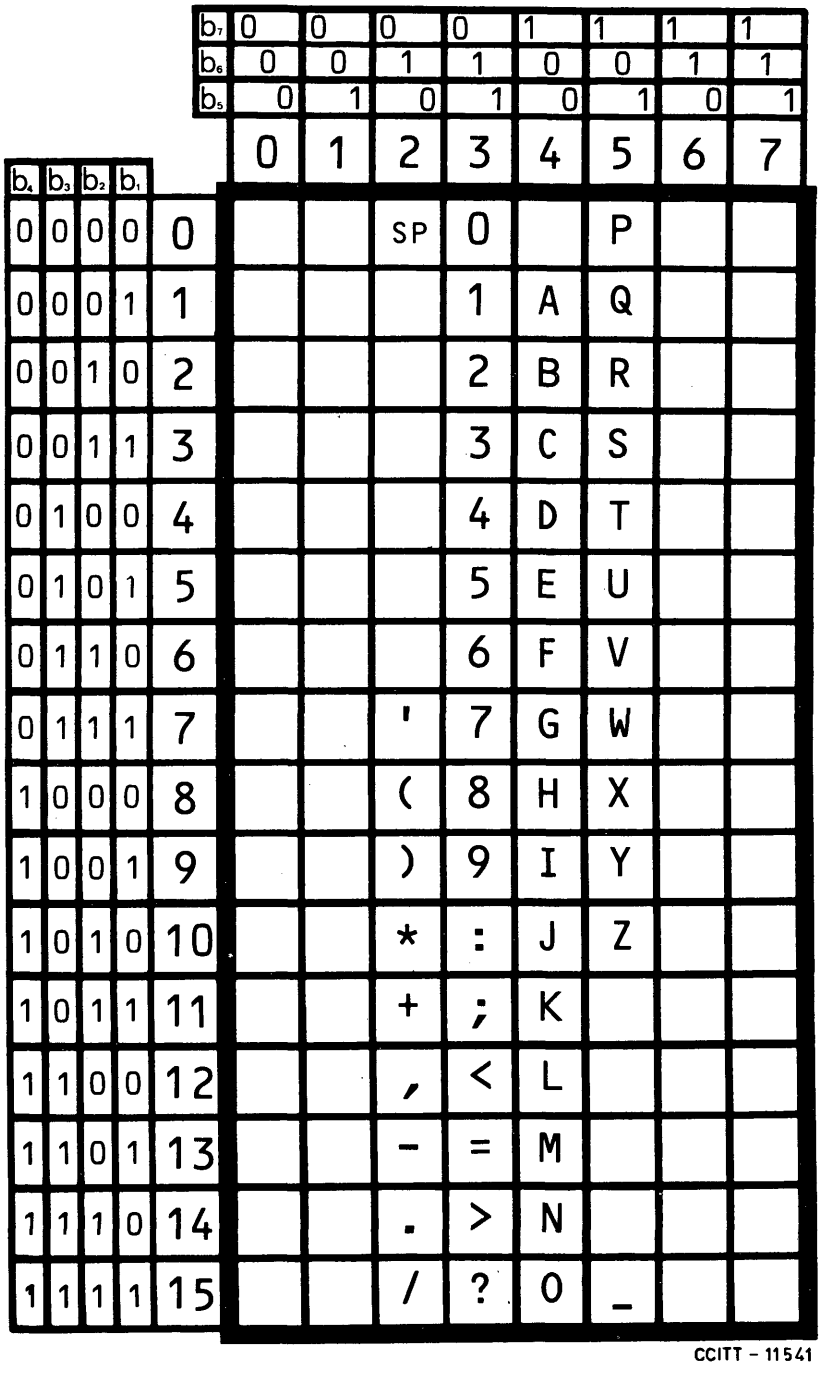
A.1 ALPHABET CCITT NO. 5 VERSION INTERNATIONALE DE RÉFÉRENCE

Recommandation V3 (la représentation interne est le nombre binaire formé par les bits b7 à b1, où b1 est le bit le moins significatif).

	b7	0	0	0	0	1	1	1	1
	b6	0	0	1	1	0	0	1	1
	b5	0	1	0	1	0	1	0	1
		0	1	2	3	4	5	6	7
b4 b3 b2 b1									
0 0 0 0	0	NUL	TC7 (DLE)	SP	0	@	P	`	p
0 0 0 1	1	TC1 (SOH)	DC1	!	1	A	Q	a	q
0 0 1 0	2	TC2 (STX)	DC2	"	2	B	R	b	r
0 0 1 1	3	TC3 (ETX)	DC3	#	3	C	S	c	s
0 1 0 0	4	TC4 (EOT)	DC4	¤	4	D	T	d	t
0 1 0 1	5	TC5 (ENQ)	TC8 (NAK)	%	5	E	U	e	u
0 1 1 0	6	TC6 (ACK)	TC9 (SYN)	&	6	F	V	f	v
0 1 1 1	7	BEL	TC10 (ETB)	'	7	G	W	g	w
1 0 0 0	8	FE0 (BS)	CAN	(	8	H	X	h	x
1 0 0 1	9	FE1 (HT)	EM	)	9	I	Y	i	y
1 0 1 0	10	FE2 (LF)	SUB	*	:	J	Z	j	z
1 0 1 1	11	FE3 (VT)	ESC	+	;	K	[	k	{
1 1 0 0	12	FE4 (FF)	IS4 (FS)	,	<	L	\	l	|
1 1 0 1	13	FE5 (CR)	IS3 (GS)	-	=	M	]	m	}
1 1 1 0	14	SO	IS2 (RS)	.	>	N	^	n	~
1 1 1 1	15	SI	IS1 (US)	/	?	O	_	o	DEL

CCITT-11540

A.2 ALPHABET MINIMAL POUR LE LANGAGE CHILL



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
b_4 b_3 b_2 b_1	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	_		

CCITT - 11541

APPENDICE B: SYMBOLES SPÉCIAUX ET COMBINAISONS DE CARACTÈRES

	Nom	Usage
;	point-virgule	terminateur d'énoncé etc.
,	virgule	séparateur dans différentes constructions
(	parenthèse gauche	parenthèse ouvrante dans différentes constructions
)	parenthèse droite	parenthèse fermante dans différentes constructions
[	crochet carré gauche	crochet ouvrant d'un multiplet
]	crochet carré droit	crochet fermant d'un multiplet
(:	crochet de multiplet gauche	crochet ouvrant d'un multiplet
:)	crochet de multiplet droit	crochet fermant d'un multiplet
:	deux points	indicateur d'étiquette, d'intervalle
.	point	symbole de sélection de champ
:=	symbole d'affectation	affectation, initialisation
<	inférieur à	opérateur relationnel
<=	inférieur ou égal à	opérateur relationnel
=	égal à	opérateur relationnel, affectation, initialisation
/=	différent de	opérateur relationnel
>=	supérieur ou égal à	opérateur relationnel
>	supérieur à	opérateur relationnel
+	plus	opérateur d' addition
—	minus	opérateur de soustraction
*	astérisque	opérateur de multiplication, valeur indéfinie, valeur anonyme
/	solidus	opérateur de division
//	double solidus	opérateur de concaténation
—>	flèche	repérage ou dérepérage
<>	diamant	début ou fin d'une clause de directive
/*	ouverture de commentaire	crochet de début de commentaire
*/	fin de commentaire	crochet de fin de commentaire
'	apostrophe	symbole de début ou de fin de divers littéraux
' '	double apostrophe	apostrophe dans un littéral de caractère ou chaîne de caractère
/.	opérateur préfixant	préfixage de noms
!	opérateur préfixant	préfixage de noms
B'	qualification littérale	base binaire pour littéral
D'	qualification littérale	base décimale pour littéral
H'	qualification littérale	base hexadécimale pour littéral
O'	qualification littérale	base octale pour littéral
C'	qualification littérale	représentation hexadécimale pour littéral de chaîne de caractères

APPENDICE C: REPRÉSENTATIONS TEXTUELLES DE NOM SIMPLE SPÉCIALES

C.1 REPRÉSENTATIONS TEXTUELLES DE NOM SIMPLE RÉSERVÉES

ACCESS
ADDR
ALL
ARRAY
ASSERT

BASED
BEGIN
BUFFER
BY

CALL
CASE
CAUSE
CONTEXT
CONTINUE

DCL
DELAY
DIRECTLY
DO
DOWN
DYNAMIC

ELSE
ELSIF
END
ENTRY
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MODULE

NEWMODE
NONREF
NOPACK

OD
OF
ON
OUT

PACK
PERVASIVE
POS
POWERSET
PREFIXED
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
REMOTE
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
SPEC
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

THEN
TO

UP

WHILE
WITH

C.2 REPRÉSENTATIONS TEXTUELLES DE NOM SIMPLE PRÉDÉFINIES

ABS
AND
ALLOCATE
ASSOCIATE
ASSOCIATION

BIN
BIT
BOOL

CARD
CHAR
CONNECT
CREATE

DELETE
DISCONNECT
DISSOCIATE

EXISTING
FALSE
FIRST

GETASSOCIATION
GETSTACK
GETUSAGE

INDEXABLE
INSTANCE
INT
ISASSOCIATED

LAST
LOWER

MAX
MIN
MOD
MODIFY

NOT
NULL
NUM

OR
OUTOFFILE

PRED
PTR

READABLE
READONLY
READRECORD
READWRITE
REM

SEQUENCIBLE
SAME
SIZE
SUCC

TERMINATE
THIS
TRUE

UPPER
USAGE

VARYING

WHERE
WRITEABLE
WRITEONLY
WRITERECORD

XOR

C.3 NOMS D'EXCEPTION

ALLOCATEFAIL
ASSERTFAIL
ASSOCIATEFAIL
CONNECTFAIL
CREATEFAIL
DELAYFAIL
DELETEFAIL
EMPTY
EXTINCT
MODIFYFAIL
NOTCONNECTED
NOTASSOCIATED
OVERFLOW
RANGEFAIL
READFAIL
RECURSEFAIL
SENDFAIL
SPACEFAIL
TAGFAIL
TERMINATEFAIL
WRITEFAIL

C.4 DIRECTIVES

FREE

APPENDICE D: EXEMPLES DE PROGRAMMES

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;
      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
5        add:
6        PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
7                RETURN [c1.re+c2.re,c1.im+c2.im];
8        END add;
9
10       mult:
11       PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
12               RETURN [c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re];
13       END mult;
14
15       GRANT add, mult;
16       SYNMODE operand_mode=complex;
17       GRANT operand_mode;
18       SYN neutral_for_add=complex [ 0,0 ],
19           neutral_for_mult=complex [ 1,0 ];
20       GRANT neutral_for_add,
21             neutral_for_mult;
22
23   END complex_operations;
```

4. **Arithmétique d'ordre général**

```
1   general_order_arithmetic: /* from collected algorithms from CACM no.93 */
2    MODULE
3        op:
4        PROC (a INT INOUT , b,c,order INT )
5                EXCEPTIONS (wrong_input) RECURSIVE ;
6            DCL d INT ;
7            ASSERT b>0 AND c>0 AND order>0
8                ON (ASSERTFAIL):
9                    CAUSE wrong_input;
10               END ;
11           CASE order OF
12               (1):        a := b+c;
13                           RETURN ;
14               (2):        d := 0;
15               ( ELSE ): d := 1;
16           ESAC ;
17           DO FOR i := 1 TO c;
18               op (a,b,d,order-1);
19               d := a;
20           OD ;
21           RETURN ;
22       END op;
23
24       GRANT op;
25
26   END general_order_arithmetic;
```

5. **Additionner bit à bit et vérifier le résultat**

```
1   add_bit_by_bit:
2    MODULE
3       adder:
4        PROC (a STRUCT (a2,a1 BOOL ) IN , b STRUCT (b2,b1 BOOL ) IN )
5                RETURNS ( STRUCT (c4,c2,c1 BOOL ));
6           DCL c STRUCT (c4,c2,c1 BOOL );
7           DCL k2,x,w,t,s,r BOOL ;
8           DO WITH a,b,c;
9              k2 := a1 AND b1;
10             c1 := NOT k2 AND (a1 OR b1);
11             x := a2 AND b2 AND k2;
12             w := a2 OR b2 OR k2;
13             t := b2 AND k2;
14             s := a2 AND k2;
15             r := a2 AND b2;
16             c4 := r OR s OR t;
17             c2 := x OR (w AND NOT c4);
18          OD ;
19          RETURN c;
20       END adder;
21       GRANT adder;
22    END add_bit_by_bit;
23
24  exhaustive_checker:
25    MODULE
26       SEIZE adder;
27       DCL a STRUCT (a2,a1 BOOL ),
28             b STRUCT (b2,b1 BOOL );
29       SYNMODE res= ARRAY (1:16) STRUCT (c4,c2,c1 BOOL );
30       DCL r INT , results res;
31       DO WITH a,b;
32          r := 0;
33          DO FOR a2 IN BOOL ;
34             DO FOR a1 IN BOOL ;
35                DO FOR b2 IN BOOL ;
36                   DO FOR b1 IN BOOL ;
37                      r+ := 1;
38                      results (r) := adder (a,b);
39                   OD ;
40                OD ;
41             OD ;
42          OD ;
43       OD ;
44       ASSERT
45          results=res [[ FALSE , FALSE , FALSE ],[ FALSE , FALSE , TRUE ],
46                       [ FALSE , TRUE , FALSE ] ,[ FALSE , TRUE , TRUE ],
47                       [ FALSE , FALSE , TRUE ] ,[ FALSE , TRUE , FALSE ],
48                       [ FALSE , TRUE , TRUE ]  ,[ TRUE , FALSE , FALSE ],
49                       [ FALSE , TRUE , FALSE ] ,[ FALSE , TRUE , TRUE ],
50                       [ TRUE , FALSE , FALSE ] ,[ TRUE , FALSE , TRUE ],
51                       [ FALSE , TRUE , TRUE ]  ,[ TRUE , FALSE , FALSE ],
52                       [ TRUE , FALSE , TRUE ]  ,[ TRUE , TRUE , FALSE ]];
53    END exhaustive_checker;
```

```
1   playing_with_dates:
2    MODULE /* from collected algorithms from CACM no. 199 */
3        SYNMODE month= SET (jan,feb,mar,apr,may,jun,
4                    jul,aug,sep,oct,nov,dec);
5        NEWMODE date= STRUCT (day INT (1:31), mo month, year INT );
6
7        gregorian_date:
8        PROC (julian_day_number INT )(date);
9            DCL j INT := julian_day_number,
10                   d,m,y INT ;
11           j := 1_721_119;
12           y := (4 * j - 1) / 146_097;
13           j := 4 * j - 1 - 146_097 * y;
14           d := j / 4;
15           j := (4 * d + 3) / 1_461;
16           d := 4 * d + 3 - 1_461 * j;
17           d := (d + 4) / 4;
18           m := (5 * d - 3) / 153;
19           d := 5 * d - 3 - 153 * m;
20           d := (d + 5) / 5;
21           y := 100 * y + j;
22           IF m<100 THEN m + := 3;
23                    ELSE m - := 9;
24                         y + := 1;
25           FI ;
26           RETURN [d,month (m+1), y];
27       END gregorian_date;
28
29       julian_day_number:
30       PROC (d date)( INT );
31           DCL c,y,m INT ;
32           DO WITH d;
33               m := NUM (mo)+1;
34               IF m>2 THEN m - := 3;
35                      ELSE m + := 9;
36                           year - := 1;
37               FI ;
38               c := year/100;
39               y := year-100*c;
40               RETURN (146_097*c)/4+(1_461*y)/4
41                          +(153+m+c)/5+day+1_721_119;
42           OD ;
43       END julian_day_number;
44       GRANT gregorian_date, julian_day_number;
45   END playing_with_dates;
46
47   test:
48   MODULE
49       SEIZE gregorian_date, julian_day_number;
50       ASSERT julian_day_number ([ 10,dec,1979 ])=julian_day_number
51                (gregorian_date(julian_day_number([ 10,dec,1979 ])));
52   END test;
```

7. Nombres romains

```
1   Roman:
2    MODULE
3        SEIZE n,rn;
4        GRANT convert;
5       convert:
6        PROC () EXCEPTIONS (string_too_small);
7            DCL r INT := 0;
8            DO WHILE n>=1_000;
9                rn(r) := 'M';
10               n - := 1_000;
11               r + := 1;
12           OD ;
13           IF n>500 THEN rn(r) := 'D';
14                         n - := 500;
15                         r + := 1;
16           FI ;
17           DO WHILE n>=100;
18               rn(r) := 'C';
19               n - := 100;
20               r + := 1;
21           OD ;
22           IF n>=50 THEN rn(r) := 'L';
23                         n - := 50;
24                         r + := 1;
25           FI ;
26           DO WHILE n>=10;
27               rn(r) := 'X';
28               n - := 10;
29               r + := 1;
30           OD ;
31           IF n>=5 THEN rn(r) := 'V';
32                        n - := 5;
33                        r + := 1;
34           FI ;
35           DO WHILE n>=1;
36               rn(r) := 'I';
37               n - := 1;
38               r + := 1;
39           OD ;
40           RETURN ;
41       END ON (RANGEFAIL): DO FOR i := 0 TO UPPER (rn);
42                               rn(i) := '.';
43                           OD ;
44                           CAUSE string_too_small;
45       END convert;
46    END Roman;
47   test:
48    MODULE
49       SEIZE convert;
50       DCL n INT INIT := 1979;
51       DCL rn CHAR (20) INIT := (20)' ';
52       GRANT n,rn;
53      convert ();
54       ASSERT rn='MDCCCCLXXVIIII'//(6)' ';
55    END test;
```

8. **Compter les lettres dans une chaîne de caractères de longueur arbitraire**

```
1   letter_count:
2    MODULE
3        SEIZE max;
4        DCL letter POWERSET CHAR INIT := ['A' : 'Z'];
5       count:
6        PROC (input ROW 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       SYNMODE results= ARRAY ('A':'Z') INT ;
19       DCL c CHAR (10) INIT := 'A-B<ZAA9K' ' ';
20       DCL output results;
21       SYN max=10_000;
22       GRANT max;
23       SEIZE count;
24      count (-> c,output);
25       ASSERT output=results [('A') : 3,('B','K','Z') : 1, ( ELSE ) : 0];
26   END test;
```

9. **Nombres premiers**

```
1   prime:
2    MODULE
3
4        SYN max = H'7FFF;
5        NEWMODE number_list = POWERSET INT (2:max);
6        SYN empty = number_list [ ];
7        DCL sieve number_list INIT := [ 2:max ],
8              primes number_list INIT := empty;
9        GRANT primes;
10       DO WHILE sieve/=empty;
11             primes OR := [ MIN (sieve)];
12              DO FOR j := MIN (sieve) BY MIN (sieve) TO max;
13                  sieve - := [j];
14              OD ;
15       OD ;
16   END prime;
```

10. **Implémenter des piles de deux manières différentes, transparentes pour l'utilisateur**

```
1   stack : MODULE
2         NEWMODE element = STRUCT (a INT , b BOOL );
3         stacks_1:
4         MODULE
5             SEIZE element;
6             SYN max=10_000,min=1;
7             DCL stack ARRAY (min : max) element,
8                   stackindex INT INIT := min;
9
10            push:
11            PROC (e element) EXCEPTIONS (overflow);
12                  IF stackindex=max
13                      THEN CAUSE overflow;
14                  FI ;
15                  stackindex + := 1;
16                  stack (stackindex) := e;
17                  RETURN ;
18            END push;
19
20            pop:
21            PROC () EXCEPTIONS (underflow);
22                  IF stackindex=min
23                      THEN CAUSE underflow;
24                  FI ;
25                  stackindex - := 1;
26                  RETURN ;
27            END pop;
28
29            elem:
30            PROC (i INT )(element LOC ) EXCEPTIONS (bounds);
31                  IF i<min OR i>max
32                      THEN CAUSE bounds;
33                  FI ;
34                  RETURN stack (i);
35            END elem;
36
37            GRANT push,pop,elem;
38        END stacks_1;
39        stacks_2:
40        MODULE
41            SEIZE element;
42            NEWMODE cell= STRUCT (pred,succ REF cell,info element);
43            DCL p,last,first REF cell INIT := NULL ;
44
45            push:
46            PROC (e element) EXCEPTIONS (overflow);
47               p := ALLOCATE (cell) ON
48                                    (ALLOCATEFAIL) : CAUSE overflow;
49                                     END ;
50                IF last= NULL
51                    THEN first := p;
52                         last := p;
53                    ELSE last ->. succ := p;
54                         p ->. pred := last;
55                         last := p;
56                FI ;
57                last ->. info := e;
58                RETURN ;
```

```
59          END push;
60
61          pop:
62          PROC () EXCEPTIONS (underflow);
63              IF last= NULL
64                  THEN CAUSE underflow;
65              FI ;
66              p := last;
67              last := last ->. pred;
68              IF last = NULL
69                  THEN first := NULL ;
70                  ELSE last ->. succ := NULL ;
71              FI ;
72              TERMINATE(p);
73              RETURN ;
74          END pop;
75
76          elem:
77          PROC (i INT ) (element LOC ) EXCEPTIONS (bounds);
78              IF first= NULL
79                  THEN CAUSE bounds;
80              FI ;
81              p := first;
82              DO FOR j := 2 TO i;
83                  IF p ->. succ= NULL
84                      THEN CAUSE bounds;
85                  FI ;
86                  p := p ->. succ;
87              OD ;
88              RETURN p ->. info;
89          END elem;
90
91          /* GRANT push,pop,elem; */
92      END stacks_2;
93  END stack;
```

```
1   chess_fragments:
2    MODULE
3        NEWMODE piece= STRUCT (color SET (white,black),
4                               kind SET (pawn,rook,knight,bishop,queen,king));
5        NEWMODE column= SET (a,b,c,d,e,f,g,h);
6        NEWMODE line= INT (1 : 8);
7        NEWMODE square= STRUCT (status SET (occupied,free),
8                                CASE status OF
9                                (occupied) : p piece,
10                               (free) :
11                               ESAC );
12       NEWMODE board= ARRAY (line) ARRAY (column) square;
13       NEWMODE move= STRUCT (lin_1,lin_2 line,
14                             col_1,col_2 column);
15
16       initialise:
17       PROC (bd board INOUT );
18           bd := [ (1): [(a,h):    [.status: occupied, .p : [white,rook]],
19                        (b,g):     [.status: occupied, .p : [white,knight]],
20                        (c,f):     [.status: occupied, .p : [white,bishop]],
21                        (d):       [.status: occupied, .p : [white,queen]],
22                        (e):       [.status: occupied, .p : [white,king]]],
23                  (2): [( ELSE ):[.status: occupied, .p : [white,pawn]]],
24                  (3:6):[( ELSE ):[.status: free]],
25                  (7): [( ELSE ):[.status: occupied, .p : [black,pawn]]],
26                  (8): [(a,h):    [.status: occupied, .p : [black,rook]],
27                        (b,g):     [.status: occupied, .p : [black,knight]],
28                        (c,f):     [.status: occupied, .p : [black,bishop]],
29                        (d):       [.status: occupied, .p : [black,queen]],
30                        (e):       [.status: occupied, .p : [black,king]]]
31               ];
32           RETURN ;
33       END initialise;
34       register_move:
35       PROC (b board LOC ,m move) EXCEPTIONS (illegal);
36           DCL starting square LOC := b (m.lin_1)(m.col_1),
37               arriving square LOC := b (m.lin_2)(m.col_2);
38           DO WITH m;
39               IF starting.status=free THEN CAUSE illegal; FI ;
40               IF arriving.status/=free THEN
41                   IF arriving.p.kind=king THEN CAUSE illegal; FI ;
42               FI ;
43               CASE starting.p.kind, starting.p.color OF
44                   (pawn),(white):
45                   IF col_1 = col_2 AND (arriving.status/=free
46                       OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
47                       OR (col_2= PRED (col_1) OR col_2= SUCC (col_1))
48                       AND arriving.status=free THEN CAUSE illegal; FI ;
49                       IF arriving.status/=free THEN
50                           IF arriving.p.color=white THEN CAUSE illegal; FI ; FI ;
51                   (pawn),(black):
52                   IF col_1=col_2 AND (arriving.status/=free
53                       OR NOT (lin_2=lin_1-1 OR lin_2=lin_1-2 AND lin_1=7))
54                       OR (col_2= PRED (col_1) OR col_2= SUCC (col_1))
55                       AND arriving.status=free THEN CAUSE illegal; FI ;
56                       IF arriving.status/=free THEN
57                           IF arriving.p.color=black THEN CAUSE illegal; FI ; FI ;
58                   (rook),(*):
```

```
59                  IF NOT ok_rook (b,m)
60                      THEN CAUSE illegal;
61                  FI ;
62                  (bishop),(*):
63                  IF NOT ok_bishop (b,m)
64                      THEN CAUSE illegal;
65                  FI ;
66                  (queen),(*):
67                  IF NOT ok_rook (b,m) AND NOT ok_bishop (b,m)
68                      THEN CAUSE illegal;
69                  FI ;
70                  (knight),(*):
71                  IF ABS ( ABS ( NUM (col_2)- NUM (col_1))
72                           - ABS (lin_2- lin_1)) /= 1
73                      OR ABS ( NUM (col_2)- NUM (col_1))
74                           + ABS (lin_2- lin_1) =/ 3 THEN CAUSE illegal; FI ;
75                  IF arriving.status/=free THEN
76                           IF arriving.p.color=starting.p.color THEN
77                           CAUSE illegal; FI ; FI ;
78                  (king),(*):
79                  IF ABS ( NUM (col_2)- NUM (col_1)) > 1
80                      OR ABS (lin_2- lin_1) > 1
81                      OR lin_2=lin_1 AND col_2=col_1 THEN CAUSE illegal; FI ;
82                  IF arriving.status/=free THEN
83                    IF arriving.p.color=starting.p.color THEN
84                    CAUSE illegal; FI ; FI ;/* checking king moving to check not implemented */

85              ESAC ;
86          OD ;
87          arriving := starting;
88          starting := [.status:free];
89          RETURN ;
90      END register_move;
91      ok_rook:
92      PROC (b board,m move)( BOOL );
93          DCL starting square := b (m.lin_1)(m.col_1),
94              arriving square := b (m.lin_2)(m.col_2);
95
96          DO WITH m;
97              IF NOT (col_2=col_1 OR lin_1=lin_2) THEN RETURN FALSE ; FI ;
98              IF arriving.status/=free THEN
99                  IF arriving.p.color=starting.p.color THEN ;
100                 RETURN FALSE ; FI ; FI ;
101             IF col_1=col_2
102                 THEN IF lin_1<lin_2
103                         THEN DO FOR lin := lin_1+1 TO lin_2-1;
104                                 IF b (lin)(col_1).status/=free
105                                     THEN RETURN FALSE ;
106                                 FI ;
107                             OD ;
108                         ELSE DO FOR lin := lin_1-1 DOWN TO lin_2+1;
109                                 IF b (lin)(col_1).status/=free
110                                     THEN RETURN FALSE ;
111                                 FI ;
112                             OD ;
113                     FI ;
114                 ELSIF col_1<col_2
115                         THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
116                                 IF b (lin_1)(col).status/=free
117                                     THEN RETURN FALSE ;
118                                 FI ;
```

```
119                                 OD ;
120                             ELSE DO FOR col := SUCC (col_2) DOWN TO PRED (col_1);
121                                     IF b (lin_1)(col).status/=free
122                                         THEN RETURN FALSE ;
123                                     FI ;
124                                 OD ;
125             FI ;
126             RETURN TRUE ;
127         OD ;
128     END ok_rook;
129     ok_bishop:
130     PROC (b board,m move)( BOOL );
131         DCL starting square := b (m.lin_1)(m.col_1),
132             arriving square := b (m.lin_2)(m.col_2),
133             col column;
134
135         DO WITH m;
136             CASE lin_2>lin_1,col_2>col_1 OF
137                 ( TRUE ),( TRUE ): col := col_1;
138                                 DO FOR lin := lin_1+1 TO lin_2-1;
139                                         col := SUCC (col);
140                                         IF b (lin)(col).status/=free
141                                             THEN RETURN FALSE ;
142                                         FI ;
143                                 OD ;
144                                 IF SUCC (col)/=col_2
145                                         THEN RETURN FALSE ;
146                                 FI ;
147                 ( TRUE ),( FALSE ): col := col_1;
148                                 DO FOR lin := lin_1+1 TO lin_2-1;
149                                         col := PRED (col);
150                                         IF b (lin)(col).status/=free
151                                             THEN RETURN FALSE ;
152                                         FI ;
153                                 OD ;
154                                 IF PRED (col)/=col_2
155                                         THEN RETURN FALSE ;
156                                 FI ;
157                 ( FALSE ),( TRUE ): col := col_1;
158                                 DO FOR lin := lin_1-1 DOWN TO lin_2+1;
159                                         col := SUCC (col);
160                                         IF b (lin)(col).status/=free
161                                             THEN RETURN FALSE ;
162                                         FI ;
163                                 OD ;
164                                 IF SUCC (col)/=col_2
165                                         THEN RETURN FALSE ;
166                                 FI ;
167                 ( FALSE ),( FALSE ): col := col_1;
168                                 DO FOR lin := lin_1-1 DOWN TO lin_2+1;
169                                         col := PRED (col);
170                                         IF b (lin)(col).status/=free
171                                             THEN RETURN FALSE ;
172                                         FI ;
173                                 OD ;
174                                 IF PRED (col)/=col_2
175                                         THEN RETURN FALSE ;
176                                 FI ;
177             ESAC ;
178             IF arriving.status=free THEN RETURN TRUE ;
179             ELSE RETURN arriving.p.color/=starting.p.color; FI ;
```

```
180         OD ;
181       END ok_bishop;
182       END chess_fragments;
```

12. Construire et manipuler une liste chaînée circulairement

```
1   circular_list:
2    MODULE
3       handle_list:
4        MODULE
5            GRANT insert, remove, node;
6            NEWMODE node= STRUCT (pred, suc REF node, value INT );
7            DCL pool ARRAY (1:1000)node;
8            DCL head node := (: NULL , NULL ,0 :);
9
10           insert: PROC (new node);
11               /* insert actions */
12           END insert;
13
14           remove: PROC ();
15               /* remove actions */
16           END remove;
17
18           initialize_list:
19           BEGIN
20               DCL last REF node := ->head;
21               DO FOR new IN pool;
22                   new.pred := last;
23                   last->.suc := ->new;
24                   last := ->new;
25                   new.value := 0;
26               OD ;
27               head.pred := last;
28               last->.suc := ->head;
29           END initialize_list;
30
31       END handle_list;
32       manipulate:
33       MODULE
34           SEIZE node, remove, insert;
35           DCL node_a node := (: NULL , NULL ,536 :);
36           remove();
37           remove();
38           insert(node_a);
39       END manipulate;
40   END circular_list;
```

13. **Une région pour donner des accès compétitifs à une ressource**

```
1   allocate_resources:
2    REGION
3        GRANT allocate, deallocate;
4        NEWMODE resource_set = INT (0:9);
5        DCL allocated ARRAY (resource_set) BOOL := (: (resource_set): FALSE :);
6        DCL resource_freed EVENT ;
7
8        allocate:
9        PROC ()(resource_set);
10           DO FOR EVER ;
11               DO FOR i IN resource_set;
12                   IF NOT allocated(i)
13                       THEN
14                           allocated(i) := TRUE ;
15                           RETURN i;
16                   FI ;
17               OD ;
18               DELAY resource_freed;
19           OD ;
20       END allocate;
21
22       deallocate:
23       PROC (i resource_set);
24           allocated(i) := FALSE ;
25           CONTINUE resource_freed;
26       END deallocate;
27
28   END allocate_resources;
```

14. **Mettre en attente les appels à un central**

```
1   switchboard:
2    MODULE
3       /* This example illustrates a switchboard which queues incoming calls
4          and feeds them to the operator at an even rate. Every time the
5          operator is ready one and only one call is let through. This is
6          handled by a call distributor which lets calls through at fixed
7          intervals. If the operator is not ready or there are other calls
8          waiting, a new call must queue up to wait for its turn. */
9        DCL operator_is_ready,
10             switch_is_closed EVENT ;
11
12      call_distributor:
13       PROCESS ();
14          wait:
15           PROC (x INT );
16              /*some wait action*/
17           END wait;
18           DO FOR EVER ;
19              wait(10 /*seconds*/);
20               CONTINUE operator_is_ready;
21           OD ;
22       END call_distributor;
23
24      call_process:
25       PROCESS ();
26           DELAY CASE
27          (operator_is_ready): /* some actions */ ;
28          (switch_is_closed): DO FOR i IN INT (1:100);
29                                    CONTINUE operator_is_ready;
30                                   /* empty the queue*/
31                              OD ;
32           ESAC ;
33       END call_process;
34
35      operator:
36       PROCESS ();
37           DCL time INT ;
38           DO FOR EVER ;
39              IF time = 1700
40                  THEN CONTINUE switch_is_closed;
41              FI ;
42           OD ;
43       END operator;
44
45       START call_distributor();
46       START operator();
47       DO FOR i IN INT (1:100);
48            START call_process();
49       OD ;
50   END switchboard;
```

```
1   <> FREE ( STEP );
2    MODULE
3    SIGNAL
4        acquire,
5        release=( INSTANCE ),
6        congested,
7        ready,
8        step,
9        readout=( INT );
10       GRANT ALL ;
11   END definitions;
12  counter_manager:
13   MODULE
14  /*   To illustrate the use of signals and the receive case, (buffers
15       might have been used instead) we will look at an example where an
16       allocator manages a set of resources, in this case a set of
17       counters. The module is part of a larger system where there are
18       users, that can request the services of the counter_manager. The
19       module is made to consist of two process definitions, one for the
20       allocation and one for the counters. initiate and terminate
21       are internal signals sent from the allocator
22       to the counters. All the other signals are external, being sent
23       from or to the users. */
24
25       SEIZE /* external signals */
26           acquire, release, congested,ready,step,readout;
27       SIGNAL initiate = ( INSTANCE ),
28                 terminate;
29       allocator:
30       PROCESS ();
31           NEWMODE no_of_counters = INT (1:100);
32       DCL counters ARRAY (no_of_counters)
33                                  STRUCT (counter INSTANCE ,status SET (busy,idle));
34       DO FOR each IN counters;
35           each := (: START counter(), idle :);
36       OD ;
37       DO FOR EVER ;
38       BEGIN
39           DCL user INSTANCE ;
40           await_signals:
41           RECEIVE CASE SET user;
42           (acquire):
43                DO FOR each IN counters;
44                    DO WITH each;
45                        IF status = idle
46                            THEN
47                                status := busy;
48                                SEND initiate (user) TO counter;
49                                EXIT await_signals;
50                        FI ;
51                    OD ;
52                OD ;
53                SEND congested TO user;
54           (release IN this_counter):
55                SEND terminate TO this_counter;
56               find_counter:
57                DO FOR each IN counters;
58                    DO WITH each;
```

```
59                      IF this_counter = counter
60                          THEN
61                              status := idle;
62                               EXIT find_counter;
63                      FI ;
64                  OD ;
65              OD find_counter;
66          ESAC await_signals;
67      END ;
68      OD ;
69      END allocator;
70      counter:
71      PROCESS ();
72          DO FOR EVER ;
73          BEGIN
74              DCL user INSTANCE ,
75                    count INT := 0;
76              RECEIVE CASE
77                  (initiate IN received_user):
78                       SEND ready TO received_user;
79                      user := received_user;
80              ESAC ;
81              work_loop:
82              DO FOR EVER ;
83                  RECEIVE CASE
84                      (step): count + := 1;
85                      (terminate):
86                           SEND readout(count) TO user;
87                           EXIT work_loop;
88                  ESAC ;
89              OD work_loop;
90          END ;
91          OD ;
92      END counter;
93      START allocator();
94  END counter_manager;
```

```
1   <> FREE ( STEP );
2
3   user_world:
4    MODULE
5   /* This example is the same as no.15 except that buffers are
6       used for communication in stead of signals.
7       The main difference is that processes are now identified
8       by means of references to local message buffers rather than
9       by instance values. There is one message buffer declared
10      local to each process. There is one set of message types
11      for each process definition. When started each process must
12      identify its buffer address to the starting process.
13      The user_world module sketches some of the environment in
14      which the counter_manager is used. */
15
16   SEIZE allocator;
17   GRANT user_buffers,user_messages,
18             allocator_messages, allocator_buffers,
19             counter_messages, counters_buffers;
20   NEWMODE
21      user_messages =
22           STRUCT (type SET (congested, ready,
23                     readout, allocator_id),
24                                  CASE type OF
25                                     (congested) : ,
26                                     (ready) : counter REF counters_buffers,
27                                     (readout) : count INT ,
28                                     (allocator_id): allocator REF allocator_buffers
29                                  ESAC ),
30      user_buffers = BUFFER (1) user_messages,
31      allocator_messages =
32           STRUCT (type SET (acquire, release, counter_id),
33                     CASE type OF
34                        (acquire) : user REF user_buffers,
35                        (release,
36                        counter_id): counter REF counters_buffers
37                     ESAC ),
38      allocator_buffers = BUFFER (1) allocator_messages,
39      counter_messages =
40           STRUCT (type SET (initiate, step, terminate),
41                     CASE type OF
42                        (initiate) : user REF user_buffers,
43                        (step,
44                        terminate):
45                     ESAC ),
46      counters_buffers = BUFFER (1) counter_messages;
47    DCL user_buffer user_buffers,
48        allocator_buf REF allocator_buffers,
49        counter_buf REF counters_buffers;
50    START allocator(->user_buffer);
51  allocator_buf := ( RECEIVE user_buffer).allocator;
52    END user_world;
53  counter_manager:
54    MODULE
55    SEIZE user_buffers,user_messages,
56          allocator_messages, allocator_buffers,
57          counter_messages, counters_buffers;
58    GRANT allocator;
```

```
59
60  allocator:
61   PROCESS (starter REF user_buffers);
62       DCL allocator_buffer allocator_buffers;
63       NEWMODE no_of_counters = INT (1:10);
64       DCL counters ARRAY (no_of_counters)
65                                  STRUCT (counter REF counters_buffers,
66                                                  status SET (busy, idle)),
67            message allocator_messages;
68       SEND starter->([allocator_id, ->allocator_buffer]);
69       DO FOR each IN counters;
70           START counter(->allocator_buffer);
71          each := [( RECEIVE allocator_buffer).counter, idle];
72       OD ;
73       DO FOR EVER ;
74           BEGIN
75           DCL user REF user_buffers;
76          message := RECEIVE allocator_buffer;
77          handle_messages:
78           CASE message.type OF
79          (acquire):
80               user := message.user;
81               DO FOR each IN counters;
82                   DO WITH each;
83                       IF status= idle
84                           THEN status := busy;
85                                   SEND counter->([initiate, user]);
86                                   EXIT handle_messages;
87                       FI ;
88                   OD ;
89               OD ;
90               SEND user->([congested]);
91          (release):
92               SEND message.counter->([terminate]);
93              find_counter:
94               DO FOR each IN counters;
95                   DO WITH each;
96                       IF message.counter = counter
97                           THEN status := idle;
98                                   EXIT find_counter;
99                       FI ;
100                  OD ;
101              OD find_counter;
102         (counter_id): ;
103          ESAC handle_messages;
104          END ;
105      OD ;
106  END allocator;
107 counter:
108  PROCESS (starter REF allocator_buffers);
109      DCL counter_buffer counters_buffers;
110      SEND starter->([counter_id, ->counter_buffer]);
111      DO FOR EVER ;
112          BEGIN
113              DCL user REF user_buffers,
114                  count INT := 0,
115                  message counter_messages;
116             message := RECEIVE counter_buffer;
117              CASE message.type OF
118                 (initiate): user := message.user;
119                                 SEND user->([ready, ->counter_buffer]);
```

```
120                 ELSE /* some error action */
121             ESAC ;
122         work_loop:
123          DO FOR EVER ;
124             message := RECEIVE counter_buffer;
125             CASE message.type OF
126                 (step) : count + := 1;
127                 (terminate): SEND user->([readout, count]);
128                                 EXIT work_loop;
129                 ELSE /* some error action */
130             ESAC ;
131         OD work_loop;
132     END ;
133     OD ;
134 END counter;
135 END counter_manager;
```

17. **Parcours de chaîne 1**

```
1   string_scanner1: /* This program implements strings by means
2                       of packed arrays of characters. */
3    MODULE
4       SYN
5          blanks ARRAY (0:9) CHAR PACK = [(*):' ' ], linelength = 132;
6       SYNMODE
7          stringptr = ROW ARRAY (lineindex) CHAR PACK ,
8          lineindex = INT (0:linelength-1);
9
10  scanner:
11   PROC (string stringptr, scanstart lineindex INOUT ,
12         scanstop lineindex, stopset POWERSET CHAR )
13          RETURNS ( ARRAY (0:9) CHAR PACK );
14      DCL count INT := 0,
15            res ARRAY (0:9) CHAR PACK := blanks;
16      DO
17          FOR c IN string->(scanstart:scanstop)
18          WHILE NOT (c IN stopset);
19             count + := 1;
20      OD ;
21      IF count>0
22          THEN
23              IF count>10
24                  THEN
25                     count := 10;
26              FI ;
27             res(0:count-1) := string->(scanstart:scanstart+count-1);
28      FI ;
29      RESULT res;
30      IF scanstart+count < scanstop
31          THEN
32             scanstart := scanstart+count+1;
33      FI ;
34   END scanner;
35
36   GRANT scanner;
37
38   END string_scanner1;
```

```
string_scanner2: /* This example is the same as no.17 but it uses
                    character string instead of packed arrays */
 MODULE
    SYN
       blanks = (10)' ', linelength = 132;
    SYNMODE
       stringptr = ROW CHAR (linelength),
       lineindex = INT (0:linelength-1);

    scanner:
        PROC (string stringptr, scanstart lineindex INOUT ,
              scanstop lineindex, stopset POWERSET CHAR )
              RETURNS ( CHAR (10));
           DCL count INT := 0;
           DO FOR i := scanstart TO scanstop
              WHILE NOT (string->(i) IN stopset);
                 count + := 1;
           OD ;
           IF count>0
              THEN
                 IF count>=10
                    THEN
                       RESULT string->(scanstart UP 10);
                    ELSE
                       RESULT string->(scanstart:scanstart+count-1)
                                 //blanks(count:9);
                 FI ;
              ELSE
                 RESULT blanks;
           FI ;
           IF scanstart+count < scanstop
              THEN
                 scanstart := scanstart+count+1;
           FI ;
        END scanner;

    GRANT scanner;

 END string_scanner2;
```

19. **Enlever un élément d'une liste doublement chaînée circulairement**

```
1   queue: MODULE
2         SYNMODE info= INT ;
3        queue_removal:
4         MODULE
5              SEIZE info;
6              GRANT remove;
7             remove:
8              PROC (p PTR ) RETURNS (info) EXCEPTIONS (EMPTY);
9                      /* This procedure removes the item referred to
10                        by p from a queue and returns the information
11                        contents of that queue element */
12                     DCL 1 x BASED (p),
13                           2 i info POS (0,8:31),
14                           2 prev PTR POS (1,0:15),
15                           2 next PTR POS (1,16:31);
16                     DCL prev, next PTR ;
17                    prev := x.prev;
18                    next := x.next;
19                    x.prev, x.next := NULL ;
20                     RESULT x.i;
21                    p := prev;
22                    x.next := next;
23                    p := next;
24                    x.prev := prev;
25             END remove;
26        END queue_removal;
27   END queue;
```

```
1   read_modify_write:
2   MODULE
3
4     /* this example indicates how the CHILL i/o concepts can be used */
5     /* to write an application where a record of a random accessible  */
6     /* file can be updated or added if not yet in use                 */
7
8     NEWMODE
9       index_set = INT (1:1000),
10      record_type = STRUCT (
11                                free    BOOL ,
12                                count   INT ,
13                                name    CHAR (20));
14
15    DCL
16      curindex          index_set,
17      file_association   ASSOCIATION ,
18      record_file        ACCESS (index_set) record_type,
19      record_buffer     record_type;
20
21    ASSOCIATE (file_association,'DSK:RECORDS.DAT');     /* create association */
22     CONNECT (record_file,file_association, READWRITE ); /* connect to file    */
23    curindex := 123;                                    /* position record    */
24     READRECORD (record_file,curindex,record_buffer);    /* read the record    */
25       IF record_buffer.free                             /* if record is free  */
26         THEN                                            /* the claim and      */
27           record_buffer.free := FALSE                   /* initialize it      */
28           record_buffer.count := 0;
29           record_buffer.name := 'CHILL I/O concept     ';
30       FI ;
31    record_buffer.count + := 1;                         /* increment its count*/
32     WRITERECORD (record_file, curindex, record_buffer); /* write the record   */
33     DISSOCIATE (file_association);                      /* end the association*/
34
35  END read_modify_write;
```

```
1   merge_sorted_files:
2   MODULE
3
4     /* this example shows how two sorted files can be merged into one */
5     /* new sorted file, where the field 'key' is used for sorting     */
6     /* the old sorted files are deleted after the merging has been done */
7
8     NEWMODE
9       record_type = STRUCT (
10                                key    INT ,
11                                name   CHAR (50));
12
13    DCL
14      flag        BOOL ,
15      infiles     ARRAY ( BOOL ) ACCESS record_type,
16      outfile     ACCESS record_type,
17      buffers     ARRAY ( BOOL ) record_type,
18      innames     ARRAY ( BOOL ) CHAR (10) INIT := ['FILE.IN.1 ','FILE.IN.2 '],
19      outname     CHAR (10) INIT := 'FILE.OUT   ',
20      inassocs    ARRAY ( BOOL ) ASSOCIATION ,
21      outassoc    ASSOCIATION ;
22
23    /* associate both sorted input files, connect an access to them for input */
24    /* and read their first record into a buffer                              */
25
26    DO
27      FOR curfile IN infiles,
28          curbuffer IN buffers,
29          curassoc IN inassocs,
30          curname IN innames;
31        CONNECT (curfile, ASSOCIATE (curassoc,curname), READONLY );
32        READRECORD (curfile, curbuffer);
33    OD ;
34
35    /* associate the output file, create a file for the association */
36    /* and connect an access to it for output                       */
37
38    ASSOCIATE (outassoc,outname);
39    CREATE (outassoc);
40    CONNECT (outfile, outassoc, WRITEONLY );
41    merge_files:
42      DO FOR EVER
43
44        /* determine which file, if any at all, to process next*/
45        /* 'flag' indicates the file                           */
46
47        CASE OUTOFFILE (infiles( FALSE )), OUTOFFILE (infiles( TRUE )) OF
48          ( TRUE ),  ( TRUE ):                    /* both files are empty */
49                     EXIT merge_files;
50          ( TRUE ),  ( FALSE ):                   /* one file is empty    */
51                     flag := TRUE ;
52          ( FALSE ), ( TRUE ):                    /* one file is empty    */
53                     flag := FALSE ;
54          ( FALSE ), ( FALSE ):                   /* no file is empty     */
55                     flag := buffers( FALSE ).key>buffers( TRUE ).key;
56        ESAC ;
57
58          /* output the buffer which currently contains a record with the */
```

```
59              /* smallest value for 'key', fill the buffer with a new record      */
60
61              WRITERECORD (outfile,buffers(flag));
62              READRECORD (infiles(flag), buffers(flag));
63      OD merge_files;
64
65      /* delete the input files and close the output file */
66
67      DO
68        FOR curassoc IN inassocs;
69          DELETE (curassoc);                    /* delete the file              */
70          DISSOCIATE (curassoc);                /* and terminate association    */
71      OD ;
72       DISSOCIATE (outassoc);                   /* disconnect and terminate     */
73
74    END merge_sorted_files;
```

22. Lire un fichier ayant des enregistrements de longueur variable

```
1   variable_length_records:
2    MODULE
3
4      /* This example shows how a file which consists of variable length   */
5      /* records can be treated.                                           */
6      /* The file consists of a number of strings of varying length; the   */
7      /* algorithm will read a string, allocate an appropriate location    */
8      /* for it, and put the reference to this location into a push down list */
9
10     NEWMODE
11       string = CHAR (80),
12       link_record = STRUCT (
13                                  next_record    REF link_record,
14                                  string_row     ROW string);
15
16     DCL
17       pushdownlist   REF link_record INIT := NULL ,
18       length         INT (1:80),
19       temporaryrow   ROW string,
20       fileaccess     ACCESS string DYNAMIC ,
21       association    ASSOCIATION ;
22
23     ASSOCIATE (association,'INPUT.DATA');          /* associate the input file   */
24     CONNECT (fileaccess, association, READONLY );  /* connect access for input   */
25    temporaryrow := READRECORD (fileaccess);        /* read the first record      */
26     DO                                             /* while not end-of-file      */
27       WHILE NOT(OUTOFFILE(fileaccess));
28         pushdownlist := ALLOCATE (link_record,     /* get a new link record      */
29                                   [pushdownlist, NULL ]);  /* and initialize it    */
30         length := 1 + UPPER (temporaryrow->);      /* determine length of string */
31          DO
32            WITH pushdowlist->;                     /* add new string to list     */
33             string_row := ALLOCATE ( CHAR (length), /* allocate space for string */
34                                      temporaryrow->);  /* and fill it              */
35          OD ;
36         temporaryrow := READRECORD (fileaccess);   /* get next record in file    */
37     OD ;
38     DISSOCIATE (association);                      /* end the association        */
39
40   END variable_length_records;
```

23. **L'emploi de spec de modules**

```
1   letter_count:
2   SPEC MODULE
3       /* This is a spec module for the corresponding module in example 8. */
4       SEIZE max;
5       count: PROC ( ROW CHAR (max) IN , ARRAY ('A':'Z') INT OUT ); END ;
6       GRANT count;
7   END letter_count;
8   test:
9   MODULE
10      /* This is the module 'test' from example 8.          */
11      /* It can now be piecewise compiled together with     */
12      /* the above spec module                              */
13      SYNMODE results = ARRAY ('A':'Z') INT ;
14      DCL c CHAR (10) INIT := 'A-B>ZAA9K' ' ';
15      SYN max = 10_000;
16      GRANT max;
17      SEIZE count;
18      count (-> c, output);
19      ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, ( ELSE ) : 0 ];
20  END test;
```

24. **Exemple de contexte**

```
1   CONTEXT
2       /* This is a context for the same module "test"   */
3       /* as used in example 23, allowing the piecewise  */
4       /* compilation of "test"                          */
5       count : PROC ( ROW CHAR (max) IN , ARRAY ('A':'Z') INT OUT ); END ;
6   END FOR
7
8   test :
9   MODULE
10      SYNMODE results = ARRAY ('A':'Z') INT ;
11      DCL c CHAR (10) INIT := 'A-B>ZAA9K' ' ';
12      SYN max = 10_000;
13      GRANT max;
14      SEIZE count;
15      count (-> c, output);
16      ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, ( ELSE ) : 0 ];
17  END test;
```

```
1     /* This example uses the module 'stacks_1' from example 10.    */
2     /* It shows how prefixes can be used to prevent name clashes.  */
3     /* It uses the remote construct to share source code.          */
4     /* It is assumed that the code of the module 'stacks_1' can    */
5     /* be referred to through the text reference name 'stack_code' */
6  char_stack:
7  MODULE
8     SYNMODE element = CHAR ;
9     MODULE REMOTE stack_code ;
10    GRANT ALL PREFIXED stack ! char ;
11 END char_stack ;
12
13 int_stack:
14 MODULE
15    SYNMODE element = INT ;
16    MODULE REMOTE stack_code ;
17    GRANT ALL PREFIXED stack ! int ;
18 END int_stack ;
19    /* Here 'push', 'pop' and 'element' are visible but            */
20    /* with prefixes 'stack ! char' and 'stack ! int' for          */
21    /* the implementations with element = CHAR and                 */
22    /* element = INT respectively.                                 */
23    /* Below are some possibilities of using the granted           */
24    /* names inside modules.                                       */
25 MODULE
26    SEIZE ALL PREFIXED stack ;
27    DCL c CHAR ;
28    int ! push (123) ;
29    char ! push ('a') ;
30    int ! pop ( ) ;
31    c = char ! elem (1) ;
32 END ;
33
34 MODULE
35    SEIZE (stack ! int -> stack) ! ALL ;
36    stack ! push (345) ;
37    stack ! pop ( ) ;
38 END ;
```

APPENDICE E: ENSEMBLE DE RÈGLES DE PRODUCTION

2 PRÉLIMINAIRES

2.2 VOCABULAIRE

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

2.4 COMMENTAIRES

<commentaire> ::=
    /* *<chaîne de caractères>* */

<chaîne de caractères> ::=
    { *<caractère>* } *

2.6 DIRECTIVES AU COMPILATEUR

<clause de directive> ::=
    <> *<directive>*{,< *directive*>}*[<>]

<directive> ::=
    <directive CHILL>
  | *<directive d'implémentation>*

<directive CHILL> ::=
    <directive de libération>

<directive de libération> ::=
    FREE (<liste de représentations textuelles de noms simples réservés>)
    <représentation textuelle de nom simple>{,*<représentation textuelle de nom simple>*} *

2.7 NOMS ET LEURS DÉFINITIONS

<nom> ::=
    <représentation textuelle de nom>

<représentation textuelle de nom> ::=
    <représentation textuelle de nom simple>
  | *<représentation textuelle de nom préfixe>*

<représentation textuelle de nom préfixe> ::=
    <préfixe> ! *<représentation textuelle de nom simple>*

<préfixe> ::=
    <préfixe simple> { ! *<préfixe simple>* } *

<préfixe simple> ::=
    <représentation textuelle de nom simple>

<définition> ::=
    <représentation textuelle de nom simple>

<liste de définitions> ::=
    <définitions >{, *<définitions>* }*

<*nom de champ*> ::=
<*représentation textuelle de nom simple*>

<*définition du nom de champ*> ::=
<*représentation textuelle de nom simple*>

<*liste de définitions de nom de champ*> ::=
<*définition de nom de champ*> {, <*définition de nom de champ*> } *

<*nom d'exception*> ::=
<*représentation textuelle de nom simple*>
| <*représentation textuelle de nom préfixe*>

<*nom de registre*> ::=
<*représentation textuelle de nom simple* >
| <*représentation textuelle de nom préfixe*>

<*nom de repère de texte*> ::=
<*représentation textuelle de nom simple*>
| <*représentation textuelle de nom préfixe*>

<*nom de repère d'implantation*> ::=
<*représentation textuelle de nom simple*>
| <*représentation textuelle de nom préfixe*>

3 MODES ET CLASSES

3.2 DÉFINITIONS DE MODES

3.2.1 Généralités

<*définition de mode*> ::=
<*liste de noms*> = <*mode définissant*>

<*mode définissant*> ::=
<*mode*>

3.2.2 Définitions de synmodes

<*énoncé de définition de synmodes*> ::=
SYNMODE <*définition de mode*> {, <*définition de mode*>}*;

3.2.3 Définitions des neumodes

<*énoncé de définition de neumodes*> ::=
NEWMODE <*définition de mode*> {, <*définition de mode*>} *;

3.3 CLASSIFICATION DES MODES

<*mode*> ::=
[**READ**] <*mode simple* >
| [**READ**] <*mode composé*>

<*mode simple*> ::=
<*mode discret*>
| <*mode ensembliste*>
| <*mode repère*>
| <*mode procédure*>

| <*mode exemplaire*>
| <*mode de synchronisation*>
| <*mode d'entrée-sortie*>

3.4 MODES DISCRETS

3.4.1 Généralités

<*mode discret*> ::=
<*mode entier*>
| <*mode booléen*>
| <*mode caractère*>
| <*mode ensemble*>
| <*mode intervalle*>

3.4.2 Modes entier

<*mode entier*> ::=
INT
| *BIN*
| <*nom de mode entier*>

3.4.3 Modes booléen

<*mode booléen*> ::=
BOOL
| <*nom de mode booléen*>

3.4.4 Modes caractère

<*mode caractère*> ::=
CHAR
| <*nom de mode caractère*>

3.4.5 Modes ensemble

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

<*extension d'ensemble*> ::=
<*extension d'ensemble avec numéros*>
| <*extension d'ensemble sans numéros*>

<*extension d'ensemble avec numéros*> ::=
<*élément d'ensemble avec numéros*> { ,<*élément d'ensemble avec numéros*>}*

<*élément d'ensemble avec numéro*> ::=
<*nom*> = <*nom littérale entière*>

<*extension d'ensemble sans numéros*> ::=
<*élément d'ensemble*> { ,<*élément d'ensemble*>} *

<*élément d'ensemble*> ::=
<*nom*>
| <*valeur anonyme*>

<*valeur anonyme*> ::=
*

3.4.6 Modes intervalle

<mode intervalle> ::=
<nom de mode discret>(<intervalle littéral>)
| **RANGE** *(<intervalle littéral>)*
| *BIN (<expression littérale entière >)*
| *<nom de mode intervalle>*

<intervalle littéral> ::=
<borne inférieure> : <borne supérieure>

<borne inférieure> ::=
<expression littérale discrète >

<borne supérieure> ::=
<expression littérale discrète >

3.5 MODES ENSEMBLISTE

<mode ensembliste> ::=
POWERSET *<mode primitif>*
| *<nom de mode ensembliste >*

<mode primitif> ::=
<mode discret>

3.6 MODES REPÈRE

3.6.1 Généralités

<mode repère> ::=
<mode repère lié>
| *<mode repère libre>*
| *<mode descripteur>*

3.6.2 Modes repère lié

<mode repère lié> ::=
REF *<mode repéré>*
| *<nom de mode repère lié >*

<mode repéré> ::=
<mode>

3.6.3 Modes repère libre

<mode repère libre> ::=
PTR
| *<nom de mode repère libre >*

3.6.4 Modes descripteur

<mode descripteur> ::=
ROW *<mode chaîne>*
| **ROW** *<mode rangée>*
| **ROW** *<nom de mode de structure variable>*
| *<nom de mode descripteur>*

3.7 MODES PROCÉDURE

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

<liste de paramètres> ::=
<spec de paramètre> { ,*<spec de paramètre>*} *

<spec de paramètre> ::=
<mode> [*<attribut de paramètre>*] [*<nom de registre>*]

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

<spec de résultat> ::=
[**RETURNS**] (*<mode>* [*<attribut de résultat>*] [*<nom de registre >*])

<attribut de résultat> ::=
[**NONREF**] **LOC** [**DYNAMIC**]

<liste d'exceptions> ::=
<nom d'exception> { ,*<nom d'exception>*} *

3.8 MODES EXEMPLAIRE

<mode exemplaire> ::=
INSTANCE
| *<nom de mode exemplaire >*

3.9 MODES SYNCHRONISATION

3.9.1 Généralités

<mode de synchronisation> ::=
<mode événement>
| *<mode tampon>*

3.9.2 Modes événement

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

<longueur d'événement> ::=
<expression littérale entière>

3.9.3 Modes tampon

<mode tampon> ::=
BUFFER [(*<longueur de tampon>*)]*<mode des éléments de tampon>*
| *<nom de mode tampon>*

<longueur de tampon> ::=
<expression littérale entière>

<mode des éléments de tampon> ::=
<mode>

3.10 MODES D'ENTRÉE-SORTIE

3.10.1 Généralités

<mode d'entrée-sortie> ::=
<mode association>
| *<mode accès>*

3.10.2 Modes association

<mode association> ::=
ASSOCIATION
| *<nom de mode association>*

3.10.3 Modes accès

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

<mode enregistrement> ::=
<mode>

<mode d'indice> ::=
<mode discret>
| *<intervalle de littéral>*

3.11 MODES COMPOSÉS

3.11.1 Généralités

<mode composé> ::=
<mode chaîne>
| *<mode rangée>*
| *<mode structure>*

3.11.2 Modes chaîne

<mode chaîne> ::=
<genre de chaîne>(<longueur de chaîne>)
| *<mode chaîne paramétré>*
| *<nom de mode chaîne>*

<mode chaîne paramétré> ::=
<nom de mode chaîne originel>(<longueur de chaîne>)
| *<nom de mode chaîne paramétré>*

<nom de mode chaîne originel> ::=
<nom de mode chaîne>

<genre de chaîne> ::=
CHAR
| *BIT*

<longueur de chaîne> ::=
<expression littérale entière>

3.11.3 Modes rangée

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

<mode rangée paramétré> ::=
<nom de mode rangée originel>(<indice supérieur>)
| *<nom de mode rangée paramétré>*

<nom de mode rangée originel> ::=
<nom de mode rangée>

<indice supérieur> ::=
<expression littérale>

<mode des éléments> ::=
<mode>

3.11.4 Modes structure

<mode structure> ::=
<mode structure emboîtée>
| *<mode structure étagée>*
| *<mode structure paramétré>*
| *<nom de mode structure>*

<mode structure emboîtée> ::=
STRUCT *(<champs>* { ,*<champs>*} **)*

<champs> ::=
<champs fixes>
| *<choix de champs>*

<champs fixes> ::=
<liste de définitions de noms de champ> *<mode>*
[*<implantation de champ>*]

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

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

<marqueurs> ::=
<nom de champ marqueur>{, *<nom de champ marqueur>*} *

<champs récurrents> ::=
    <liste de définitions de noms de champ> <mode>
    [<implantation de champ>]

<modé structure paramétré> ::=
    <nom de mode structure variable originel>
  (<liste d'expressions littérales>)
  | <nom de mode structureparamétré>

<nom de mode structure variable originel> ::=
    <nom de mode de structurevariable>

<liste d'expressions littérales> ::=
    <expression littérale> {,<expression littérale>} *

3.11.5 Notation étagée de structures

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

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

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

<champs à choisir de l'étage (n)> ::=
    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

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

<champs récurrents de l'étage (n)> ::=
    n <liste de noms> <mode>

[<*implantation de champ*>]
| *n* <*liste de noms* > [<*spécification de rangée*>]
[**READ**] [<*implantation de champ*>] { ,<*champs de l'étage (n+1)*>}$^{+}$

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

3.11.6 Description d'implantation pour modes rangée et modes structure

<*implantation d'élément*> ::=
PACK | **NOPACK** | <*pas*>

<*implantation de champ*> ::=
PACK | **NOPACK** | <*pos*>

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

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

<*mot*> ::=
<*expression littérale entière* >

<*taille de pas*> ::=
<*expression littérale entière* >

<*bit initial*> ::=
<*expression littérale entière* >

<*bit final*> ::=
<*expression littérale entière* >

<*longueur*> ::=
<*expression littérale entière* >

4 LOCUS ET LEURS ACCÈS

4.1 DÉCLARATIONS

4.1.1 Généralités

<*énoncé déclaratif*> ::=
DCL <*déclaration*> { ,<*déclaration*>} *;

<*déclaration*> ::=
<*déclaration de locus*>
| <*déclaration de loc-identité*>
| <*déclaration de locus avec base*>

4.1.2 Déclarations de locus

<*déclaration de locus*> ::=
<*liste de définitions*> <*mode*> [**STATIC**] [<*initialisation*>]

<*initialisation*> ::=
<*initialisation domaniale*>
| <*initialisation viagère*>

<*initialisation domaniale*> ::=
<*symbole d'affectation*> <*valeur*> [<*filet*>]

<*initialisation viagère*> ::=
INIT <*symbole d'affectation*> <*valeur constante*>

4.1.3 Déclarations de loc-identité

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

4.1.4 Déclarations de locus avec base

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

4.2 LES LOCUS

4.2.1 Généralités

<*locus*> ::=
<*nom d'accès*>
| <*repère lié dérepéré*>
| <*repère libre dérepéré*>
| <*rangée dérepéré*>
| <*élément de chaîne*>
| <*tranche de chaîne*>
| <*élément de rangée*>
| <*tranche de rangée*>
| <*champ de structure*>
| <*appel de procédure rendant locus*>
| <*appel d'opération prédéfinie rendant locus*>
| <*conversion de locus*>

4.2.2 Noms d'accès

<*nom d'accès*> ::=
<*nom de locus* >
| <*nom de loc-identité* >
| <*nom basé* >
| <*nom d'énumération de locus* >
| <*nom de locus faire-avec* >

4.2.3 Repères liés dérepérés

<*repère lié dérepéré*> ::=
<*valeur primitive repère lié* > -> [<*nom de mode* >]

4.2.4 Repères libres dérepérés

<repère libre dérepéré> ::=
<valeur primitive repère libre > -> <nom de mode >

4.2.5 Rangées dérepérées

<rangée dérepérée> ::=
<valeur primitive rangée > ->

4.2.6 Eléments de chaîne

<élément de chaîne> ::=
<locus chaîne > ('<élément de début>)

4.2.7 Tranches de chaîne

<tranche de chaîne> ::=
<locus chaîne > (<élément de gauche> : <élément de droite>)
| <locus chaîne > (<élément de début> **UP** *<taille de chaîne>)*

<élément de gauche> ::=
<expression littérale entière >

<élément de droite> ::=
<expression littérale entière >

<élément de début> ::=
<expression entière >

<taille de tranche> ::=
<expression entière >

4.2.8 Eléments de rangée

<élément de rangée> ::=
<locus rangée > (<liste d'expressions>)

<liste d'expressions> ::=
<expression> { , <expression>} *

4.2.9 Tranches de rangée

<tranche de rangée> ::=
<locus rangée >(<élément inférieur> : <élément supérieur>)
| <locus rangée > (<premier élément> **UP** *<taille de rangée>)*

<élément inférieur> ::=
<expression>

<élément supérieur> ::=
<expression>

<premier élément> ::=
<expression>

4.2.10 Champs de structure

<champ de structure> ::=
<locus structure > . <nom de champ >

4.2.11 Appels de procédure rendant locus

<*appel de procédure rendant locus*> ::=
<*appel de procédure rendant locus* >

4.2.12 Appels d'opération prédéfinie rendant locus

<*appel d'opération prédéfinie rendant locus*> ::=
<*appel d'opération prédéfinie par l'implémentation rendant locus*>
| <*appel d'opération prédéfinie par CHILL rendant locus*>

<*appel d'opération prédéfinie par CHILL rendant locus*> ::=
<*io CHILL appel d'opération prédéfinie par io CHILL rendant locus*>

4.2.13 Conversions de locus

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

5 VALEURS ET LEURS OPÉRATIONS

5.1 DÉFINITIONS DE SYNONYMES

<*énoncé de définition de synonymes*> ::=
SYN <*définition de synonyme*> { ,<*définition de synonyme*>} *;

<*définition de synonyme*> ::=
<*liste de définitions*> [<*mode*>] = <*valeur constante* >

5.2 VALEUR PRIMITIVE

5.2.1 Généralités

<*valeur primitive*> ::=
<*contenu de locus*>
| <*nom de valeur*>
| <*littéral*>
| <*multiplet*>
| <*valeur élément de chaîne*>
| <*valeur tranche de chaîne*>
| <*valeur élément de rangée*>
| <*valeur tranche de rangée*>
| <*valeur champ de structure*>
| <*conversion d'expression*>
| <*appel de procédure rendant valeur*>
| <*appel d'opération prédéfinie rendant valeur*>
| <*expression démarrer*>
| <*opérateur nullaire*>
| <*expression parenthésée*>

5.2.2 Contenu de locus

<*contenu de locus*> ::=
<*locus*>

5.2.3 Noms de valeur

<nom de valeur> ::=
<nom de synonyme >
| <nom d'énumération de valeur >
| <nom de valeur faire-avec >
| <nom de valeur reçue >
| <nom de procédure générale >

5.2.4 Littéraux

5.2.4.1 Généralités

<littéral> ::=
<littéral d'entier>
| <littéral de booléen>
| <littéral d'ensemble>
| <littéral de vide>
| <littéral de chaîne de caractères>
| <littéral de chaîne de bits>

5.2.4.2 Littéraux d'entier

<littéral d'entier> ::=
<littéral décimal d'entier>
| <littéral binaire d'entier>
| <littéral octal d'entier>
| <littéral hexadécimal d'entier>

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

<littéral binaire d'entier> ::=
B' { 0 | 1 | _} $^{+}$

<littéral octal d'entier> ::=
O' { <chiffre octal> | _} $^{+}$

<littéral hexadécimal d'entier> ::=
H' { <chiffre hexadécimal> | _} $^{+}$

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

<chiffre hexadécimal> ::=
<chiffre> | A | B | C | D | E | F

<chiffre octal> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

5.2.4.3 Littéraux de booléen

<littéral de booléen> ::=
FALSE | TRUE

5.2.4.4 Littéraux d'ensemble

<littéral d'ensemble> ::=
<nom d'élément d'ensemble >

5.2.4.5 Littéral de vide

<littéral de vide> ::=
NULL

5.2.4.6 Littéraux de chaîne de caractères

<littéral de chaîne de caractères> ::=
' { <caractère non apostrophe > | <apostrophe>}'*
| C' { <chiffre octal> <chiffre hexadécimal> | _ }'*

<caractère> ::=
<lettre>
| <chiffre>
| <symbole>
| <espace>

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

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

<espace> ::=
SP

<apostrophe> ::=
' '

5.2.4.7 Littéraux de chaîne de bits

<littéral de chaîne de bits> ::=
<littéral binaire de chaîne de bits >
| <littéral octal de chaîne de bits >
| <littéral hexadécimal de chaîne de bits >

<littéral binaire de chaîne de bits> ::=
*B' { 0 | 1 | _} *'*

<littéral octal de chaîne de bits> ::=
*O' { <chiffre octal> | _ } *'*

<littéral hexadécimal de chaîne de bits > ::=
*H' { <chiffre hexadécimal> | _ } *'*

5.2.5 Multiplets

<multiplet> ::=
[<nom de mode >] (: { <multiplet ensembliste> | <multiplet de rangée>
| <multiplet de structure>} :)
| <littéral de chaîne de caractères>
| <littéral de chaîne de bits>

<multiplet ensembliste> ::=
*[{ <expression> | <intervalle>} { , { <expression> | <intervalle>} } *]*

<intervalle> ::=
<expression> : <expression>

<multiplet de rangée> ::=
<multiplet de rangée sans indices>
| <multiplet de rangée avec indices>

<multiplet de rangée sans indices> ::=
<valeur> { ,*<valeur>*} *

<multiplet de rangée avec indices> ::=
<liste d'étiquettes de cas> : *<valeur>* { , *<liste d'étiquettes de cas>* : *<valeur>*} *

<multiplet de structure> ::=
<multiplet de structure sans noms de champ>
| *<multiplet de structure avec noms de champ>*

<multiplet de structure sans noms de champ> ::=
<valeur> { ,*<valeur>*} *

<multiplet de structure avec noms de champ> ::=
<liste de noms de champ> : *<valeur>* { , *<liste de noms de champ>* : *<valeur>*} *

<liste de noms de champ> ::=
.*<nom de champ>* { , .*<nom de champ>* } *

5.2.6 Valeurs élément de chaîne

<valeur élément de chaîne> ::=
<valeur primitive chaîne>(<élément de début>)

5.2.7 Valeurs tranche de chaîne

<valeur tranche de chaîne> ::=
< valeur primitive chaîne>(<élément de gauche> : <élément de droite>)
| *< valeur primitive chaîne>(<élément de début>* **UP** *<taille de chaîne>)*

5.2.8 Valeurs élément de rangée

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

5.2.9 Valeurs tranche de rangée

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

5.2.10 Valeurs champ de structure

<valeur champ de structure> ::=
< valeur primitive structure> . <nom de champ>

5.2.11 Conversions d'expression

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

5.2.12 Appels de procédure rendant valeur

<appel de procédure rendant valeur> ::=
< appel de procédure rendant valeur>

5.2.13 Appels d'opération prédéfinie rendant valeur

<appel d'opération prédéfinie rendant valeur> ::=
< appel d'opération prédéfinie par l'implémentation rendant valeur>
| <appel d'opération prédéfinie par CHILL rendant valeur >

<appel d'opération prédéfinie par CHILL rendant valeur > ::=
NUM (< expression discrète>)
| *PRED* (< expression discrète>)
| *SUCC* (< expression discrète>)
| *ABS* (< expression entière>)
| *CARD* (< expression ensembliste>)
| *MAX* (< expression ensembliste>)
| *MIN* (< expression ensembliste>)
| *SIZE* ({ < nom de mode> | < locus de mode statique>})
| *UPPER* (<argument pour upper lower >)
| *LOWER* (<argument pour upper lower >)
| *GETSTACK* (<argument pour getstack > [, <valeur>])
| *ALLOCATE* (<argument pour allocate > [,
<valeur>])

<argument pour getstack> ::=
<argument>

<argument pour allocate> ::=
<argument>

<argument> ::=
< nom de mode>
| < nom de mode rangée>(<expression>)
| < nom de mode chaîne>(< expression d'entier >)
| <nom de mode structure récurrente >(<liste d'expressions >)

<argument pour upper lower> ::=
< locus rangée>
| <valeur primitive rangée >
| < nom de mode rangée>
| < locus chaîne>
| <valeur primitive chaîne >
| < nom de mode chaîne>
| < locus discret>
| < expression discrète>
| < nom de mode discret>

5.2.14 Expressions démarrer

<expression démarrer> ::=
START <nom de processus> ([<liste de paramètres effectifs>])

5.2.15 Opérateur nullaire

<opérateur nullaire> ::=
THIS

5.2.16 Expressions parenthésées

<expression parenthésée> ::=
(<expression>)

5.3 VALEURS ET EXPRESSIONS

5.3.1 Généralités

<valeur> ::=
<expression>
| *<valeur indéfinie>*

<valeur indéfinie> ::=
| *< nom de synonyme indéfini>*

5.3.2 Expressions

<expression> ::=
<opérande-1>
| *<sous-expression>* { *OR* | *XOR* } *<opérande-1>*

<sous-expression> ::=
<expression>

5.3.3 Opérande-1

<opérande-1> ::=
<opérande-2>
| *<sous-opérande-1> AND <opérande-2>*

<sous-opérande-1> ::=
<opérande-1>

5.3.4 Opérande-2

<opérande-2> ::=
<opérande-3>
| *<sous-opérande-2> <opérateur-3> <opérande-3>*

<sous-opérande-2> ::=
<opérande-2>

<opérateur-3> ::=
<opérateur relationnel>
| *<opérateur d'appartenance>*
| *<opérateur d'inclusion ensembliste>*

<opérateur relationnel> ::=
= | /= | > | >= | < | <=

<opérateur d'appartenance> ::=
IN

<opérateur d'inclusion ensembliste> ::=
<= | >= | < | >

5.3.5 Opérande-3

<opérande-3> ::=
<opérande-4>
| *<sous-opérande-3> <opérateur-4> <opérande-4>*

<sous-opérande-3> ::=
<opérande-3>

<opérateur-4> ::=
<opérateur arithmétique additif >
| <opérateur de concaténation de chaîne >
| <opérateur de différence ensembliste >

<opérateur arithmétique additif > ::=
+ | -

<opérateur de concaténation de chaîne > ::=
//

<opérateur de différence ensembliste > ::=
-

5.3.6 Opérande-4

<opérande-4> ::=
<opérande-5>
| <sous-opérande-4> <opérateur arithmétique multiplicatif> <opérande-5>

<sous-opérande-4> ::=
<opérande-4>

<opérateur arithmétique multiplicatif> ::=
* | / | *MOD* | *REM*

5.3.7 Opérande-5

<opérande-5> ::=
[<opérateur unaire>] <opérande-6>

<opérateur unaire> ::=
- | *NOT*
| <opérateur de répétition de chaîne>

<opérateur de répétition de chaîne> ::=
(< expression littérale entière>)

5.3.8 Opérande-6

<opérande-6> ::=
<locus repéré >
| <expression recevoir >
| <valeur primitive >

<locus repéré> ::=
-> <locus>
| **ADDR** (<locus>)

<expression recevoir> ::=
RECEIVE <locus tampon>

6 ACTIONS

6.1 GÉNÉRALITÉS

<énoncé d'action> ::=
[<définition > :] <action> [<filet>] [<représentation textuelle de nom simple>];
| <module>
| <module de spec>

<*action*> ::=
<*action parenthésée*>
| <*action d'affectation*>
| <*action appeler*>
| <*action sortir*>
| <*action revenir*>
| <*action résulter*>
| <*action aller*>
| <*action affirmer*>
| <*action vide*>
| <*action démarrer*>
| <*action arrêter*>
| <*action mettre en attente*>
| <*action continuer*>
| <*action envoyer*>
| <*action causer*>

<*action parenthésée*> ::=
<*action conditionnelle*>
| <*action de cas*>
| <*action faire*>
| <*bloc début-fin*>
| <*action mettre en attente et choisir*>
| <*action recevoir et choisir*>

6.2 ACTION D'AFFECTATION

<*action d'affectation*> ::=
<*action d'affectation simple*>
| <*action d'affectation multiple*>

<*action d'affectation simple*> ::=
<*locus*> { <*symbole d'affectation*> | <*opérateur affectant*>} <*valeur*>

<*action d'affectation multiple*> ::=
<*locus*> { ,<*locus*>}$^{+}$ <*symbole d'affectation*> <*valeur*>

<*opérateur affectant*> ::=
<*opérateur binaire fermé*> <*symbole d'affectation*>

<*opérateur binaire fermé*> ::=
OR | *XOR*
| *AND*
| <*opérateur de différence ensembliste*>
| <*opérateur arithmétique additif*>
| <*opérateur arithmétique multiplicatif*>

<*symbole d'affectation*> ::=
:= | =

6.3 ACTION CONDITIONNELLE

<*action conditionnelle*> ::=
IF < *expression booléenne*> <*clause alors*> [<*clause sinon*>] **FI**

<*clause alors*> ::=
THEN <*liste d'énoncés d'action*>

<*clause sinon*> ::=
ELSE <*liste d'énoncés d'action*>
| **ELSIF** < *expression booléenne*> <*clause alors*> [<*clause sinon*>]

6.4 ACTION DE CAS

<*action de cas*> ::=
CASE <*liste de sélecteurs de cas*> **OF** [<*liste d'intervalles*>;] { <*cas à choisir*>} $^{+}$
[**ELSE** <*liste d'énoncés d'action*>]
ESAC

<*liste de sélecteur de cas*> ::=
< *expression discrète*> { ,< *expression discrète*>} *

<*liste d'intervalles*> ::=
<*mode discret*> { ,< *mode discret*>} *

<*cas à choisir*> ::=
<*spécification d'étiquettes de cas*> : <*liste d'énoncés d'action*>

6.5 ACTION FAIRE

6.5.1 Généralités

<*action faire*> ::=
DO [<*commande* >;] <*liste d'énoncés d'action*> **OD**

<*commande*> ::=
<*commande pour*> [<*commande tandis*>]
| <*commande tandis*>
| <*partie avec*>

6.5.2 Commande pour

<*commande pour*> ::=
FOR { <*itération*> { ,<*itération*>} * | **EVER** }

<*itération*> ::=
<*énumération de valeur*>
| <*énumération de locus*>

<*énumération de valeur*> ::=
<*énumération par pas*>
| <*énumération par intervalle*>
| <*énumération ensembliste*>

<*énumération par pas*> ::=
<*compteur de boucle*> <*symbole d'affectation*>
<*valeur initiale*> [<*valeur de pas*>] [**DOWN**] <*valeur finale*>

<*compteur de boucle*> ::=
<*définition* >

<*valeur initiale*> ::=
< *expression discrète*>

<*valeur de pas*> ::=
BY < *expression entière*>

<*valeur finale*> ::=
TO < *expression discrète*>

<*énumération par intervalle*> ::=
<*compteur de boucle*> [**DOWN**] **IN** < *mode discret*>

<énumération ensembliste> ::=
<compteur de boucle> [**DOWN**] **IN** < expression ensembliste>

<énumération de locus> ::=
<compteur de boucle> [**DOWN**] **IN** <locus composite >

<locus composite> ::=
< locus rangée>
| < locus chaîne>

6.5.3 Commande tandis

<commande tandis> ::=
WHILE < expression booléenne>

6.5.4 Partie avec

<partie avec> ::=
WITH <commande avec> { ,<commande avec>} *

<commande avec> ::=
< locus structure>
| < valeur primitive structure>

6.6 ACTION SORTIR

<action sortir> ::=
EXIT <représentation textuelle de nom simple >

6.7 ACTION APPELER

<action appeler> ::=
[**CALL**] { <appel de procédure>
| <appel d'opération prédéfinie CHILL>
| < appel d'opération prédéfinie d'implémentation>}

<appel procédure> ::=
{ < nom de procédure > | <valeur primitive procédure >}
([<liste de paramètres effectifs>])

<liste de paramètres effectifs> ::=
<paramètre effectif> { ,<paramètre effectif>} *

<paramètre effectif> ::=
<valeur>
| <locus>

<appel d'opération prédéfinie CHILL> ::=
<appel d'opération prédéfinie de valeur CHILL >
| <appel d'opération prédéfinie de locus CHILL >
| <appel d'opération prédéfinie simple CHILL >

<appel d'opération prédéfinie simple CHILL > ::=
TERMINATE (< expression repère>)
| <appel d'opération prédéfinie simple io CHILL >

6.8 ACTION RÉSULTER ET ACTION REVENIR

<action revenir> ::=
RETURN [<résultat>]

<*action résulter*> ::=
RESULT <*résultat*>

<*résultat*> ::=
<*valeur*>
| <*locus*>

6.9 ACTION ALLER

<*action aller*> ::=
GOTO <*représentation textuelle de nom simple*>

6.10 ACTION AFFIRMER

<*action affirmer*> ::=
ASSERT <*expression booléenne*>

6.11 ACTION VIDE

<*action vide*> ::=
<*vide*>

<*vide*> ::=

6.12 ACTION CAUSER

<*action causer*> ::=
CAUSE <*nom d'exception* >

6.13 ACTION DÉMARRER

<*action démarrer*> ::=
<*expression démarrer*> [**SET** <*locus exemplaire*>]

6.14 ACTION ARRÊTER

<*action arrêter*> ::=
STOP

6.15 ACTION CONTINUER

<*action continuer*> ::=
CONTINUE <*locus événement*>

6.16 ACTION METTRE EN ATTENTE

<*action mettre en attente*> ::=
DELAY < *locus événement*> [<*priorité*>]

<*priorité*> ::=
PRIORITY < *expression littérale entière*>

6.17 ACTION METTRE EN ATTENTE ET CHOISIR

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

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

<liste d'événements> ::=
< locus événement> { ,< locus événement>} ***

6.18 ACTION ENVOYER

6.18.1 Généralités

<action envoyer> ::=
<action envoyer signal>
| <action envoyer tampon>

6.18.2 Action envoyer signal

<action envoyer signal> ::=
SEND *< nom de signal> [(<valeur> { ,<valeur>}* *)]*
[**TO** *< valeur primitive exemplaire >] [<priorité>]*

6.18.3 Action envoyer tampon

<action envoyer tampon> ::=
SEND *< locus tampon>(<valeur>) [<priorité>]*

6.19 ACTION RECEVOIR ET CHOISIR

6.19.1 Généralités

<action recevoir et choisir> ::=
<action recevoir signal et choisir >
| <action recevoir tampon et choisir >

6.19.2 Action recevoir signal et choisir

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

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

6.19.3 Action recevoir tampon et choisir

<action recevoir tampon et choisir> ::=
RECEIVE CASE [**SET** *<locus exemplaire>;*]
{ <tampon à choisir >} +
[**ELSE** *<liste d'énoncés d'action >*]
ESAC

<tampon à choisir> ::=
(< locus tampon> **IN** *<définition>) : <liste d'énoncés d'action >*

7.4 OPÉRATIONS PRÉDÉFINIES POUR ENTRÉE-SORTIE

7.4.1 Généralités

<appel d'opération prédéfinie d'e/s rendant valeur de CHILL > ::=
< appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association>
| < appel d'opération prédéfinie d'e/s rendant valeur de CHILL est associé>
| < appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'accès>
| < appel d'opération prédéfinie d'e/s rendant valeur de CHILL lire article>

<appel d'opération prédéfinie d'e/s simple de CHILL > ::=
< appel d'opération prédéfinie d'e/s simple de CHILL désassocier>
| < appel d'opération prédéfinie d'e/s simple de CHILL modification>
| < appel d'opération prédéfinie d'e/s simple de CHILL connecter>
| < appel d'opération prédéfinie d'e/s simple de CHILL déconnecter>
| < appel d'opération prédéfinie d'e/s simple de CHILL écrire article>

<appel prédéfini d'e/s rendant locus de CHILL > ::=
< appel d'opération prédéfinie d'e/s rendant locus de CHILL >

7.4.2 Association avec un objet du monde extérieur

<appel d'opération prédéfinie d'e/s rendant locus de CHILL associer> ::=
ASSOCIATE (< locus association>[,<liste de paramètres pour associer>])

<appel d'opération prédéfinie d'e/s rendant valeur de CHILL est associé> ::=
ISASSOCIATED (< locus association>)

<liste de paramètres pour associer> ::=
*<paramètre pour associer> { ,<paramètre pour associer> } **

<paramètre pour associate> ::=
<locus>
| <valeur>

7.4.3 Dissociation d'un objet du monde extérieur

< appel d'opération prédéfinie d'e/s simple de CHILL désassocier > ::=
DISSOCIATE (< locus association>)

7.4.4 Accès aux attributs association

< appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association> ::=
EXISTING (< locus association>)
| READABLE (< locus association>)
| WRITEABLE (< locus association>)
| INDEXABLE (< locus association>)
| SEQUENCIBLE (< locus association>)
| VARYING (< locus association>)

7.4.5 Modification des attributs association

< appel d'opération prédéfinie d'e/s simple de CHILL modification> ::=
CREATE (< locus association>)
| DELETE (< locus association>)
| MODIFY (< locus association>[,<liste de paramètres pour modifier>])

<*liste de paramètres pour modifier*> ::=
        <*paramètre pour modifier*> { , <*paramètre pour modifier*> } *

<*paramètre pour modifier*> ::=
        <*valeur*>
      | <*locus*>

7.4.6 Connection d'un locus accès

< *appel d'opération prédéfinie d'e/s simple de CHILL connecter*> ::=
        CONNECT (< *locus accès*>,< *locus association*>, <*expression usage*>
[
      ,<*expression positionnement* >[,<*expression indice* >]])

<*expression usage*> ::=
        <*expression*>

<*expression positionnement*> ::=
        <*expression*>

<*expression indice*> ::=
        <*expression*>

7.4.7 Déconnexion d'un locus accès

< *appel d'opération prédéfinie d'e/s simple de CHILL déconnecter*> ::=
        DISCONNECT (< *locus accès*>*)*

7.4.8 Attributs d'accès de locus accès

< *appel d'opération prédéfinie d'e/s rendant valeur de CHILL attribut d'association*> ::=
        GETASSOCIATION (< *locus accès*>*)*
      | *GETUSAGE (*< *locus accès*>*)*
      | *OUTOFFILE (*< *locus accès*>*)*

7.4.9 Opérations de transfert de données

< *appel d'opération prédéfinie d'e/s rendant valeur de CHILL lire article*> ::=
        READRECORD (< *locus accès*>[,<*expression d'indice*>] [,<*locus de lecture*>])

< *appel d'opération prédéfinie d'e/s simple de CHILL écrire article*> ::=
        WRITERECORD (< *locus accès*>[,<*expression d'indice*>], <*expression écrire*>*)*

<*locus de lecture*> ::=
        < *locus de mode statique*>

<*expression écrire*> ::=
        <*expression*>

8 STRUCTURE DE PROGRAMME

8.2 DOMAINES ET IMBRICATION

<*corps de bloc*> ::=
        <*liste d'énoncés informatifs*> <*liste d'énoncés d'action*>

<*corps de procédure*> ::=
        <*liste d'énoncés informatifs*> { <*énoncé d'action*> | <*énoncé d'entrée*>} *

<corps de processus> ::=
<liste d'énoncés informatifs> <liste d'énoncés d'action>

<corps de module> ::=
{ <énoncé informatif> | <énoncé de visibilité> | <région> | <spec de région> } *
<liste d'énoncés d'action>

<corps de région> ::=
{ <énoncé informatif> | <énoncé de visibilité>} *

<corps de spec de module> ::=
{ <quasi énoncé informatif> | <énoncé de visibilité> | <quasi module>
| <spec de module> | <quasi région> | <spec de région> | <quasi action causer> } *

<corps de spec de région> ::=
*{ <quasi énoncé informatif> | <énoncé de visibilité> | <quasi action causer> }**

<corps de contexte> ::=
{ <quasi énoncé informatif> | <énoncé de visibilité> | <quasi module> |
<spec de module> | <quasi région> | <spec de région> } *

<corps de quasi module> ::=
{ <quasi énoncé informatif> | <énoncé de visibilité> | <quasi module> |
*<spec de module> | <quasi région> | <spec de région> }**

<quasi corps de région> ::=
{ <quasi énoncé informatif> | <énoncé de visibilité> } *

<liste d'énoncés d'action> ::=
{ <énoncé d'action>} *

<liste d'énoncés informatifs> ::=
{ <énoncé informatif>} *

<énoncé informatif> ::=
<énoncé déclaratif>
| <énoncé définissant>

<énoncé définissant> ::=
<énoncé de définition de synmodes>
| <énoncé de définition de neumodes>
| <énoncé de définition de synonymes>
| <énoncé de définition de procédure>
| <énoncé de définition de processus>
| <énoncé de définition de signal>
| <vide>;

8.3 BLOCS DÉBUT-FIN

<bloc début-fin> ::=
BEGIN *<corps de bloc>* **END**

8.4 DÉFINITIONS DE PROCÉDURE

<énoncé de définition de procédure> ::=
<définition > : <définition de procédure>
[<filet>] [<représentation textuelle de nom simple>];

<définition de procédure> ::=
PROC *([<liste de paramètres formels>])* *[<spec de résultat>]*
[**EXCEPTIONS** *(<liste d'exceptions>)] <attributs de procédure>;*
<corps de procédure> **END**

<*liste de paramètres formels*> ::=
<*paramètre formel*> { ,<*paramètre formel*>} *

<*paramètre formel*> ::=
<*liste de définitions*> <*spec de paramètre*>

<*attributs de procédure*> ::=
<*généralité*>] [**RECURSIVE**]

<*généralité*> ::=
GENERAL
| **SIMPLE**
| **INLINE**

<*énoncé d'entrée*> ::=
<*définition* > : <*définition d'entrée*>;

<*définition d'entrée*> ::=
ENTRY

8.5 DÉFINITIONS DE PROCESSUS

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

<*définition de processus*> ::=
PROCESS ([<*liste de paramètres formels*>]); <*corps de processus*> **END**

8.6 MODULES

<*module*> ::=
[<*contexte*>] [<*définition*>:]
MODULE <*corps de module*> **END** [<*filet*>] [<*représentation textuelle de nom simple*>];
| [<*contextes*>] <*module distant*>

8.7 RÉGIONS

<*région*> ::=
[<*contextes*>] [<*définition*> :] **REGION** <*corps de région*> **END**
[<*filet*>] [<*représentation textuelle de nom simple*>];
| [<*contextes*>] <*région distante*>;

8.8 PROGRAMMES

<*programme*> ::=
{ <*module*> | <*spec de module*> | <*région*> | <*spec de région*>} +

8.10 CONSTRUCTIONS POUR LA PROGRAMMATION PAR FRAGMENTS

8.10.1 Fragments distants

<*module distant*> ::=
[<*représentation textuelle de nom simple*>:]
MODULE REMOTE <*indicateur de texte d'origine*>;

<*région distante*> ::=
[<*représentation textuelle de*
REGION REMOTE *nom simple*>:] **REGION REMOTE** <*indicateur de texte d'origine*>;

<*spec de module distante*> ::=
[<*représentation textuelle de nom simple*>:]
SPEC MODULE REMOTE <*indicateur de texte d'origine*>;

<*spec de région distante*> ::=
[<*représentation textuelle de nom simple*>:]
SPEC REGION REMOTE <*indicateur de texte d'origine*>;

<*contexte distant*> ::=
CONTEXT REMOTE <*indicateur de texte d'origine*> **FOR**

<*indicateur de texte d'origine*> ::=
<*littéral de chaîne de caractères*>
| <*nom repère de texte*>
| <*vide*>

8.10.2 Spec de modules, spec de régions et contextes

<*spec de module*> ::=
[<*contextes*>] [<*représentation textuelle de nom simple*> :] **SPEC MODULE**
<*corps de spec de module*> **END** [<*représentation textuelle de nom simple*>];
| <*spec de module distante*>

<*spec de région*> ::=
[<*contextes*>] [<*représentation textuelle de nom simple*> :] **SPEC REGION**
<*corps de spec de région*> **END** [<*représentation textuelle de nom simple*>];
| <*spec région distante*>

<*contextes*> ::=
<*contexte*> { <*contexte*> } *

<*contexte*> ::=
CONTEXT <*corps de contexte*> **END** [<*quasi filet*>] **FOR**
| <*contexte distant*>

8.10.3 Quasi énoncés

<*quasi énoncé informatif*> ::=
<*quasi énoncé déclaratif*>
| <*quasi énoncé définissant*>

<*quasi énoncé déclaratif*> ::=
DCL <*quasi déclaration*> {, <*quasi déclaration*> } *;

<*quasi déclaration*> ::=
<*définition*> <*mode*>
[**STATIC**] [**NONREF**] [**DYNAMIC**]

<*quasi énoncé définissant*> ::=
<*énoncé de définition de synmode*>
| <*énoncé de dfinition de neumode*>
| <*énoncé de définition de synonyme*>
| <*quasi énoncé de définition de procédure*>
| <*quasi énoncé de définition de processus*>
| <*énoncé de définition de signal*>
| <*vide*>;

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

<quasi énoncé d'entrée> ::=
<définition> : **ENTRY** ;

<quasi liste de paramètres formels> ::=
<quasi paramètre formel> {, *<quasi paramètre formel>* } *

<quasi paramètre formel> ::=
[*<représentation textuelle de nom simple>*
{, *<représentation textuelle de nom simple>* } *] *<spec de paramètre>*

<quasi énoncé de définition de processus> ::=
<définition> : **PROCESS** ([*<quasi liste de paramètre formel>*])
END [*<représentation textuelle de nom simple>*];

<quasi région> ::=
[*<définition>* :] **REGION** *<quasi corps de région>*
END [*<représentation textuelle de nom simple>*];

<quasi module> ::=
[*<définition>* :] **MODULE** *<quasi corps de module>*
END [*<représentation textuelle de nom simple>*];

<quasi action causer> ::=
CAUSE *<liste d'exceptions>*;

<quasi filet> ::=
ON ELSE END
| **ON** *<liste d'exceptions>* [**ELSE**] **END**

9 EXÉCUTION CONCURRENTE

9.5 ENONCÉ DE DÉFINITION DE SIGNAL

<énoncé de définition de signal> ::=
SIGNAL *<définition de signal>* { ,*<définition de signal>*} *;

<définition de signal> ::=
<définition> [= (*<mode>* { ,*<mode>*} *)] [**TO** *<nom de processus>*]

10 PROPRIÉTÉS SÉMANTIQUES GÉNÉRALES

10.1 VÉRIFICATION DE MODES

10.1.3 Sélection de cas

<spécification d'étiquettes de cas> ::=
<liste d'étiquettes de cas> { ,*<liste d'étiquettes de cas>*} *

<liste d'étiquettes de cas> ::=
(*<étiquette de cas>* { ,*<étiquette de cas>*} *)
| *<indifférent>*

<étiquette de cas> ::=
<
expression littérale discrète>
| *<intervalle littéral>*
| < *nom de mode discret*>
| **ELSE**

<indifférent> ::=
(*)

10.2 VISIBILITÉ ET IDENTIFICATION

10.2.4 Visibilité dans les domaines

10.2.4.2 Enoncés de visibilité

<énoncé de visibilité> ::=
<énoncé d'octroi>
| <énoncé de saisie>

10.2.4.3 Clause renommer préfixe

<clause renommer préfixe> ::=
(<ancien préfixe> -> <nouveau préfixe>) ! <postfixe>

<ancien préfixe> ::=
<préfixe>
| <vide>

<nouveau préfixe> ::=
<préfixe>
| <vide>

<postfixe> ::=
*<postfixe de saisie> {,<postfixe de saisie>}**
*| <postfixe d'octroi> {,<postfixe d'octroi>}**

10.2.4.4 Enoncé d'octroi

<énoncé d'octroi> ::=
GRANT *<clause renommer préfixe>* {,*<clause renommer préfixe>*}*
[[**DIRECTLY**] **PERVASIVE**] ;
| **GRANT** *<fenêtre d'octroi>* [*<clause préfixe>*]
[[**DIRECTLY**] **PERVASIVE**] ;

<fenêtre d'octroi> ::=
<postfixe d'octroi> { , *<postfixe d'octroi>* }*

<postfixe d'octroi> ::=
<représentation textuelle de nom>
| <représentation textuelle de nom de neumode > <clause d'interdiction >
| [*<préfixe> !*] **ALL**

<clause préfixe> ::=
PREFIXED [*<préfixe>*]

<clause d'interdiction> ::=
FORBID { *<liste de noms d'interdiction>* | **ALL** }

<liste de noms d'interdiction> ::=
(<nom de champ> { , *<nom de champ>*} * *)*

10.2.4.5 Enoncé de saisie

<énoncé de saisie> ::=
SEIZE *<clause renommer préfixe>* { , *<clause renommer préfixe>*}*;

| **SEIZE** <*fenêtre de saisie*> [<*clause de préfixe*>] ;

<*fenêtre de saisie*> ::=
<*postfixe de saisie*> { , <*postfixe de saisie*> }*

<*postfixe de saisie*> ::=
<*représentation textuelle de nom*>
| <*représentation textuelle de nom de modulion*> **ALL**
| [<*préfixe*> *!*] **ALL**

<*représentation textuelle de nom de modulion*> ::=
< *représentation textuelle de nom de modulion*>

11 FILETS D'EXCEPTION

11.2 FILETS

<*filet*> ::=
ON { <*choix d'exceptions*>}* [**ELSE** <*liste d'énoncés d'action*>] **END**

<*choix d'exceptions*> ::=
(<*liste d'exceptions*>) : <*liste d'énoncés d'action*>

12 OPTIONS POUR L'IMPLÉMENTATION

12.1 OPÉRATIONS PRÉDÉFINIES

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

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

<*paramètre d'opération prédéfinie*> ::=
<*valeur*>
| <*locus*>
| <*nom non réservé*>

APPENDICE F: INDEX DES RÈGLES DE PRODUCTION

non terminal	défini dans la section	page	employé page(s)
<argument pour allocate>	5.2.13	65	65
<argument pour getstack>	5.2.13	65	65
<argument pour upper lower>	5.2.13	65	65
<attribut de paramètre>	3.7	23	23
<attribut de résultat>	3.7	23	23
<attributs de procédure>	8.4	111	110,118
<bit final>	3.11.6	36	36
<bit initial>	3.11.6	36	36
<bloc début-fin>	8.3	110	77
<borne inférieure>	3.4.6	20	20
<borne supérieure>	3.4.6	20	20
<caractère>	5.2.4.6	55	9,55
<cas à choisir>	6.4	79	79
<chaine de caractères>	2.4	9	9
<champ de structure>	4.2.10	49	43
<champs>	3.11.4	29	29
<champs à choisir>	3.11.4	30	30
<champs à choisir de l'étage (n)>	3.11.5	34	34
<champs de l'étage>			34
<champs de l'étage (n)>	3.11.5	34	
<champs de l'étage (n+1)>			34,35
<champs fixes>	3.11.4	30	30
<champs fixes de l'étage (n)>	3.11.5	34	34
<champs recurrents>	3.11.4	30	30
<champs recurrents de l'étage (n)>	3.11.5	34	34
<chiffre>	5.2.4.2	54	8,54,55
<chiffre hexadécimal>	5.2.4.2	54	54,55,56
<chiffre octal>	5.2.4.2	54	54,55,56
<choix de champs>	3.11.4	30	30
<choix de champs de l'étage (n)>	3.11.5	34	34
<choix d'exceptions>	11.2	152	152
<clause alors>	6.3	79	79
<clause de directive>	2.6	9	
<clause de préfixe>			148
<clause d'interdiction>	10.2.4.4	147	147
<clause préfixe>	10.2.4.4	147	146
<clause renommer préfixe>	10.2.4.3	145	146,148
<clause sinon>	6.3	79	79
<commande>	6.5.1	80	80
<commande avec>	6.5.4	84	84
<commande pour>	6.5.2	81	80
<commande tandis>	6.5.3	84	80
<commentaire>	2.4	9	
<compteur de boucle>	6.5.2	81	81,82
<contenu de locus>	5.2.2	52	51
<contexte>	8.10.2	117	114,117
<contexte distant>	8.10.1	116	117
<contextes>	8.10.2	117	114,115,117
<conversion de locus>	4.2.13	50	43
<conversion d'expression>	5.2.11	64	51
<corps de bloc>	8.2	108	110

non terminal	défini dans la section	page	employé page(s)
<corps de contexte>	8.2	108	117
<corps de module>	8.2	108	114
<corps de procédure>	8.2	108	110
<corps de processus>	8.2	108	113
<corps de quasi module>	8.2	108	
<corps de région>	8.2	108	115
<corps de spec de module>	8.2	108	117
<corps de spec de région>	8.2	108	117
<déclaration>	4.1.1	41	41
<déclaration de loc-identité>	4.1.3	42	41
<déclaration de locus>	4.1.2	41	41
<déclaration de locus avec base>	4.1.4	43	41
<définition>	2.7	10	77,81,94,110,111,114,115,118,119,124
<définition de mode>	3.2.1	14	16
<définition de nom de champ>			11
<définition d'entrée>	8.4	111	111
<définition de procédure>	8.4	110	110
<définition de processus>	8.5	113	113
<définition de signal>	9.5	124	124
<définition de synonyme>	5.1	51	51
<définition du nom de champ>	2.7	10	
<définitions>			10
<directive>	2.6	9	9
<directive CHILL>	2.6	9	9
<directive de libération>	2.6	9	9
<directive d'implémentation>			9
<élément de chaine>	4.2.6	46	43
<élément de début>	4.2.7	46	46,61
<élément de droite>	4.2.7	46	46,61
<élément de gauche>	4.2.7	46	46,61
<élément d'ensemble>	3.4.5	19	19
<élément d'ensemble avec numéro>	3.4.5	19	
<élément d'ensemble avec numéros>			19
<élément de rangée>	4.2.8	47	43
<élément inférieur>	4.2.9	48	48,62
<élément supérieur>	4.2.9	48	48,62
<énoncé d'action>	6.1	77	108
<énoncé déclaratif>	4.1.1	41	109
<énoncé de définition de neumodes>	3.2.3	16	109
<énoncé de définition de procédure>	8.4	110	109
<énoncé de définition de processus>	8.5	113	109
<énoncé de définition de signal>	9.5	124	109,118
<énoncé de définition de synmode>			118
<énoncé de définition de synmodes>	3.2.2	16	109
<énoncé de définition de synonyme>			118
<énoncé de définition de synonymes>	5.1	51	109
<énoncé de définition de neumode>			118
<énoncé définissant>	8.2	109	109
<énoncé d'entrée>	8.4	111	108
<énoncé de saisie>	10.2.4.5	148	144
<énoncé de visibilité>	10.2.4.2	144	108

non terminal	défini dans la section	page	employé page(s)
<énoncé d'octroi>	10.2.4.4	146	144
<énoncé informatif>	8.2	109	108,109
<énumération de locus>	6.5.2	81	81
<énumération de valeur>	6.5.2	81	81
<énumération ensembliste>	6.5.2	81	81
<énumération par intervalle>	6.5.2	81	81
<énumération par pas>	6.5.2	81	81
<espace>	5.2.4.6	55	55
<étiquette de cas>	10.1.3	136	136
<événement à choisir>	6.17	91	91
<expression>	5.3.2	70	20,24,25,27,28,30,36,46,47,48 56,64,65,69,70,75,79,81,84,86 89,90,101,104,137
<expression démarrer>	5.2.14	68	52,90
<expression d'indice>			104
<expression écrire>	7.4.9	104	104
<expression indice>	7.4.6	101	101
<expression parenthèsée>	5.2.16	69	52
<expression positionnement>	7.4.6	101	101
<expression recevoir>	5.3.8	76	75
<expression usage>	7.4.6	101	101
<extension d'ensemble>	3.4.5	19	19
<extension d'ensemble avec numéros>	3.4.5	19	19
<extension d'ensemble sans numéros>	3.4.5	19	19
<fenêtre de saisie>	10.2.4.5	148	148
<fenêtre d'octroi>	10.2.4.4	146	146
<filet>	11.2	152	41,42,77,110,113,114,115
<généralité>	8.4	111	111
<genre de chaînè>	3.11.2	27	27
<implantation de champ>	3.11.6	36	30,34,35
<implantation d'élément>	3.11.6	36	28,35
<indicateur de texte d'origine>	8.10.1	116	116
<indice supérieur>	3.11.3	28	28
<indifférent>	10.1.3	137	136
<initialisation>	4.1.2	41	41
<initialisation domaniale>	4.1.2	41	41
<initialisation viagère>	4.1.2	41	41
<intervalle>	5.2.5	56	56
<intervalle de littéral>			26
<intervalle littéral>	3.4.6	20	20,137
<itération>	6.5.2	81	81
<lettre>	5.2.4.6	55	8,55
<liste de définitions>	2.7	10	41,42,43,51,93,111
<liste de définitions de nom de champ>	2.7	11	
<liste de définitions de noms de champ>			30,34
<liste de noms>			14,35
<liste de noms de champ>	5.2.5	57	57
<liste de noms d'interdiction>	10.2.4.4	147	147

non terminal	défini dans la section	page	employé page(s)
<mode descripteur>	3.6.4	22	21
<mode des éléments>	3.11.3	28	28
<mode des éléments de tampon>	3.9.3	25	25
<mode de synchronisation>	3.9.1	24	16
<mode d'indice>	3.10.3	26	26,28,35
<mode discret>	3.4.1	17	16,79
<mode enregistrement>	3.10.3	26	26
<mode ensemble>	3.4.5	19	17
<mode ensembliste>	3.5	21	16
<mode entier>	3.4.2	18	17
<mode événement>	3.9.2	24	24
<mode exemplaire>	3.8	24	16
<mode intervalle>	3.4.6	20	17
<mode primitif>	3.5	21	21
<mode procédure>	3.7	23	16
<mode rangée>	3.11.3	28	27
<mode rangée paramètre>	3.11.3	28	28
<mode repère>	3.6.2	22	16,22
<mode repère libre>	3.6.3	22	21
<mode repère lié>	3.6.2	22	21
<mode simple >	3.3	16	16
<mode structure>	3.11.4	29	27
<mode structure emboîtée>	3.11.4	29	29
<mode structure étagée>	3.11.5	34	29
<mode structure paramètre>	3.11.4	30	29
<mode tampon>	3.9.3	25	24
<module>	8.6	114	77,115
<module de spec>			77
<module distant>	8.10.1	116	114
<mot>	3.11.6	36	36
<multiplet>	5.2.5	56	51
<multiplet de rangée>	5.2.5	56	56
<multiplet de rangée avec indices>	5.2.5	57	57
<multiplet de rangée sans indices>	5.2.5	57	57
<multiplet de structure>	5.2.5	57	56
<multiplet de structure avec noms de champ>	5.2.5	57	57
<multiplet de structure sans noms de champ>	5.2.5	57	57
<multiplet ensembliste>	5.2.5	56	56
<nom>	2.7	10	18,19,20,21,22,23,24,25,26,29 30,43,44,45,49,50,52,54,56,64 65,68,69,86,92,93,113,124,137,154
<nom d'accès>	4.2.2	44	43
<nom de>			26,27,28
<nom de champ>	2.7	10	57,63,147
<nom de mode chaîne originel>	3.11.2	27	27
<nom de mode rangée originel>	3.11.3	28	28
<nom de mode structure variable originel>	3.11.4	30	30
<nom de registre>	2.7	11	23
<nom de repère de texte>	2.7	11	
<nom de repère d'implantation>	2.7	11	
<nom de valeur>	5.2.3	52	51
<nom d'exception>	2.7	11	23,89

non terminal	défini dans la section	page	employé page(s)
<quasi liste de paramètre formel>			118
<quasi liste de paramètres formels>	8.10.3	118	118
<quasi module>	8.10.3	118	108
<quasi paramètre formel>	8.10.3	118	118
<quasi région>	8.10.3	118	108
<rangée derepère>			43
<rangée derepèrée>	4.2.5	45	
<région>	8.7	115	108,115
<région distante>	8.10.1	116	115
<repère libre derepère>	4.2.4	45	43
<repère lié derepère>	4.2.3	45	43
<représentation textuelle de nom>	2.7	10	10,147,149
<représentation textuelle de nom de modulion>	10.2.4.5	149	149
<représentation textuelle de nom préfixe>	2.7	10	10,11
<représentation textuelle de nom simple>	2.2	8	10,11,77,85,89,110,113,114,115,116 117,118,119
<résultat>	6.8	88	88
<signal à choisir>	6.19.2	93	93
<sous-expression>	5.3.2	70	70
<sous-opérande-1>	5.3.3	71	71
<sous-opérande-2>	5.3.4	71	71
<sous-opérande-3>	5.3.5	72	72
<sous-opérande-4>	5.3.6	74	74
<spec de module>	8.10.2	117	108,115
<spec de module distante>	8.10.1	116	117
<spec de paramètre>	3.7	23	23,111,118
<spec de région>	8.10.2	117	108,115
<spec de région distante>	8.10.1	116	
<spec de résultat>	3.7	23	23,110,118
<spécification de rangée>	3.11.5	35	34,35
<spécification d'étiquettes de cas>	10.1.3	136	30,34,79
<spec région distante>			117
<symbole>	5.2.4.6	55	55
<symbole d'affectation>	6.2	78	41,42,77,78,81
<taille de chaîne>			46,61
<taille de pas>	3.11.6	36	36
<taille de patron>			36
<taille de rangée>			48
<taille de tranche>	4.2.7	46	62
<tampon à choisir>	6.19.3	94	94
<tranche de chaîne>	4.2.7	46	43
<tranche de rangée>	4.2.9	48	43
<valeur>	5.3.1	69	41,51,57,65,77,86,88,92,99,100 154
<valeur anonyme>	3.4.5	19	19
<valeur champ de structure>	5.2.10	63	51
<valeur de pas>	6.5.2	81	81
<valeur élément de chaîne>	5.2.6	61	51
<valeur élément de rangée>	5.2.8	62	51

non terminal	defini dans la section	page	employe page(s)
<valeur finale>	6.5.2	81	81
<valeur indéfinie>	5.3.1	69	69
<valeur initiale>	6.5.2	81	81
<valeur primitive>	5.2.1	51	45,61,62,63,65,75,85,86,92
<valeur tranche de chaîne>	5.2.7	61	51
<valeur tranche de rangée>	5.2.9	62	51
<vide>	6.11	89	89,109,116,118,145

APPENDICE G: INDEX

PARTIE II

Suppléments à la Recommandation Z.200

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

LISTE DES DOCUMENTS CONCERNANT LA FORMATION DU PERSONNEL AU LANGAGE CHILL

CCITT, Commission d'études XI, période 1977-1980: Introduction to CHILL (en anglais). Manuel du CCITT établi par la Commission d'études XI.

Disponible auprès du service des ventes de l'UIT, Place des Nations, CH-1211 Genève 20, Suisse.

Introduction au CHILL (en japonais). K. Maruyama, NTT, Tokyo, mai 1981.

Disponible auprès de M. Norio Sato, Musashino-ECL, Nippon Telegraph and Telephone, Musashino-shi, Tokyo 180, Japon.

Introduction to the CCITT high level language for SPC systems: CHILL (en anglais). W. Buerger, H. Sorgenfrei, Siemens Telefone System Division, 1978. Recueil de copies faites à partir de transparents.

N'est pas disponible auprès du service des publications.

Management information on CHILL. H. Sorgenfrei, Siemens Telefone Systems Division, 1980. Recueil de copies faites à partir de transparents.

N'est pas disponible auprès du service des publications.

A software development system based on CHILL. H. Krafka, Siemens OeV ET S3. Rapport de 3 pages plus copies faites à partir de transparents.

N'est pas disponible auprès du service des publications.

Introduzione all linguaggio CHILL (en italien). R. Martucci, ITALTEL. Progetto Finalizzato Informatica C.N.R., ETS/PISA 1982.

Disponible auprès de CSELT. Via Reiss Romoli 274, I-10148 Turin, Italie.

CHILL: Die Neue CCITT-sprache (en allemand). H. Zwittlinger, Abend-Technikum und Software-Schule, Berne, Suisse.

H. Lang & Cie, Berne 1981 (vol. 4, Beiträge Mathematik, Informatik, Nachrichtentechnik). Cours, composé de 16 tableaux, donné à l'Université de Berne en 1980/81.

Disponible en librairie.

CHILL: the standard language for SPC systems. K. Rekdal, RUNIT, Trondheim, Norvège. Recueil de copies faites à partir de transparents imprimés par ordinateur.

Disponible auprès de M. K. Rekdal, RUNIT, Strindvegen 2, N-7034 Trondheim, Norvège.

CHILL/D: A self-instructional manual (volumes I, II, III). T. Valk-Fai (Philips PITTC), Philips PTI, Hilversum, 1982. Manuel de cours portant sur des sujets plus vastes que le sujet indiqué.

A commander à: M. J. van Doggenaar. PITTC, PO Box 32, 1200 JD Hilversum, Pays-Bas.

CHILL: Eine moderne Programmiersprache für die Systemtechnik. W. Sammer, H. Schwaertzel (Siemens AG), Springer Verlag, 1982. Manuel sur le CHILL et la description de l'implémentation Siemens (en allemand).

Disponible en librairie.

Elementos del Lenguaje CHILL. CTNE (J. Munera, editeur), Madrid, mai 1982. Manuel didactique à développer, décrivant un sous-ensemble de base du CHILL (en espagnol).

Disponible auprès de M. J. Munera, CTNE, Dept. de Normativa Técnica, Apartado 753, Madrid 13, Espagne.

Einführung in die CCITT High Level Programming Language CHILL. W. Buerger, H. Sorgenfrei, W. Eldon, Siemens AG, Bereich Fernsprechsysteme, Munich, mai 1980. Cours avec exercices et exemples, en allemand, 3 volumes.

Disponible auprès de Siemens Lehr- und Lernmittel, Postfach 830951, D-8000 Munich 83, République fédérale d'Allemagne.

CCITT, Commission d'études XI, période 1980-1981. Rapporteur for CHILL Maintenance. CHILL User's Manual (5e version). Manuel sur le CHILL pour enseignants et étudiants.

Non encore disponible au service des ventes de l'UIT mais publié dans le CHILL Bulletin, volume 4, n° 1, mars 1984.

Disponible auprès de M. Kees Smedema, AT&T and Philips Telecommunications, rue des Deux-Gares 80, B-1070, Bruxelles, Belgique.

Supplément n° 2

LISTE D'ACCÈS AUX SYSTÈMES DE PROGRAMMATION CHILL POUR UTILISATION À BUT NON LUCRATIF PAR DES ORGANISMES SCIENTIFIQUES ET ÉDUCATIFS

(Jusqu'au mois d'octobre 1984)

Siemens: Le compilateur et le programme de mise au point peuvent être mis à la disposition d'organismes scientifiques et éducatifs, moyennant une redevance pour couvrir les frais de port. Une implémentation à l'Université technique de Berlin a été effectuée.

S'adresser à M. Reithmaier, Siemens AG, K OeV EP D13, POB 700073, D-8000 Munich 70, République fédérale d'Allemagne.

STERIA: Le prix d'une licence de base pour utilisation à des fins éducatives ou scientifiques du front avant du CHILL s'élève à 300 000 FF.

S'adresser à STERIA, avenue de l'Europe, F-78140 Vélizy-Villacoublay, France.

PTT Dr. Neher Laboratory: Le front avant du CHILL avec une machine virtuelle d'exécution et un programme de mise au point peuvent être utilisés par des organismes scientifiques ou éducatifs aux Pays-Bas. Le temps machine sera facturé.

S'adresser à M. G. H. te Sligte, Dr. Neher Laboratory, POB 421, NL-2260 Leidschendam, Pays-Bas.

Nordic CHILL compiler: Des organismes scientifiques et éducatifs peuvent accéder au système de programmation intégré du CHILL (CHIPSY) moyennant une redevance nominale. Le système de programmation, sur l'installation Hasler de Berne, est utilisé par les étudiants de l'Ecole suisse de logiciel et le Département d'informatique de l'Université de Berne.

S'adresser à RUNIT, Strindvegen 2, N-7034 Trondheim, Norvège.

Danish Telecom Research Laboratory: Le compilateur et l'interpréteur CHILL peuvent être obtenus par des organismes scientifiques ou éducatifs moyennant une redevance nominale.

S'adresser à Peter Haff, Telecom Research Laboratory, Borups Alle 43, DK-2200 Copenhague N, Danemark.

NTT, Japon: La méthode et les conditions d'accès au système de programmation CHILL sont à l'étude.

Supplément n° 3

LISTE DE RÉFÉRENCES AUX PUBLICATIONS CONCERNANT LE LANGAGE CHILL

ANDERSEN (T.): A Portable CHILL Runtime System. Scandpower. *2e Conférence CHILL*, Lisle, Illinois, USA. 10 pages, 1983.

BORDELON (E. P.): Name Binding in CHILL. Bell Laboratories. *2e Conférence CHILL*, Lisle, Illinois, USA. 10 pages, 1983.

BOTSCH (D.): The use of high level language programming and its impact on the software of digital switching systems. *ISS'79*, Paris, 1979.

BOURGONJON (R. H.): CHILL... The standard high level language for programming SPC telephone exchanges. Philips' Telecommunicatie Industrie B.v., Hilversum, Pays-Bas. *Journal A*, volume 22, n° 4, 4 pages en anglais, 1981.

BOURGONJON (R. H.), BREEUS (C.): Implementation experience with the CCITT high level language CHILL. *ISS'79*, Paris, 1979.

BOURGONJON (R. H.): The CCITT high level programming language. *3e Conférence SETSS*, Helsinki, Finlande, 1978.

BOURGONJON (R. H.), REKDAL (K.): CHILL user's manual, 1981.

BOURGONJON (R. H.): Programming Languages, Environments and CHILL. Philips' Telecommunicatie Industrie B.v., Pays-Bas. *2e Conférence CHILL*, Lisle, Illinois, USA. 6 pages, 1983.

BOUTE (R. T.), JACKSON (M. I.): A joint evaluation of the programming languages ADA and CHILL. BTM,

Anvers, Belgique. Standard Telecom. Laboratory, Harlow, Royaume-Uni. *IEE Conference*, publication n° 198, 6 pages en anglais, 1981.

BRANQUART (P.), LOUIS (G.), WODON (P.): Aspects de CHILL, le langage du CCITT. MBLE, Bruxelles, Belgique. *Revue TSI (Technique et Science Informatique)*, 9 pages en français, 1982.

BRANQUART (P.), LOUIS (G.), WODON (P.): On the Analytical Description of CHILL. Philips Research Laboratory, Bruxelles. *2e Conférence CHILL*, Lisle, Illinois, USA. 6 pages, 1983.

BUTCHER (B. A.): Selecting an Appropriate CHILL Subset. ITT-ATC. *2e Conférence CHILL*, Lisle, Illinois, USA. 6 pages en anglais, 1983.

CAIN (G. J.), JACKSON (L. N.), VESETAS (R.), WALTER (A.), YONG (W. B.): Computer aided software generation (the MELBA system for generating CHILL code). *4e Conférence SETSS*, Université de Warwick, Coventry, Royaume-Uni, 1981.

CAMICI (A.), GIARRATANA (V.), NIRO (F.), MODESTI (M.): CHILL for supporting software engineering environments. CSELT & SIP, Italie. *5e Conférence SETSS*, Lund, Suède. 4 pages en anglais, 1983.

CAMICI (A.), GIARRATANA (V.), NIRO (F.), PANARONI (P.): Criteri di progetto nell'implementazione del linguaggio di programmazione CHILL. *Congrès AICA 1980*, Bologne, 1980.

CAMICI (A.), GIARRATANA (V.), MANUCCI (F.): A CHILL software development system for distributed architecture. *ICC'81*, Denver, USA, 1981.

CARRELLI (C.), MANUCCI (F.), ROSCI (G.): CHILL Programming System: implementation and operational aspects. *Congress ISS'81'CIC Symposium*, Montréal, Canada (en anglais), 1981.

CARRELLI (C.), MANUCCI (F.), MARTUCCI (R.): The CCITT high level language: an approach in Italy. *ISS'79*, Paris, 1979.

CCITT: CHILL language definition. *Recommandation Z.200*, 1984.

CONROY (R. A.): Impacts of CHILL on System Design. ITT-ATC. *2e Conférence CHILL*, Lisle, Illinois, USA. 3 pages, 1983.

DACKER (B.), JACOBSON (I.): Real time system design using CHILL. *3e Conférence SETSS*, Helsinki, Finlande, 1978.

DE BACHTIN (O.), LINDROOS (L.), TONNBY (I.): Programmed testing of AXE systems using a CHILL based language PILOT. *4e Conférence SETSS*, Université de Warwick, Coventry, Royaume-Uni, 1981.

DENENBERG (C. G.): CHILL implementation techniques. *3e Conférence SETSS*, Helsinki, Finlande, 1978.

DE NICOLA (R.), MARTUCCI (R.), ROBERTI (P.): A CHILL based distributed architecture. *International Computing Symposium*, Londres, 1981.

DENIS (G.), LANGLOIS (C.): Présentation d'une machine langage orientée vers le langage CHILL. CNET, Paris, France. *Revue L'écho des Recherches*, 7 pages en français, 1981.

DENIS (G.), LANGLOIS (C.), D'ISSERNIO (J. P.): Machine langage adaptée à CHILL: premiers résultats d'évaluation. CNET, Paris, France. *Congrès ISS'81'CIC Symposium*, Montréal, Canada. 6 pages en anglais et en français, 1981.

FEICHT (E. J.): «CHILL Factory»: production and maintenance of a large CHILL software system. Siemens, République fédérale d'Allemagne. *5e Conférence SETSS*, Lund, Suède. 6 pages en anglais, 1983.

GIARRATANA (V.), MODESTI (M.): An SDL into CHILL skeleton translation system. CSELT & SIP, Italie. *5e Conférence SETSS*, Lund, Suède. 6 pages en anglais, 1983.

GIARRATANA (V.), GIANNETI (B.), MUSSA (P. L.): Verso la formalizzazione della generazione di codice nei compilatori: un generatore di codice indipendente della macchina per il CHILL. *Congrès AICA 1980*, Bologne, 1980.

GREEN (G. A.), HALLSTEINSEIN (S. O.), WANVIK (D. H.), NOKKEN (L.): Separate Compilation in CHIPSY. RUNIT. STK. *2e Conférence CHILL*, Lisle, Illinois, USA, 1983.

GUTFELDT (H.): Modelling telecom processes with CHILL process. Hassler AG, Berne, Suisse. *5e Conférence SETSS*, Lund, Suède. 4 pages en anglais, 1983.

GUTFELDT (H.): CHILL. Hassler AG, Berne, Suisse. *Hassler Werk Zeitung*, 1 page en allemand, 1981.

GUTFELDT (H.): SDL and CHILL Structured Programming. Hassler AG, Berne, Suisse. *2e Conférence CHILL*, Lisle, Illinois, USA. 13 pages, 1983.

GUTTMAN (N.): Efficient Implementation of Nested Non-recursive Procedures in CHILL. Bell Laboratories. *2e Conférence CHILL*, Lisle, Illinois, USA. 6 pages, 1983.

HAFF (P.), BJOERNER (D.): CHILL formal definition. Dansk Datamatik Center, Lyngby, Danemark, 1981.

HAMMER (D.), FRANKEN (G.), GREEN (P. C.): A distributed Operating System for the TCP16 System. Philips' Telecommunicatie Industrie B.v., Pays-Bas. *5e Conférence SETSS*, Lund, Suède. 6 pages en anglais, 1983.

HAQUE (T. A.), DALEY (R. W.): ITT 1240 Digital Exchange – CHILL programming environment. ITT-ESC, 1983.

HAQUE (T. A.): ITT CHILL programming environment, *Electrical Communication 1983*, volume 9. ITT-ESC, 1983.

HAQUE (T. A.): CHILL programming environment, *Proceedings of COMPSAC'83*, Chicago, USA. ITT-ESC, 1983.

HRVENSALO (J.): CHILL – unsi standardikieli. Finnish State Research Center, Finland. *Elektroniikka No. 19*, 3 pages en finlandais, 1981.

KEEDY (J. L.): A report on the concurrent processing features of the CCITT language CHILL. Telecom Australia. 45 pages en anglais, 1981.

KRAFKA (H. H.): A software development system based on CHILL. *4e Conférence SETSS*, Université de Warwick, Coventry, Royaume-Uni, 1981.

KURKI-SUONIO REINO: Mikrojen uudet. Université de Tampere, Finlande. *Elektroniikka No. 19*, 3 pages en finlandais, 1981.

LANGLOIS (C.): Evaluation of a CHILL Oriented Processor. CNET. *2e Conférence CHILL*, Lisle, Illinois, USA. 9 pages, 1983.

LO (P.), SHAW (F.): A Distributed Operating System for CHILL. ITT-ATC. *2e Conférence CHILL*, Lisle, Illinois, USA. 5 pages, 1983.

McCULLOUGH (R. H.): A CHILL Compiler Based on the Portable C Compiler. Bell Laboratories. *2e Conférence CHILL*, Lisle, Illinois, USA. 13 pages, 1983.

MARUYAMA (K.), KONISHI (K.), SATO (N.): NTT CHILL Implementation Aspects and its Application Experience. Musashino Electrical Communication Laboratory, NTT, Japon. 6 pages en anglais, 1981.

MEIJER (R. W.), te SLIGTE (G. H.): Status report of CCITT HLL implementation at the Dr. Neher Laboratory of the Netherlands PTT. *3e Conférence SETSS*, Helsinki, Finlande, 1978.

MEILING (E.), STEEN (U.) PALM: A comparative Study of CHILL and ADA on the Basis of Denotational Descriptions. Dansk Datamatik Center, Lyngby. *2e Conférence CHILL*, Lisle, Illinois, USA. 14 pages, 1983.

MODESTI (M.), GIARRATANA (V.): An SDL to CHILL Skeleton Transformer. SIP DG Rome, CSELT Turin. *2e Conférence CHILL*, Lisle, Illinois, USA. 13 pages, 1983.

MOORE (B. G.), CHANDRASEKHARAN (M.): Tools for maintaining consistency in large programs compiled in parts. GTE Laboratories, USA. *5e Conférence SETSS*, Lund, Suède. 5 pages en anglais, 1983.

OLSEN (N. C.): Alternatives for handling I/O in CHILL. ITT-ATC. *2e Conférence CHILL*, Lisle, Illinois, USA. 5 pages, 1983.

PANARONI (P.), RUGANI (U.): Il parallelismo nel linguaggio CHILL: descrizione, analisi comparata e sua implementazione. *Congrès AICA 1980*, Bologne, 1980.

REITHMAIER (E.): Compilation Control in a large CHILL Application. Siemens. *2e Conférence CHILL*, Lisle, Illinois, USA. 10 pages, 1983.

RIETSCHOTE van (H. F.): Debugging in a CHILL Oriented Program Development System. Philips' Telecommunicatie Industrie B.v., Pays-Bas. *2e Conférence CHILL*, Lisle, Illinois, USA. 4 pages, 1983.

REKDAL (K.): CHILL, the standard language for programming SPC systems. *IEEE Transactions on Telecommunications*, juin 1981. Proceedings of NTC, New Orleans, USA, novembre 1981. 20 pages en anglais, 1981.

REKDAL (K.): CHILL, the standard language for programming SPC systems. Mini-micro Systems, Shenyang, Chine. 7 pages en chinois, 1982.

REKDAL (K.): Introduction to CHILL, 1980.

REKDAL (K.), BOTNEVIK (H.), HALLSTEINSEIN (S. O.), VENSTAD (A.): Some implementation aspects of CHILL. *3e Conférence SETSS*, Helsinki, Finlande, 1978.

REKDAL (K.): CHILL in the Software Engineering Context. *5e Conférence SETSS*, Lund, Suède, 1983.

REKDAL (K.): CHILL, The Standard Language. *Proceedings of COMPSAC'83*, Chicago, USA, 1983.

RUDMIK (A.), MOORE (B. G.): The separate Compilation of Very Large CHILL Programs. GTE. *2e Conférence CHILL*, Lisle, Illinois, USA. 11 pages, 1983.

SMEDEMA (C. H.), BISHOP (R.), CHEUNG (R. C. H.), BORDELON (E. P.), FEAY (M. R.), LOUIS (G.): Separate Compilation and the development of large programs in CHILL. PTI Pays-Bas, British Telecom Royaume-Uni, Bell Laboratories Etats-Unis et PRL Belgique. *5e Conférence SETSS*, Lund, Suède. 7 pages en anglais, 1983.

SMEDEMA (C. H.), MEDEMA (P.), BOASSON (M.): The programming languages: PASCAL, MODULA, CHILL, ADA. Prentice-Hall International. 160 pages en anglais, 1983.

SMEDEMA (C. H.): A test approach for CHILL based systems. Philips' Telecommunicatie Industrie B.v., Hilversum, Pays-Bas. *2e Conférence CHILL*, Lisle, Illinois, USA. 4 pages, 1983.

TEICHROEW (D.), BLOCK (C.), KYO CHUL KANG, CHIKOFSKY (E.): Usage of the System Encyclopedia Manager (SEM) System with the CCITT Functional Specification and Description Language (CCITT/SDL). *2e Conférence CHILL*, Lisle, Illinois, USA. 15 pages, 1983.

THALHAMER (J. A.): Design Issues of a High Level Symbolic Debugger for CHILL. ITTE-PSC, 1983.

THEURETZBACHER (N.): Implementation of the CHILL Tasking Concept in a Compiler and a Real Time Operating System. ITT-Austria. *2e Conférence CHILL*, Lisle, Illinois, USA. 6 pages, 1983.

VALK-FAI (T.): A training course in the use of CHILL. PTI. *2e Conférence CHILL*, Lisle, Illinois, USA. 11 pages, 1983.

VENSTAD (A.): On rehosting CHIPSY. RUNIT. *2e Conférence CHILL*, Lisle, Illinois, USA. 10 pages, 1983.

WEN (W.): Problem Oriented Languages. ITT-ATC. *2e Conférence CHILL*, Lisle, Illinois, USA. 4 pages, 1983.

WINKLER (J. F. H.): A new Methodology for I/O and its application to CHILL. Siemens AG, Munich. *2e Conférence CHILL*, Lisle, Illinois, USA. 16 pages, 1983.

ZWITTLINGER (H.) Dr: CHILL, die neue CCITT Sprache. Ingenieurschule, Berne, Suisse. Programmiersprachen für die Nachrichtentechnik, Université de Berne. 76 pages en allemand, 1980/81.

Supplément n° 4

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
République fédérale d'Allemagne	Siemens AG	7700	BS 1000/ BS 2000	7700	I	EWSD,			I
				7800	I	ETS,	Système de développement et de production complet		I
		7800	BS3000	IBM 370	I				
				SSP 103	I	BIGFON,			I
		IBM 370	OS/MVS	SSP 112D	I	EMS			U
				Famille 8086	I				
						ISDN			U
	Tekade	VAX II	VMS (Front avant)	6800 (Front avant)	U U				
	Standard Electrik Lorenz (ITT/SEL)	IBM 370	MVS	iAPX 86	I	Système 12			I
		IBM 370	MVS	Famille 8086	I	5600 BCS			U
							Système de développement et de production		I
						ISDN			U
						DFS			U

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(suite)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Autriche	ITT Autriche	HP VAX	 VMS	8085 8086	I I	5200 BCS (AMANDA)			I
Belgique	BTM	IBM 370	MVS	8086	I	Système 12 ISDN	 Système de développement et de production		I I U
Brésil	CPqD TELEBRAS	VAX 11	VMS	iAPX 286	P	TROPICO			P
Danemark	Laboratoire de recherche en télécommunication	VAX (portatif)	VMS	Machine virtuelle centrale en VAX	I (Disponible dans le commerce)		Eliminateur de fautes CHILL symbolique		U

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(suite)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Espagne	CTNE	Aucun compilateur CHILL n'est prévu pour l'instant *Remarque* – Nous utilisons actuellement des instruments ITT en SESA (SESARC); aucune décision n'a été prise en ce qui concerne les instruments qui devront être utilisés dans l'avenir.				Système 12			U
Etats-Unis d'Amérique	ATT-Bell Labs.				P				
	ITT	IBM 370	MVS	8086	I	Système 12	Système de développement et de production		I I
	CTE Laboratories	IBM 370 VAX 11/780 TANDEM	OS basé sur IBM 370	IBM 370 TANDEM VAX 11/780 I 8086 I 432	I I I I U				

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(suite)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Finlande	Technical Research Centre, Telecommunications Laboratories						Logiciel d'appui de temps de traitement CHILL pour programmes compilés	Micro-ordinateur UMC	U
France	PTT et STERIA	DPS 8	MULTICS	CHILL Processeur frontal	I I	Expérimentation du système SPC			I
Italie	TELETTRA	UNIVAC 1100/60	EXEC 8	MIC 30 MIC 10	U				
	SIP-CSELT	VAX 780 VAX 780 VAX 780	VMS UNIX VMS	MIC 20 VAX 780 VAX 780	I U I				
	ITALTEL-SIT SPA	VAX 11/750	MVS	MIC 20	U	UT-100 Nouveau système de commutation	Processeur d'optimisation fonctionnant à la sortie du compilateur Du langage de type LDS au traducteur CHILL		I I
	FACE (ITT)	IBM 370	MVS	Famille 8086	I	Système 12	Système de développement et de production		I I

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(suite)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Japon	KDD	D10 (Compilateur NTT)	D180	D10	I	XE-10 Digital INTS			U
						XE-20 Digital INTS			U
	NTT	D10	D180	D10	I	– D10ESS (D100B, téléphone mobile, téléphone de données)	Générateur de données de bureau (voir le Document SETSS'75)		I
						– D60 (transit téléphonique numérique)			I
		DIPS	DIPS (TSS de NTT et OS en temps réel)	D10	I	– Système d'enregistrement et de conversion de télécopie			I
				D10-VLSI		– D70 (transit téléphonique mixte numérique et local)			I
						– D50 (commutation de circuits de données)			U
						– Système de central à large bande			U
		DIPS	DISP	D10 et D10-VLSI	U [2]	– D70 version INS			U
						INS			U

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

[2] Amélioré avec génération de code partiel et décomposition de modules (voir le Document de la 2e Conférence CHILL).

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(suite)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Norvège	STK (ITT)	ND100	SINTRAN III	iAPX 86	I	5500 BCS Digimat 2000			I U
	Administration norvégienne des télécommunications	ND 100	SINTRAN III	I 8086	I	X25 Télécommande des stations côtières Réseau national expérimental (maintenance et exploitation) Service expérimental, commutateur vidéo intégré			U U P P
Pays-Bas	Administration	PDP 10	(TOPS-10 (TOPS-20	PDP-11 MC 68000 PDP-10	I I U		Système d'appui PRX/A OMC	OS Signalisation C7 Protocole de voie D RNIS Application de base de données Multiplex terminaux	I I U U U P I I
	AT&T et Philips Telecom	IBM 370	UNIX	IBM 370	U				

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

LISTE D'IMPLÉMENTATIONS ET D'APPLICATIONS DU CHILL *(fin)*

Origine		Compilateur CHILL				Application du CHILL			
Pays	Organisation	Ordinateur principal	OS	Ordinateur cible	Etat [1]	Système SPC	Logiciel d'appui	Autres	Etat [1]
Royaume-Uni	BT	ND 100	SINTRAN	8086	I			Protocole télétexte	I P
	ITT/ESC	IBM 370	MVS	8086	I	Système 12 5600 BCS	Logiciel d'appui	—	I U
Suisse	Hasler	ND-100 (compilateur fourni par RUNIT)	SINTRAN III	I 8086	I		OS de destination prévue	Convertisseur télex télétex MARK II	U U
URSS	Institut des télécommunications de Moscou	ES-1033 CC-NEVA-1M	OS ES NEVA-1M	CC NEVA-1M	I				
	Université d'Etat de Moscou	BESM-6		CC NEVA-1M	I				
Danemark Finlande Norvège Suède Royaume-Uni	RUNIT Administrations des télécommunications des pays nordiques et British Telecom	ND 100 Intellec MDS VAX 11 IBM PC	SINTRAN III [ISIS II CP/M-86 [VMS UNIX CP/M-86	iAPX 86 iAPX 286 ND 100 iAPX 86 iAPX 86 iAPX 86 iAPX 86 iAPX 86	I U U I I P U I		Système de temps d'exécution CHILL		I

[1] P: Prévu.
U: A l'étude.
I: Mise en service.

Imprimé en Suisse — ISBN 92-61-02252-9