BULL-IMAG
Rapport Bull - IMAG 3-90
Décembre 90

Manuel du langage Guide (V1.5)

Hiep NGUYEN VAN, Michel RIVEILL et Cécile ROISIN

Unité Mixte Bull - IMAG
Z.I de Mayencin - 2, avenue de Vignate
F-38610 Gières - FRANCE
e-mail : (riveill, roisin)@imag.imag.fr

Résumé :
Ce rapport constitue le manuel de Guide, un langage pour la programmation d'applications réparties (version 1.5, décembre 1990). Ce langage fait partie intégrante du système réparti Guide, qui fournit son environnement d'exécution. Pour chaque trait du langage, le rapport présente une description syntaxique et des exemples d'illustration. Des annexes présentent la carte syntaxique, les types de base, la bibliothèque des types et des classes et le manuel d'utilisation du compilateur. Ce document est complété par un rapport présentant des exemples de programmes écrits en langage Guide.
NB : Les commentaires sont à transmettre à Michel Riveill ou à Cécile Roisin. Signaler en particulier toute incohérence entre ce manuel et le compilateur.

1 Introduction

Le programmeur trouvera dans ce manuel une description précise du langage Guide dans la version 1.5 disponible au 15 décembre 1990 ; ses différents éléments sont présentés à l'aide d'exemples (tous compilés et exécutés sur le système Guide) et leur définition est accompagnée des règles de syntaxe nécessaires à leur utilisation. La syntaxe du langage est récapitulée dans l'annexe A. L'annexe B contient la définition et les règles d'utilisation des types de base du langage et des types et constructeurs prédéfinis ; le code source des types de la bibliothèque Guide est donné dans l'annexe C et l'annexe D décrit le principe de fonctionnement du compilateur ainsi que son utilisation. Pour une description du modèle d'objets de Guide et des motivations de ce langage, le lecteur peut consulter [1].

Conventions d'écriture des éléments de syntaxe :

- les terminaux sont notés en majuscules ou entre guillemets,
- les non-terminaux sont notés entre < et >,
- un nombre quelconque de x s'écrit { x },
- au moins un x s'écrit { x }*,
- au plus un x s'écrit [x],
- exactement un x s'écrit (x),
- l'alternative s'écrit |.

Conventions d'écriture des exemples de ce manuel :

- les mots-clés du langage sont en majuscules,
- les identificateurs de type, de classe et de méthode ont leur première lettre en majuscules,
- les autres identificateurs sont entièrement en minuscules.

2 Règles de programmation

2.1 Jeu de caractères

Tous les caractères de l'alphabet IA5 sont autorisés. Le code '^L' (touche de contrôle + L) peut être utilisé pour provoquer le saut de page explicite à l'impression et il est ignoré par le compilateur.

2.2 Identificateurs

Les identificateurs doivent être alphanumériques (le premier symbole étant une lettre) et peuvent contenir le caractère souligné '_'. Les chaînes de caractères correspondant aux mots-clé et aux mots réservés ne doivent pas être utilisées. Distinction est faite entre minuscules et majuscules ; par exemple, 'MyIdent' et 'myident' sont deux identificateurs différents.

Les identificateurs des types et des classes forment deux espaces indépendants. Il est donc possible d'utiliser un même identificateur pour désigner un type et une classe. C'est ce qui est souvent fait lorsqu'un type est implémenté par une seule classe.

Syntaxe

<ident> = <letter> { <letter> | <digit> | '_' }
<letter> = ('A'..'Z' | 'a'..'z')
<digit> = ('0'..'9');

Exemple

```
Count, PERSON_1 et my_name sont des identificateurs corrects ;
string?, 12age et _name sont des identificateurs incorrects.
```

2.3 Mots-clés

```
ALL
AND
ASSERTYPE
ATTRIBUTE
BEGIN
BY
BLOCK
CASE
CLASS
CONTROL
CO_BEGIN
CO_END
CONST
NOT
OF
OR
OTHERS
OUT
POWER
PROCEDURE
RAISE
READER
REF
REPLACE
REPEAT
RETRY
```

```
CONSTRUCTOR                    RETURN
DIV                            SELF
DO                             SIGNALS
ELSE                           SUBCLASS
END                            SUBTYPE
EXCEPT                         SUPER
EXCLUSIVE                      SYNONYM
FIRST                          SYSTEM
FOR                            THEN
FROM                           TO
IF                             TYPE
IMPLEMENTS                     TYPECASE
IN                             UNTIL
IN_OUT                         UNMOVABLE
IS                             WHILE
MOD                            WRITER
METHOD                         XOR
```

Les mots-clé peuvent être écrits indifféremment soit tout en minuscules soit tout en majuscules.

2.4 Symboles

```
=                              +
#  (différence)                -
>                              /
<                              |
<=                             *
>=
&
```

2.5 Mots réservés

Types

```
Array[n] OF ...                LongInt
Boolean                        ShortInt
Char                           Record ... END
Integer                        String[n]
```

Constantes

```
FALSE (ou false)
TRUE (ou true)
NIL (ou nil)
```

Variables d'environnement prédéclarées

```
input
output
catal
myActivity
myDomain
```

Mots interdits

Pour faciliter l'utilisation d'une version ultérieure du compilateur Guide, nous donnons ci-dessous la liste des mots qu'il est interdit d'employer comme identificateurs dans les programmes Guide.

```
ADDRESS
ATOMIC
BEGIN_TRANSACTION
END_TRANSACTION
```

```
ABORT_TRANSACTION
BEGIN_C
END_C
BEGIN_LOCK
END_LOCK
INHERIT
OPERATOR
```

2.6 Commentaires

Les commentaires sur une ligne sont introduits par la chaîne '//' et terminés par le caractère de fin de ligne. Les longs commentaires sont maintenant autorisés en les plaçant entre "/*" et "*/".

Exemple

```
toto.Print; // voici un commentaire
// et un autre
// commentaire
/* voici un commentaire long. Il peut s'étendre à plusieurs lignes.
Ces commentaires ne sont pas imbriqués */
```

3 Programme Guide

Un programme Guide est composé d'un ensemble de types et de classes. Pour plus d'informations sur le modèle d'objets de Guide, se reporter à la référence [1].

Le type d'un objet définit son comportement sous la forme des spécifications des méthodes qui lui sont applicables. La partie syntaxique de cette spécification correspond à l'ensemble des signatures des variables visibles et des méthodes (ou opérations) qui permettent d'accéder aux objets de ce type (on l'appelle aussi interface d'accès). La partie sémantique d'un type correspond à la spécification formelle de l'effet de ces méthodes sur l'objet, donnée sous forme de commentaires.

Une classe décrit une réalisation possible d'un type. Un type peut être réalisé par plusieurs classes. Une classe définit un modèle de données (c'est-à-dire l'ensemble des variables qui définissent l'état des objets de cette classe), le code des méthodes déclarées dans le type qu'elle implémente, et pour chacune de ces méthodes, une condition d'activation.

On distingue deux ensembles de types et de classes :

1. les types de base du langage décrits dans l'annexe B. On les décompose en deux catégories :
   - les types élémentaires : Boolean, Integer, LongInt, ShortInt, et Char (cf. Annexe B 1) ;
   - les types structurés : Array [n] OF, Record, String[n] (cf. Annexe B 2) ;
2. les types et les classes construits. Un ensemble de types et de classes construits est fourni en standard dans la bibliothèque Guide. Ces types sont décrits dans l'annexe C.

Syntaxe

<type> = <type_base> | <type_construit>
<type_base> = <type_élémentaire> | type_structuré>

<type_élémentaire> = 'Integer' | 'ShortInt' | 'LongInt' | 'Char' | 'Boolean'
<type_structuré> = type_record | type_array | type_string

Guide est un système à objets persistants, c'est-à-dire que tous les objets connus du Système de Gestion d'Objets (SGO) sont permanents et leur destruction doit être explicite : elle peut être provoquée par le programmeur (cf. méthode Destroy définie dans le type Top Annexe C) ou par le système de ramasse-miettes [3]. La persistance est rendue possible par un mécanisme de désignation globale (ou OID) des objets gérés par le système.

4 Objets, variables et références

4.1 Définitions

Un objet, instance d'une classe, est une entité typée composée d'un état (les données) et de méthodes (le code). L'état est constitué d'un ensemble de constantes et de variables. Chaque variable (ou constante) est typée et le type peut être soit un type élémentaire (Integer, Char, Boolean, LongInt, ShortInt), soit un type structuré (Array, String, Record), soit une référence sur un type construit (REF aType).

On distingue deux catégories de variables en fonction de la nature des objets qu'elles désignent :

1. soit elles permettent de désigner d'autres objets connus du Système de Gestion d'Objets (SGO) et on les appelle variables références (ou références) ; les références sont typées (pour permettre le contrôle statique) par un type construit,
2. soit les objets qu'elles désignent sont des instances des types de base (types élémentaires ou types structurés) et sont des valeurs accédées directement (comme dans les langages de programmation classiques). Ces objets ne sont pas connus du Système de Gestion d'Objets.

On remarquera que la valeur d'une variable référence est constituée de la valeur de l'OID de l'objet qu'elle désigne (c'est une généralisation de la notion de pointeur).

Du fait que la distinction entre les deux catégories de variables (variables références et variables de types de base) est faite par les types qui forment deux ensembles disjoints (types construits et types de base), le mot-clé REF pourrait être omis comme c'est le cas dans Eiffel. Dans Guide, nous avons choisi de le laisser explicite.

Exemple

```
//déclaration d'une variable de type construit :
élément : REF Element;
//déclaration d'une variable de type élémentaire :
i : Integer ;
// déclaration de variables de type structuré :
table : Array[12] OF Boolean;
structure : Record
       nom : String[20];
       élément : REF Element;
END;
```

En plus des variables servant à définir l'état des objets, il existe une autre catégorie de variables, appelées variables de travail, qui sont déclarées dans le corps des méthodes et des procédures. Les paramètres des méthodes et des procédures font aussi partie de cette même catégorie.

Le modèle Guide étant un modèle à objets persistants, la durée de vie des objets est indépendante de celle des programmes qui les utilisent, et par conséquent des variables qui les désignent.

La durée de vie d'une variable dépend de l'endroit où elle a été déclarée :

- Déclaration dans l'état (variable visible déclarée dans un type ou variable d'état déclarée dans une classe) : la variable a la même durée de vie que l'objet englobant.
- Déclaration comme paramètre d'une méthode, d'une procédure ou comme variable de travail : la variable est créée à chaque appel et a une durée vie égale au temps d'exécution de la méthode ou de la procédure.

4.2 Portée d'une variable

La portée d'une variable dépend de l'endroit de sa déclaration.

- Déclaration dans le type (cf. 5.2) : la variable est visible dans tous les sous-types et toutes les classes qui les réalisent.
- Déclaration dans la classe (cf. 6.3) : la variable est visible dans toutes les sous-classes.
- Déclaration comme paramètre d'une méthode ou d'une procédure (cf. 6.4) : la variable est visible dans le corps de la méthode ou de la procédure. Si c'est un paramètre IN (ou IN_OUT) d'une méthode, elle est aussi visible dans les conditions d'activation associées à la méthode.
- Déclaration comme variable de travail : la variable est visible dans le corps de la méthode ou de la procédure.

4.3 Objets et références

Un objet permanent est un objet de type construit dont la création et la destruction sont explicites et qui n'est accessible que par le biais d'une référence. La transmission d'un tel objet en paramètre d'une opération consiste en la transmission de la valeur d'une référence qui le désigne.

4.3.1 Création d'un objet

La création d'un objet permanent consiste à appeler la méthode New sur la classe de l'objet. On peut donner en paramètre à cette méthode le nom du site sur lequel on veut que l'objet soit créé (par défaut, l'objet est créé sur le site d'exécution de la méthode New). En retour de cet appel, on obtient une référence sur un nouvel exemplaire de cette classe. Les champs de l'état de cet objet ne sont pas initialisés sauf ceux pour lesquels une valeur initiale a été définie dans le type ou la classe de l'objet. Pour créer des exemplaires d'une classe avec des valeurs initiales de l'état différentes, il faut définir dans la classe une méthode d'initialisation dont la fonction sera d'initialiser l'état avec les paramètres associés. Après la

création d'un exemplaire (méthode New sur la classe, cf. Annexe C.8), il faudra alors appeler une méthode d'initialisation avec les paramètres désirés.

Exemple

```
// déclaration d'une variable référence pour désigner l'objet:
a : REF aType;

// appel de la méthode New sur la classe de l'objet:
a := aClass.New;
// création de l'objet sur le site d'exécution;
a := aClass.New("nom_de_site");
// création de l'objet sur le site spécifié;

// initialisation explicite de l'objet aux valeurs désirées:
a.Init ( ... );
```

4.3.2 Destruction explicite d'un objet

La destruction explicite d'un objet permanent se fait par appel de l'opération Destroy sur l'objet (cf. méthodes définies sur le type Top, cf. Annexe C.9). Cependant, le système Guide dispose d'un ramasse-miettes [3] de façon à décharger le programmeur de la destruction explicite des objets inutilisés.

Exemple

```
// soit x la référence qui désigne l'objet que l'on veut le détruire;
// appel de la méthode Destroy sur l'objet :
x.Destroy ;
```

4.3.3 Désignation de l'objet courant : SELF

La variable référence prédéfinie SELF (ou self) contient en permanence la valeur de l'OID de l'objet en cours d'exécution. Cette variable est utilisée lorsque l'on veut appliquer une méthode sur l'objet courant.

4.3.4 Référence constante NIL

Pour exprimer qu'une variable référence ne désigne aucun objet, il suffit d'affecter à cette variable la valeur NIL (ou nil). C'est ce que fait le système lors de l'exécution de l'opération Destroy : la valeur NIL est affectée à la variable référence qui désignait l'objet détruit.

Par contre, une variable référence non encore initialisée a une valeur indéfinie (elle n'est pas initialisée à NIL).

Pour les besoins de la généricité non contrainte (cf. section 8.1.1), l'utilisation de NIL a été étendue aux types élémentaires. La valeur correspondant à NIL pour chaque type de élémentaire est celle qui a le même codage, à savoir : 0 pour les types numériques, '\0' pour le type Char et FALSE pour le type Boolean).

La constante NIL n'est pas typée. Comme cette valeur peut être affectée à toute variable référence, quel que soit son type, le type qu'on pourrait attribuer à NIL serait un type conforme à tous les types. Il n'est pas possible de définir un tel type dans notre modèle.

4.4 Variables d'environnement

Un certain nombre de variables références, appelées variables d'environnement, sont accessibles par le programmeur pour lui permettre d'interagir avec son environnement d'exécution. Ces variables sont positionnées par le système ou par le shell. A la création d'une activité, les variables input, output et catal sont initialisées à la valeur des variables correspondantes de l'activité créatrice. Elles sont modifiables par le programmeur, mais par contre myActivity et myDomain ne le sont pas. Les opérations que l'on peut appliquer sur les objets désignés par ces références sont définies par leur type (cf Annexe C). On trouve actuellement les variables prédéfinies suivantes :

- input : REF ChanIn, pour les opérations d'entrée (cf. Annexe C 6),
- output : REF ChanOut, pour les opérations de sortie (cf. Annexe C 7),
- catal : REF Catal, pour l'accès au service de noms (cf. Annexe C 4),
- myActivity : REF Activity ; contient la référence de l'activité dans laquelle la méthode courante s'exécute,
- myDomain : REF Domain ; contient la référence du domaine dans lequel la méthode courante s'exécute (cf. Annexe C 3).

5 Types construits

La déclaration d'un type construit est constituée d'un en-tête, d'une liste de déclarations de synonymes (noms de types ou de classes), d'une liste de variables d'état visibles (cf. 5.2) et d'une liste de signatures de méthodes.

Exemple

```
TYPE Boite IS
  // synonymes
  SYNONYM Name = String [80];
  // variable d'état visible
  propriétaire : Name; // variable d'état visible
  // méthodes
  METHOD Mettre (IN REF Element); SIGNALS boite_pleine;
    // ajout d'élément dans la boîte
  METHOD Prendre (OUT REF Element); SIGNALS boite_vide;
    // retrait d'un élément de la boîte
END Boite.
```

Syntaxe

<type_construit> = <en-tête_type>
{ <synonymes> }
{ <variable d'état visible> }
{ <signature_méthode> }
<fin_type>
<en-tête_type> = TYPE <ident_type_construit>
[SUBTYPE OF <ident_type_construit>] IS
<fin_type> = END [<ident_type_construit>] '.'

5.1 Synonymes

La déclaration d'un synonyme consiste à associer une chaîne de caractères à une autre de façon à faciliter l'écriture et la lecture d'un texte Guide. L'utilisation des synonymes est restreinte au renommage des types et des classes dans une déclaration de type ou de classe.

Une déclaration de synonymes dans la déclaration d'un type a une portée restreinte à ce type, à tous ses sous-types, aux classes qui le réalisent et aux sous-classes de ces dernières.

Exemple

```
SYNONYM BitMap = Array [1000,1000] OF Boolean;
SYNONYM MailBoxList = List OF REF MailBox;
```

Syntaxe

<synonymes> = { SYNONYM <ident> '='
(REF <ident_type_construit>) | <type_élémentaire> ';' }

5.2 Variables d'état visibles

Une variable d'état visible d'un objet est une variable d'état pour laquelle le programmeur demande la déclaration implicite d'opérations de lecture ou d'écriture ayant le même nom que la variable. Comme pour toute méthode, ces opérations peuvent être synchronisées en leur associant une condition d'activation (cf. 6.7).

Exemple

```
TYPE Circle IS
  rayon : Integer;
  METHOD Surface : Integer;
END Circle.
```

Utilisation d'un objet de type cercle :

```
// déclarations
i : Integer;
cer : Circle;
...
// instructions
cer.rayon := 3;
i := cer.rayon;
```

5.2.1 Déclaration et initialisation

La déclaration d'une variable d'état visible a la même forme que la déclaration d'une variable d'état dans le corps de la classe de l'objet (cf. 6.3). Si une variable d'état visible est initialisée, chaque classe implémentant ce type aura cette variable initialisée. Dans le cas d'une variable référence, l'initialisation (c'est-à-dire la recherche de l'objet dont le nom symbolique est donné à droite du '=', et l'affectation de sa référence à la variable) est effectuée à la création de chaque instance. La recherche de l'objet se fait dans le catalogue courant, c'est-à-dire celui désigné par la référence catal. Si aucun objet n'est trouvé, la variable référence est mise à NIL. Si l'objet trouvé n'est pas conforme au type de la variable référence, l'exception CONFORMANCE_ERROR est levée.

Exemple

```
size : Integer = 10;
toto : REF Circle = "my_circle";
                        // my_circle est un objet du type Circle
```

5.2.2 Variable d'état visible constante

Si une variable d'état visible est déclarée constante, seule l'opération de lecture est définie ; la déclaration d'une telle variable est précédée du mot-clé CONST et doit indiquer sa valeur. Cette variable d'état visible est non modifiable. Dans le cas d'une variable référence, cela signifie que cette variable désignera toujours le même objet.

Exemple

```
CONST toto : REF Circle = "my_circle";
                        // my_circle est un objet du type Circle
```

5.2.3 Syntaxe

<variable d'état visible> =
<ident> { ',' <ident> } ':'(<type_base> | REF <ident_type_construit>)
['=' <expr_constante>]
| CONST <ident> { ',' <ident> } ':'
<type_base> | REF <ident_type_construit>) '=' <expr_constante>

5.2.4 Restrictions

L'utilisation des variables d'état visibles de type String, Array et Record est restreinte de la façon suivante :

- Pour les variables de type Record, on peut lire et écrire uniquement leurs valeurs. On ne peut pas accéder directement aux sous-champs de ces variables.
- Pour les variables de type String, on peut lire et écrire directement leurs valeurs ainsi que les valeurs des éléments (ici des caractères).
- Pour les variables de type Array, on peut lire et écrire directement leurs valeurs ainsi que les valeurs des éléments. Si les éléments des tableaux sont de type String, Array ou Record on ne peut pas écrire directement dans les sous-éléments de ces éléments.
- On ne peut pas initialiser (à la déclaration) les variables de type Array et Record, ni les déclarer constantes.

5.3 Méthodes

L'ensemble des signatures des méthodes définies sur un objet constitue l'interface d'accès à l'objet (les variables d'état visibles en sont un cas particulier).

La signature d'une méthode spécifie son nom, la liste de ses paramètres arguments, précédée du mot-clé IN, la liste de ses paramètres résultats, précédée du mot-clé OUT, les paramètres à la fois arguments/résultats précédés du mot-clé IN_OUT, et la liste des signaux d'exception qu'elle peut générer, précédée du mot-clé SIGNALS.

Chaque paramètre est soit une référence typée soit une variable d'un type élémentaire (type de base ou type structuré) ; dans le premier cas le type de la référence est indiqué précédé du mot-clé REF, dans le deuxième cas c'est uniquement le type élémentaire de

l'objet qui doit être spécifié. Un des résultats peut être déclaré comme la valeur de retour de la méthode.

Exemple : voici deux façons de définir la méthode Prendre du type BoiteAObjets.

```
TYPE BoiteAObjets IS
  METHOD Mettre (IN r : REF Element);
  METHOD Prendre (OUT r : REF Element); SIGNALS boite_vide;
END BoiteAObjets.
```

```
TYPE BoiteAObjets IS
  METHOD Mettre (IN r : REF Element);
  METHOD Prendre : REF aType; SIGNALS boite_vide;
END BoiteAObjets.
```

Remarque : Les identificateurs des arguments et résultats ne sont pas analysés par le compilateur. On peut donc écrire les signatures des méthodes de la façon suivante :

```
METHOD mettre (IN REF Element);
```

Syntaxe

<signature_méthode> = METHOD <ident>
['(' <liste_param> ')']
[':' <type_retour>] ';'
[<liste_signaux>]
<liste_param> = (IN | OUT | IN_OUT) <décl_param>
{ ';' (IN | OUT | IN_OUT) <décl_param> }
<décl_param> =
[<ident> { ',' <ident> } ':'] <type_base> | REF <ident_type_construit>
<type_retour> = <type_base> | REF <ident_type_construit>
<liste_signaux> = SIGNALS <ident> { ',' <ident> } ';'

Méthodes de même nom

Deux méthodes de même identificateur peuvent être définies dans le même type si et seulement si leurs signatures respectives diffèrent par le nombre de leurs arguments et de leurs résultats ou par le fait que l'une a une valeur de retour, et l'autre non.

Cette facilité présente l'avantage de ne pas avoir à utiliser des noms de méthodes différents pour des méthodes de même fonction mais de signatures différentes. Par exemple, la méthode Prendre de l'exemple précédent peut aussi être définie avec deux paramètres.

Exemple

```
TYPE BoiteAObjets IS
  METHOD Prendre : REF Element;
    // pas d'argument, pas de résultat, une valeur de retour
  METHOD Prendre (OUT nbObjetsRestants : Integer) : REF Element;
    // pas d'argument, un résultat, une valeur de retour
END BoiteAObjets.
```

```
TYPE List IS
  METHOD Next;
    // pas d'argument, pas de résultat, pas de valeur de retour
  METHOD Next : REF Top;
```

```
    // pas d'argument, pas de résultat, une valeur de retour
END List.
```

5.4 Construction d'un sous-type

La relation de sous-typage permet d'exprimer la spécialisation de comportement. La relation de sous-typage est simple, un type ne peut avoir qu'un seul super-type. Un sous-type hérite de toutes les définitions faites dans son super-type, il a la possibilité de les surcharger (voir ci-dessous) et de les compléter par de nouvelles définitions de synonymes, de variables d'état visibles ou de méthodes.

La construction d'un type par sous-typage se fait en indiquant le nom du super-type après le mot-clé SUBTYPE OF sur la ligne de l'en-tête de sa déclaration. Lorsqu'on ne donne aucune clause de sous-typage dans l'en-tête d'un type, cela signifie que le type est un sous-type du type prédéfini Top (cf. Annexe C.9).

Syntaxe

<entête_type> = TYPE <ident_type_construit>
[SUBTYPE OF <ident_type_construit>] IS

5.4.1 Surcharge des variables d'état visibles

Aucune surcharge n'est permise sur les variables d'état visibles déclarées dans le super-type.

5.4.2 Surcharge d'une signature

Les signatures des méthodes définies dans le super-type peuvent être surchargées à condition qu'elles restent conformes aux signatures originales (cf. règle de conformité définie dans la section 9). En particulier, il n'est pas possible d'ajouter des signaux, on peut seulement en supprimer.

5.4.3 Exemple

Exemple 1

```
TYPE Personne IS
  age : Integer;
  nom : String[20];
END Personne.

TYPE Etudiant SUBTYPE OF Personne IS
  universite : String[20];
END Etudiant.
```

Dans cet exemple, le sous-typage porte uniquement sur des variables d'état visibles. Le type 'Etudiant' est défini par trois variables d'état visibles : 'âge' et 'nom', hérités du type 'Personne', et 'université' qui est un nouveau champ de ce type.

Exemple 2

```
TYPE Compte IS
  solde : Integer ;
      // solde du compte bancaire;
  METHOD Crediter (IN val : Integer);
      // augmentation du solde de la valeur val;
```

```
  METHOD Debiter (IN val : Integer); SIGNALS opération_impossible;
    // opération de débit de la valeur val; si le compte n'est pas
    // assez approvisionné, l'exception opération_impossible est
    // générée;
END Compte.

TYPE Livret_A SUBTYPE OF Compte IS
  METHOD Calcul_interets (IN taux : Integer);
      // calcul des intérêts et mise à jour du solde;
END Livret_A.
```

Dans l'exemple ci-dessus le type Livret_A est construit par sous-typage du type Compte. Le type Livret_A hérite de la variable d'état visible solde ainsi que des méthodes créditer et débiter du type Compte. Il possède en plus la méthode calcul_intérêts.

6 Classes construites

Une classe est composée d'un en-tête, de la déclaration des variables d'état (cf. 6.3), du code des méthodes (cf. 6.4) et d'une clause de contrôle (cf. 6.7). Dans l'en-tête de la déclaration d'une classe, le mot-clé IMPLEMENTS permet de spécifier le nom du type qu'elle implémente. Si l'on veut donner les caractéristiques communes à toutes les instances de la classe, on peut utiliser la clause ATTRIBUTE (cf. 6.1).

Dans l'exemple ci-dessous, les deux classes 'PetiteBoîte' et 'GrandeBoîte' sont deux réalisations du même type 'BoiteAObjets'.

Exemple

```
TYPE BoiteAObjets IS
  METHOD Concat (IN REF BoiteAObjets) : REF BoiteAObjets;
  METHOD Mettre (IN REF Element); SIGNALS boite_pleine;
  METHOD Prendre (OUT REF Element); SIGNALS boite_vide;
END BoiteAObjets.

CLASS PetiteBoite
  IMPLEMENTS BoiteAObjets IS
  // boîte ne pouvant pas contenir plus de 5 éléments;
  // Caractéristiques des objets de la classe:
  ATTRIBUTE UNMOVABLE;
  // objets non déplaçables (ils restent sur le site de création);
  // variables d'état:
  CONST taille : Integer = 5;
  nb_elt, dernier, premier : Integer;
  contenu : Array [taille] OF REF Element;
  // méthodes:
  METHOD Concat (IN boite: REF BoiteAObjets) : BoiteAObjets;
                 SIGNALS boite_trop_petite;
  BEGIN
   ...
  END Concat ;
  METHOD Mettre (IN elt : REF Element); SIGNALS boite_pleine;
  BEGIN
```

```
    ...
  END Mettre ;
  METHOD Prendre (OUT elt : REF Element); SIGNALS  boite_vide;
  BEGIN
    ...
  END Prendre;
  // clause de contrôle:
  CONTROL
   Concat, Mettre, Prendre : EXCLUSIVE;
END PetiteBoite.

CLASS GrandeBoite
  IMPLEMENTS BoiteAObjets IS
  // boîte ne pouvant pas contenir plus de 100 éléments;
  // variables d'état:
  CONST taille : Integer = 100;
  //suite similaire à la classe PetiteBoîte;
END GrandeBoite.
```

Syntaxe

<classe> = <en-tête_classe>
[<caractéristiques>]
{ <synonymes> }
{ <variable_d'état> }
{ <méthode> | <procédure> }
[<bloc_traitement_exception>]
[<bloc_de_restauration>]
[<contrôle>]
<fin_classe>

<en-tête_classe> = CLASS <ident_classe> [SUBCLASS OF <ident_classe>]
IMPLEMENTS <ident_type_construit> IS

<fin_classe> = END [<ident_classe>] '.'

6.1 Caractéristiques des objets d'une classe

Dans la déclaration d'une classe, on peut indiquer les caractéristiques communes à tous ses objets en utilisant la clause ATTRIBUTE. Il y a actuellement un seul attribut défini : UNMOVABLE. Un objet UNMOVABLE est un objet lié à son site de création et qui ne peut pas être déplacé sur un autre site.

Syntaxe

<caractéristiques> = ATTRIBUTE <caractéristique> { ',' <caractéristique> } ';'

<caractéristique> = UNMOVABLE

6.2 Synonymes

Comme dans un type, la déclaration d'un synonyme consiste à associer une chaîne de caractères à une autre de façon à faciliter l'écriture et la lecture d'un texte Guide.

Une déclaration de synonymes dans la déclaration d'une classe a une portée restreinte à la classe dans laquelle elle est faite et à toutes les sous-classes de cette classe.

6.3 Variables d'état

La déclaration des variables d'état définit le modèle de données des exemplaires de la classe. L'état d'un objet est composé d'une part des variables d'état visibles déclarées dans le type (cf. 5.2) et d'autre part des variables d'état (non visibles) qui sont déclarées dans la classe. Les variables d'état visibles n'ont pas à être redéclarées.

Une variable d'état permet de désigner des objets. Les différents cas de désignation possibles ont été définis dans la section 4 : variable référence typée par un type construit pour désigner des objets externes à celui dans lequel il est déclaré, ou variable simple typée par un type de base pour désigner un objet interne (variable au sens des langages de programmation usuel) créé statiquement dans l'état de l'objet dans lequel elle est déclarée.

Exemple

```
proprietaire : REF Personne; // référence à un objet externe;
nb_elt, dernier,: Integer; // variable du type de base Integer;
contenu : Array [5] OF REF Element;
                          // variable de type de base Array;
```

6.3.1 Variables d'état initialisées

A la déclaration, on peut donner une valeur d'initialisation aux variables d'état. Cette valeur est nécessairement une constante et dépend du type de la variable :

- Booléen : TRUE ou FALSE,
- Caractère : 'a' ou '\078',
- Chaîne de caractères : "chaîne de caractères\n",
- Entier : 10, 0xa, 012,
- Référence : NIL (cf. 4.3.4), SELF (cf. 4.3.3) ou "nom d'objet". Dans ce dernier cas, le compilateur fait appel au serveur de noms lors de la création de l'objet pour initialiser la référence.

Exemple

```
premier : Integer = 1; // variable entière initialisée à 1
person1 : REF Personne = NIL; // référence initialisée à NIL
person2 : REF Personne = "toto";
                           // référence initialisée à la valeur
                           // de la référence d'un objet externe
```

Attention : lors de leur déclaration, on ne peut pas initialiser les tableaux ou les enregistrements.

6.3.2 Variables d'état constantes

La déclaration d'une constante est similaire à celle d'une variable initialisée précédé du mot-clé CONST. Les règles de déclaration et d'utilisation sont les mêmes.

Lorsqu'une variable référence est déclarée constante, cela signifie qu'elle ne pourra jamais désigner un autre objet que celui qu'elle désigne à l'initialisation.

Exemple

```
CONST taille : Integer = 5; // constante du type de base Integer;
CONST grandetaille : Integer = taille*10;
          // déclaration correcte si 'taille' est définie;
CONST person : REF Personne = "toto";
          // référence constante initialisée à un objet externe;
CONST taille : Integer = 5 + 0x10 + 077;
          // variable constante et de classe Integer;
```

Attention : la déclaration de tableaux ou d'enregistrements constants ne présente aucun intérêt dans la mesure où il n'est pas possible de les initialiser lors de leur déclaration.

6.3.3 Syntaxe

<variable_d'état> =
(<ident> { ',' <ident> } ':'
<type_base> | REF <ident_type_construit> ['=' <expr_constante>])
| (CONST <ident> { ',' <ident> } ':'
<type_base> | REF <ident_type_construit> '=' <expr_constante>)

6.4 Méthodes

Les méthodes définissent les points d'accès aux objets de la classe dans laquelle ils sont déclarés. La déclaration d'une méthode est constituée d'un en-tête, de la liste des déclarations de ses variables de travail, ou variables locales, et de la liste des instructions qui constituent son corps. L'en-tête de la déclaration est constitué de la signature de la méthode dans laquelle les noms des arguments, des résultats, et des paramètres à la fois arguments/résultats doivent apparaître. De plus, un bloc de traitement des exceptions peut être défini à la fin du corps de la méthode pour traiter toutes les exceptions engendrées par l'ensemble des instructions du corps.

Exemple

```
METHOD Mettre (IN elt : REF Element) ; SIGNALS boite_pleine ;
BEGIN
  IF nb_elt < taille THEN
    contenu[dernier] := elt ;
    dernier := (dernier+1) MOD taille ;
    nb_elt := nb_elt + 1 ;
  ELSE RAISE boite_pleine;
  END ;
END Mettre ;
```

Syntaxe

<méthode> = <en-tête_méthode>
{ <variable_de_travail> ';' }
BEGIN
<liste_d'instructions>
[<bloc_traitement_exception>]
[<bloc_de_restauration>]
END [<ident>] ';'
<en-tête_méthode> = METHOD <ident>
['(' <liste_param> ')']
[':' <type_retour>] ';'
[<liste_signaux>]
<variable_de_travail> = <variable_d'état>

6.5 Procédures

Les procédures permettent de factoriser le code nécessaire à l'écriture du corps des méthodes définies dans une classe. Une procédure est constituée d'un en-tête qui décrit son nom et sa signature, d'une liste de déclarations de variables de travail (variables locales) et d'un corps composé d'une liste d'instructions.

Les procédures doivent être déclarées au même niveau que les méthodes. Comme les méthodes, on peut déclarer les paramètres à la fois arguments/résultats et un bloc de traitement d'exception associé au corps de la procédure.

Syntaxe

<procédure> = <en-tête_procédure>
{ <variable_de_travail> ';' }
BEGIN
<liste_d'instructions>
[<bloc_traitement_exception>]
[<bloc_de_restauration>]
END [<ident>] ';'
<en-tête_procédure> = PROCEDURE <ident>
['(' <liste_param> ')']
[':' <type_retour>] ';'
[<liste_signaux>]

6.6 Passage de Paramètres

Dans le langage Guide, tous les paramètres sont passés par valeur selon l'un des trois modes IN, IN_OUT ou OUT. Le mode 'paramètre de retour' est un cas particulier du mode OUT.

Restriction

Il est interdit d'utiliser une même variable comme paramètre effectif de plusieurs paramètres OUT (ou IN_OUT) lors d'un même appel de méthode ou de procédure. Cette limitation vient du fait qu'on ne peut pas assurer un même comportement entre une exécution locale et une exécution à distance.

Syntaxe

<liste_param> = (IN | OUT | IN_OUT) <param>
{ ';' (IN | OUT | IN_OUT) <param> }
<param> = <ident> { ',' <ident> } ':'
(<type_base> | REF <type_construit>)
<type_retour> = (<type_base> | REF <type_construit>)

6.6.1 Paramètres IN

Un paramètre de type IN consiste en une variable locale à la procédure ou à la méthode initialisée à la valeur du paramètre effectif. Nous donnons un exemple simple ci-dessous pour montrer l'effet de ce choix :

Exemple

```
PROCEDURE Proc (IN x : Integer);
BEGIN
 x := x + 1;
END Proc;
...
a := 0;
Proc (a); // a = 0 (et non pas 1) après l'appel de procédure proc.
```

6.6.2 Paramètres OUT

Un paramètre de type OUT consiste en une variable locale à la procédure ou à la méthode non initialisée. A la fin de la procédure ou de la méthode la valeur de cette variable est recopiée dans le paramètre effectif. Nous donnons un exemple simple ci-dessous pour montrer l'effet de ce choix :

Exemple

```
PROCEDURE Proc (OUT x : Integer);
BEGIN
 x := 10;
END Proc ;
...
a := 0;
Proc (a) ; // a = 10 après l'appel de procédure proc.
```

6.6.3 Paramètres IN_OUT

Un paramètre de type IN_OUT consiste en une variable locale à la procédure ou à la méthode initialisée à la valeur du paramètre effectif. A la fin de la procédure ou de la méthode la valeur de cette variable est recopiée dans le paramètre effectif. Nous donnons un exemple simple ci-dessous pour montrer l'effet de ce choix :

Exemple

```
PROCEDURE Proc (IN_OUT x, y : Integer);
BEGIN
 x := x + 1;
 y := x;
END Proc ;
...
a := 0; b := 3;
Proc (a, b); // b = 1 (pas 3) après l'appel de procédure proc.
```

6.6.4 Paramètre de retour

Le paramètre de retour permet d'utiliser une méthode ou une procédure sous la forme fonctionnelle. Voir instruction RETURN (cf. 7.13) pour sa manipulation.

Exemple

```
PROCEDURE Double (IN x : Integer) : Integer;
BEGIN
 RETURN 2*x;
END Double;
```

6.7 Conditions d'activation

La clause de contrôle sert à exprimer les conditions de synchronisation entre les appels de méthodes sur chaque objet. Par exemple, on peut spécifier qu'une méthode doit être exécutée en exclusion mutuelle. Les conditions de synchronisation sont des expressions booléennes fonction de l'état de l'objet, des paramètres d'entrée de la méthode et des compteurs de synchronisation associés à chaque méthode. Ces compteurs font partie de l'état des objets et sont aussi visibles dans le code des méthodes et des procédures de la classe. La syntaxe d'utilisation est : nom_compteur (nom_méthode).

Les compteurs définis sont les suivants :

- invoked(m) : nombre total d'appels de la méthode m depuis le début de la vie de l'objet,
- started(m) : nombre d'appels de la méthode m qui ont été acceptés (non bloqués),
- completed(m) : nombre d'exécutions terminées,
- pending(m) : nombre d'activités qui sont en attente d'exécution de la méthode m,
- current(m) : nombre d'appels en cours d'exécution.

Les trois premiers compteurs (invoked, started, completed) sont créés et initialisés à 0 à la création d'un objet. Leur valeur ne peut que croître au cours de la vie de l'objet. Les deux autres compteurs (pending et current) sont calculés à partir des premiers avec les relations :

pending(m) = invoked(m) - started(m),
current(m) = started(m) - completed (m).

A chaque méthode, on associe une clause de contrôle et toutes les clauses sont regroupées en fin de classe après le mot-clé CONTROL. Comme cette association se fait en

donnant le nom de la méthode, il n'est pas possible de définir des clauses de contrôle différentes pour les méthodes de même nom ; c'est la même clause qui s'applique pour toutes les méthodes de même nom. De plus, les seuls paramètres d'entrée qui peuvent apparaître dans la condition d'activation commune sont ceux qui sont communs à toutes les méthodes.

Il est de même possible d'associer une clause de contrôle aux variables d'état visibles puisque chacune correspond à une méthode de lecture et une méthode d'écriture. Comme ces deux méthodes ne sont définies syntaxiquement que par le nom de la variable visible, on ne peut définir qu'une clause de contrôle commune aux deux opérations.

Pour les conditions les plus fréquemment rencontrées dans les programmes, un certain nombre de mots-clés les représentant sont offerts au programmeur :

- EXCLUSIVE (m1, m2, ... mk) : pour désigner la condition d'exclusion mutuelle entre les méthodes mi, c'est-à-dire : Σcurrent(mi) = 0.
- READER (m1, m2, ... mk), WRITER (m'1, m'2, ... m'l) : pour désigner que les méthodes mi et m'j seront utilisées avec une politique de lecture/écriture.

  Une méthode ayant la caractéristique READER ne s'exécute que quand aucune méthode ayant la caractéristique WRITER ne s'exécute.

  Une méthode ayant la caractéristique WRITER ne s'exécute que quand aucune autre méthode WRITER et READER ne s'exécute.

Il peut y avoir plusieurs clauses EXCLUSIVE dans un bloc de contrôle. Si il y a une clause READER, il y a obligatoirement une clause WRITER (et réciproquement). Il peut y avoir plus d'un groupe de clauses READER/WRITER.

Les méthodes surchargées héritent automatiquement de la condition d'activation définie pour la méthode de même nom de la super-classe. Cependant, il est possible de redéfinir les conditions d'activation dans la sous-classe (cf. 6.8.4).

Lorsqu'il n'y a pas de clause définie pour une méthode, il n'y a aucune contrainte de synchronisation pour l'accès à cette méthode.

Syntaxe

<contrôle> = CONTROL
{ <condition_prédéfinie> | <condition_non_prédéfinie> ';' }*
<condition_non_prédéfinie> =
<ident> { ',' <ident> }* ':' < expression_booléenne>
<condition_prédéfinie> =
EXCLUSIVE '(' <liste_méthode> ')'
| WRITER'(' <liste_méthodes> '),' READER '(' <liste_méthodes> ')'
| READER '(' <liste_méthodes> '),' WRITER'(' <liste_méthodes> ')'

Exemple

```
CLASS FixedSizeBuffer
 IMPLEMENTS ProducerConsumer IS
 CONST size : Integer = 10;
 buffer : Array [size] OF REF Element;
 premier, dernier: Integer = 0;

 METHOD Put(IN m: REF Element);
```

```
  BEGIN
  buffer[dernier]:=m;
  dernier:=(dernier+1) MOD size;
  END Put;

 METHOD Get(OUT m: REF Element);
  BEGIN
  m:=buffer[premier];
  premier:= (premier +1) MOD size;
  END Get;

 CONTROL
  Put: (completed(Put) - completed (Get)<size) AND current(Put)=0;
  Get: (completed (Put) > completed (Get)) AND current(Get)=0;
END FixedSizeBuffer.
```

6.8 Construction d'une sous-classe

La construction d'une classe par héritage d'une autre permet de définir des objets plus spécialisés (spécialisation du modèle d'état et des méthodes). C'est aussi un moyen de partager (entre une classe et ses sous-classes) une implémentation commune. Le type implémenté par une sous-classe d'une classe donnée est obligatoirement conforme au type implémenté par cette dernière. L'héritage est simple : une classe a au plus une super-classe.

Syntaxe

<entête_classe> = CLASS <ident_classe>[SUBCLASS OF <ident_classe>]

Lorsque la clause SUBCLASS OF n'existe pas, cela signifie que la classe est une sous-classe de la classe Top qui réalise le type Top (cf. Annexe C.9).

L'héritage porte sur les éléments définis dans la classe héritée (appelée classe mère ou super-classe) : les variables d'état (visibles ou non), les méthodes, la clause de contrôle et les blocs liés aux exceptions : bloc de traitement EXCEPT et bloc de restauration RESTORE. Par contre, les procédures ne sont pas héritées.

La surcharge d'une méthode héritée est possible de différentes façons : surcharge de code sans surcharge de signature, surcharge de signature et de code, surcharge de la clause de contrôle associée à la méthode, et surcharge des clauses liées aux exceptions..

6.8.1 Surcharge du modèle d'état

La définition de nouvelles variables d'état est possible. Les variables d'état (visibles ou non) définies au niveau de la classe mère sont héritées sans surcharge possible.

Exemple

```
CLASS Figure
 IMPLEMENTS Figure IS
  // variables d'état
  vue : REF BitMap ;
END Figure.

CLASS Cercle SUBCLASS OF Figure
```

```
 IMPLEMENTS Cercle IS
  // variables d'état – complément du modèle hérité
  centre : REF Point ;
  rayon : Integer ;
END Cercle.
```

6.8.2 Surcharge d'une signature

La surcharge de signature apparaît dans le type et entraîne forcément une redéfinition du corps de la méthode dans la classe.

6.8.3 Surcharge du code

La surcharge du corps d'une méthode est autorisée indépendamment de la surcharge de sa signature.

Dans le nouveau corps d'une méthode, on ne peut en aucun cas faire appel à une procédure définie dans la classe mère ; les procédures permettent une factorisation de code qui n'est pas visible à l'extérieur de la classe dans laquelle elles sont définies.

Dans la définition d'une sous-classe d'une classe donnée, il est possible d'appeler une méthode de la classe mère qui est surchargée dans la classe courante en utilisant le mot-clé SUPER. Cette possibilité permet un partage de code entre classe mère et classe fille même en cas de surcharge.

Exemple

```
CLASS Refbib
IMPLEMENTS Refbib IS
  numref : Integer;
  nom : String[15];
  METHOD Imprimer ;
  BEGIN
    output.WriteInteger (numref);
    output.WriteString (nom);
  END Imprimer ;
END Refbib.

CLASS Reflivre SUBCLASS OF Refbib
IMPLEMENTS Reflivre IS
  editeur : String[10]; // surcharge de l'état;
  METHOD Imprimer ;     // surcharge de la méthode imprimer;
  BEGIN
   SUPER.Imprimer;
   // impression des champs définis dans la classe Refbib
   // en appelant la méthode imprimer de la super-classe (Refbib);
   output.WriteString (editeur);
  END Imprimer ;
END Reflivre.
```

6.8.4 Surcharge de la clause de contrôle

La clause de contrôle définie dans la super-classe est automatiquement héritée dans la sous-classe. Cependant, il est possible d'avoir une nouvelle clause de contrôle dans la sous-classe, d'une part pour définir la condition d'activation des nouvelles méthodes de la

sous-classe, et d'autre part pour changer la condition d'activation des méthodes héritées ou surchargées.

Lorsqu'une condition d'activation d'une méthode héritée ou surchargée est redéfinie dans la sous-classe, elle remplace complètement celle de la super-classe (il n'y a pas d'ajout de la condition de la sous-classe à celle de la super-classe).

6.8.5 Surcharge des blocs EXCEPT et RESTORE

Une sous-classe hérite toujours des blocs EXCEPT définis dans sa super-classe. Elle peut cependant définir des blocs localement : les traitants de la nouvelle clause EXCEPT seront prioritaires sur ceux définis dans la super-classe.

Pour les blocs RESTORE, le même principe de surcharge que pour la clause de contrôle est appliqué : ils sont hérités sauf s'ils sont redéfinis dans la sous-classe (un nouveau bloc RESTORE remplace complètement celui de la super-classe).

7 Instructions

Une instruction peut être une affectation, un appel de méthode ou de procédure, une instruction conditionnelle ou itérative, l'instruction de parallélisme, les instructions de vérification dynamique de conformité, ou les instructions de retour (avec exception : RAISE, ou normal : RETURN). Il est à remarquer que les points-virgules sont utilisés comme des terminateurs des instructions plutôt que comme des séparateurs. En effet, chaque instruction doit être suivie par un point-virgule.

Syntaxe

<liste_d'instructions> = <instruction> { <instruction> }
<instruction> = (<affectation>
| <appel de méthode>
| <appel de procédure>
| <instruction_IF>
| <instruction_CASE>
| <instruction_TYPECASE>
| <instruction_ASSERTYPE>
| <instruction_FOR>
| <instruction_WHILE>
| <instruction_REPEAT>
| <instruction_CO_BEGIN/CO_END>
| <instruction_BLOCK>
| <instruction_RETURN>
| <instruction RAISE>
| BEGIN <liste_d'instructions> [<bloc_traitement_exception>] END
) ';'

7.1 Affectation et expression

L'affectation est l'opération qui permet de modifier la valeur d'une variable. Une instruction d'affectation a la forme suivante : <variable> ':=' <expression>.

La validité sémantique d'une instruction d'affectation dépend de la nature de la variable identifiée par le désignateur (variable référence ou variable désignant un objet interne) et de la nature de la valeur produite par l'évaluation de l'expression donnée en partie droite. Cette validité est contrôlée statiquement par le compilateur en utilisant la relation d'identité des types pour les types élémentaires et la relation de conformité (voir 9) pour les variables référence. Bien que ce mécanisme de vérification statique soit intéressant pour détecter les erreurs le plus tôt possible, il n'est pas suffisant dans certains cas et un contrôle de type peut être effectué dynamiquement avec l'instruction ASSERTYPE (cf. 7.7).

Pour les types de base, Guide offre une extension à la règle d'identité en permettant l'affectation d'un entier à un caractère (et réciproquement) , d'un entier à un ShortInt (et réciproquement), de tableaux d'éléments de même type mais de taille différente, etc. Les règles d'affectation entre types de base sont définies dans l'annexe B sections 1.6 et 2.4.

Remarque

D'après les règles énoncées ci-dessus, il n'est pas possible d'affecter une variable référence à une variable d'un type élémentaire.

Exemple

```
toto := titi ;
// valide si toto et titi sont soit des références telles que le type
// de titi est conforme à celui de toto, soit des variables de
// type élémentaire identiques (ou compatibles).
bool := bill or ball ;
// bool, bill et ball sont des variables booléennes.
toto := titi.UneMethode (a,b) ;
// la méthode 'Unemethode' a un paramètre retour d'un type
//  compatible avec le type de la variable toto.
```

Syntaxe

<affectation> = <variable> ':=' <expression>
<expression> = <ident>
| <appel de méthode>
| <appel de procédure>

7.2 Appel de méthode

L'appel de méthode sur un objet est l'instruction clé d'un langage à objets tel que Guide. Une telle instruction a la forme suivante : toto.op (paramètres effectifs). Elle signifie l'application de la méthode op sur l'objet désigné au moment de l'exécution par la variable référence toto.

Si la méthode possède une valeur de retour, l'instruction d'appel est une expression.

La validité d'une instruction d'appel de méthode dépend du type de la variable référence toto (la méthode appelée doit être spécifiée dans le type) et de la validité du passage de paramètres (les règles d'affectation définies ci-dessus en 7.1 s'appliquent).

Syntaxe

<appel de méthode> = <variable_référence>’.’<ident>
[’(’<expression> { ’,’ <expression> } ’)’]

7.3 Appel de procédure

L’appel de procédure est une instruction qui permet d’exploiter une factorisation de code faite dans le corps d’une classe. Une instruction d’appel de procédure a la forme suivante : <nom_procédure> [’(’ <paramètres> ’)’].

La validité d’un appel de procédure est controlée par le compilateur qui vérifie d’une part si la procédure dont l’exécution est demandée a été déclarée et d’autre part et si le passage de paramètres est correct (les règles d’affectation définies ci-dessus en 7.1 s’appliquent).

Syntaxe

<appel de procédure> = <ident> [’(’<expression> { ’,’ <expression } ’)’]

7.4 Instruction IF

Exemple

```
IF bool THEN <une action> ;
        ELSE <une autre action> ;
END ;
```

Syntaxe

<instruction_IF> = IF <expression_booléenne> THEN
<liste_d’instructions>
[ELSE <liste_d’instructions>]
[<bloc_traitement_exception>]
END

7.5 Instruction CASE

L’instruction CASE spécifie la sélection et l’exécution d’une suite d’instructions en fonction de la valeur d’une expression.

Syntaxe

<instruction_CASE> = CASE <expression> OF
{ <liste_d’étiquettes> ’:’ <liste_d’instructions> END ’;’ }*
[OTHERS ’:’ <liste_d’instructions>]
[<bloc_traitement_exception>]
END

L’expression dans l’instruction CASE doit être de type Boolean, Char, Integer, ShortInt, LongInt ou String. Les étiquettes sont des expressions constantes de même type que l’expression de l’instruction. Seule la suite d’instructions correspondant à l’étiquette de même valeur que l’expression est exécutée. Si elle est définie, la suite d’instructions

correspondante à l'étiquette OTHERS est exécutée quand aucune autre étiquette n'a la même valeur que l'expression.

Exemple

```
CASE i OF
  1:  <une action> ;
    END ;
  2,3: <une autre action> ;
    END ;
  OTHERS : <une troisième action> ;
END ;
```

7.6 Instruction TYPECASE

L'instruction TYPECASE spécifie la sélection et l'exécution d'une suite d'instructions en fonction du type de l'objet désigné par la variable référence donnée comme sélecteur. Cette instruction ne peut pas s'appliquer pour les variables de type de base.

Syntaxe

<instruction_TYPECASE> = TYPECASE <variable_référence> OF
{ <ident_type> ':' <liste_d'instructions> END ';' }*
[OTHERS ':' <liste_d'instructions>]
[<bloc_traitement_exception>]
END

Le principe de sélection et d'exécution de cette instruction est le même que pour l'instruction CASE. Au lieu de porter sur la valeur de la variable, la sélection porte sur le type de la variable référence : la séquence d'instruction est exécutée si le type de l'objet désigné par <variable_référence> est égal à celui de <ident_type>, les clauses étant évaluées séquentiellement. A l'intérieur de chaque groupe d'instructions sélectionné par le type <ident_type>, la référence <variable_référence> qui sert de sélecteur est considérée comme étant du type <ident_type>. Pour la clause OTHERS, cette référence est utilisée avec le type défini lors de sa déclaration et non avec le type de l'objet effectivement désigné.

Exemple

```
TYPECASE ref_figure OF
  Figure : ref_figure.Afficher (absisse, ordonnee);
      // appel de 'afficher' sur le type 'Figure'
      END ;
  Cercle : ref_figure.Afficher (centre, rayon);
      // appel de 'afficher' sur le type 'Cercle'
      END ;
  OTHERS : <traitement d'erreur> ;
END ;
```

7.7 Instruction ASSERTYPE

Cette instruction est de la forme : x := ASSERTYPE (y). Elle permet de demander l'exécution d'un contrôle de type dynamique entre le type de l'objet désigné par la référence

y et le type de la variable référence x. De plus, si les types sont compatibles, l'affectation x := y est effectuée. L'intérêt d'une telle instruction est montré par l'exemple ci-dessous.

Exemple

```
ref1 : REF Document;
...
ref1 := catal.Search ("Note54");
```

Comme l'objet rendu par la méthode "Search" est déclaré de type "Top" (voir les méthodes définies sur la classe Catal Annexe C.4), le compilateur n'autorise pas l'affectation ci-dessus car l'objet rendu peut être de n'importe type. Cependant, on aimerait bien que l'affectation soit autorisée dans le cas où l'objet est de type Document (ou d'un type conforme à Document). Pour résoudre ce problème, on utilise l'instruction ASSERTYPE :

```
ref1 : REF Document ;
...
ref1 := ASSERTYPE (catal.Search("Note54"));
```

L'instruction ASSERTYPE a un paramètre d'entrée qui est une expression de type référence et un paramètre de retour qui est de type référence. En outre elle peut signaler l'exception 'CONFORMANCE_ERROR'. A l'exécution, le code généré par le compilateur va évaluer l'expression d'entrée et vérifier que son type est le même que (ou est conforme à) celui de la référence de la partie gauche de l'affectation. Si oui, l'affectation de références est effectuée, sinon l'exception 'CONFORMANCE_ERROR' est signalée et la valeur NIL est affectée à la référence.

Limitation

Dans la version V 1.5, on limite la vérification de conformité effectuée par le compilateur lors de l'instruction ASSERTYPE à celle induite par la relation de sous-typage.

Syntaxe

<instruction_ASSERTYPE> = <variable_référence> ':='
"ASSERTYPE" '(' <expression> ')'

7.8 Instruction FOR

Exemple

```
FOR i := 1 TO 10 BY 2 DO
  <une action> ;
END ;
```

Syntaxe

<instruction_FOR> = FOR <ident> ':=' <exp_entière> TO <exp_entière>
[BY <step>] DO
<liste_d'instructions>
[<bloc_traitement_exception>]
END

7.9 Instruction WHILE

Exemple

```
WHILE  bool DO
  <une action> ;
END ;
```

Syntaxe

<instruction_WHILE> = WHILE <expression_booléenne> DO
<liste_d'instructions>
[<bloc_traitement_exception>]
END

7.10 Instruction REPEAT

Exemple

```
REPEAT
  <une action> ;
UNTIL (i = 0);
```

Syntaxe

<instruction_REPEAT> = REPEAT
<liste_d'instructions>
[<bloc_traitement_exception>]
UNTIL <expression_booléenne>

7.11 Instruction CO_BEGIN

L'instruction CO_BEGIN CO_END permet de lancer plusieurs activités en parallèle, l'activité mère étant suspendue pendant l'exécution de cette instruction. Cette instruction possède une expression de terminaison qui spécifie dans quelles conditions l'instruction se termine. Le corps de chacune des activités filles est constitué d'une seule instruction restreinte à l'appel d'une méthode sur un objet.

Syntaxe

<instruction_CO_BEGIN/CO_END> = CO_BEGIN
{ <ident> ':' <appel de méthode> ';' }*
[<bloc_traitement_exception>]
CO_END [<expression_booléenne> | ALL | FIRST]

A chaque nom d'activité est associée une variable booléenne qui prend les valeurs suivantes :

- elle est indéterminée tant que l'activité est en cours d'exécution,
- elle devient fausse dès que l'activité termine de façon anormale (c'est-à-dire en levant une exception),
- et elle devient vraie dès que l'activité termine correctement.

L'activité mère, qui était suspendue à la création des activités filles, est réactivée dès que l'expression booléenne devient vraie. Les activités filles qui sont alors encore actives sont

automatiquement détruites. Les mots-clés ALL et FIRST permettent une écriture plus synthétique de l'expression booléenne de terminaison.

Si la condition de terminaison n'est pas spécifiée, l'instruction parallèle se termine quand toutes les activités filles sont terminées correctement (équivalent au mot-clé ALL).

Si la terminaison en exception d'une activité fille (la variable booléenne associée devient fausse) rend fausse l'expression booléenne du CO_END, l'instruction CO_BEGIN /CO_END se termine en exception (pour plus de précisions sur la terminaison en exception de l'instruction CO_BEGIN/CO_END voir 7.14.5).

Exemple

Une exécution simultanée des méthodes imprimer et dupliquer sur un même objet désigné par une référence de nom rapport peut être effectuée par le bloc suivant :

```
CO_BEGIN
   impression : rapport.Imprimer;
   duplication : rapport.Dupliquer;
CO_END impression AND duplication;  // équivalent à CO_END ALL;
```

L'activité mère reprendra son exécution si les deux activités impression et duplication se sont terminées correctement.

Le bloc parallèle se terminera de façon anormale (en exception) si l'une ou l'autre des deux activités se termine en exception.

7.12 Instruction BLOCK

Cette instruction permet d'empaqueter un morceau de code avec son contexte d'exécution et de l'affecter à une variable référence de type Block (cf. Annexe C 1), afin de pouvoir l'utiliser ultérieurement (notamment en passant la variable en paramètre) soit directement en demandant l'évaluation du bloc à l'aide de l'instruction <réf de bloc>.Value, soit en demandant à un domaine d'exécuter ce bloc à l'aide de l'instruction <réf de domaine>.Start(<réf de Bloc>).

Le contexte d'exécution d'un bloc doit être tel qu'il n'induise pas d'effet de bord lorsqu'il est exécuté. D'autre part, comme ce bloc pourra être exécuté par toute méthode (et donc tout objet) qui aura obtenu sa référence en paramètre, il ne peut pas avoir accès aux variables d'état de l'objet dans lequel il a été créé, car sinon on violerait le principe d'encapsulation des objets. Suite à ces remarques, les règles qui définissent le contexte d'exécution d'un bloc sont les suivantes :

Règles d'utilisation

- Les variables de travail d'un bloc sont obtenues par duplication de celles de la méthode dans lequel il est déclaré ainsi que des paramètres d'entrée de la méthode (IN ou IN_OUT). Ces variables sont initialisées à la valeur qu'elles ont au moment de la création du bloc.
- Il est en outre possible de déclarer des variables locales dans le bloc.
- Les paramètres OUT et les signaux de la méthode ne font pas partie du contexte du bloc (le compilateur vérifie qu'ils ne sont pas utilisés dans le code du bloc.

- Il en est de même pour les variables d'état de l'objet (sauf bien entendu pour les variables d'état visibles, qui, comme toutes les méthodes de l'objet, peuvent être accédées avec la référence SELF).
- On ne peut pas déclarer un bloc dans une procédure.

La définition d'un bloc correspond en fait à la définition d'une classe comportant deux méthodes : une méthode Init exécutée à la création du bloc pour initialiser le contexte, et une méthode Value pour son exécution.

Le traitement des exceptions générées à l'intérieur d'un bloc ne peut être effectué que par les traitants définis dans le bloc ou ceux définis au niveau de la classe.

Syntaxe

<instruction_BLOCK> = <ident> ':=' BLOCK
{<variable_de_travail>}
BEGIN
<liste_d'instructions>
[<bloc_traitement_exception>]
END

Exemple

```
TYPE T IS
x : Integer;
METHOD M (IN i : Integer; IN_OUT io : Integer; OUT o : Integer);
METHOD N (...);
END T;

CLASS C IMPLEMENTS T IS
y : Integer;
METHOD M (IN i : Integer; IN_OUT io : Integer; OUT o : Integer);
z : Integer;
r : REF D;
bloc : REF Block; // bloc est une variable référence de type Block;
BEGIN
...  // code de la méthode;
z := 2;
bloc := BLOCK  // création du bloc et de son contexte;
          // le contexte est constitué des variables i, io et z;
           t : Integer;     // variable locale au bloc;
           BEGIN            // code du bloc;
           t := 0;          // permis;
           t := z;          // permis (z vaut 2);
           z := 0;        // permis (la variable z de la méthode M
                          // est inchangée) ;
           i := i+1;      // permis;
           o := z;        // interdit;
           x :=1;         // interdit;
           y := 1;        // interdit;
           SELF.x := 1;  // permis car x est une variable visible.
           SELF.y :=1;       // interdit;
```

```
            SELF.N;           // permis;
           EXCEPT
           ...               // bloc de traitement d'exception;
           END;              // fin de l'instruction BLOCK
...  // suite du code de la méthode M;
r.op (..., bloc, ...);     // passage en paramètre de la référence
                           // du bloc;
END M ;
...  // code des autres méthodes;
END C.

Class D IMPLEMENTS D IS
//Code de la méthode op;
METHOD op (..., IN b : REF Block, ...);
BEGIN
...
b.Value;    // utilisation du bloc dont la référence a été passée;
            // le code du bloc est exécuté avec le contexte
            // défini à la création du bloc;
...
END op;
END D.
```

7.13 Instruction RETURN

L'instruction RETURN provoque la fin de l'exécution d'une méthode ou d'une procédure. Si cette méthode ou cette procédure possède une valeur de retour, cette dernière doit être spécifiée par l'expression qui suit le mot-clé RETURN.

Exemple

```
METHOD QuiNeFaitRien;
i : Integer;
BEGIN
  RETURN;
  i := 10; // instruction jamais atteinte
END QuiNeFaitRien ;

METHOD QuiRendDeux : Integer ;    // méthode sous forme fonction
BEGIN
  RETURN 2 ;
END QuiRendDeux ;
```

Syntaxe

<instruction_RETURN> = RETURN [<expression>]

7.14 Instructions relatives aux exceptions

7.14.1 Description du mécanisme d'exception

Principe

Le mécanisme d'exception proposé dans Guide [2] permet à toute méthode de signaler par l'instruction RAISE une exception qui fait partie de sa signature. Lorsqu'une exception est signalée, la méthode qui la signale peut éventuellement exécuter un bloc d'instructions de restauration (défini par le mot-clé RESTORE, cf. section 7.14.6) avant de se terminer et de rendre le contrôle à la méthode appelante. Pour l'opération signalante, l'instruction de signalement RAISE a la même sémantique que l'instruction RETURN.

```
Exemple :
METHOD M (...); SIGNALS erreur;
BEGIN
  ...
  IF cond THEN RAISE erreur;
  ...;
RESTORE
  // traitement exécuté juste avant de rendre le contrôle à
  // l'appelant lors de l'exécution de l'instruction RAISE.
END M;
```

A tout retour de méthode se terminant en exception, l'opération appelante peut associer un traitement spécial, appelé traitant (défini par le mot-clé EXCEPT), qui est exécuté avant de passer à l'instruction suivante.

```
Exemple :
// appel de la méthode M définie ci-dessus
BEGIN
  toto.M (...);
EXCEPT
  erreur : output.WriteString("c'est une erreur");
END;
```

Traitant

Un traitant est une instruction Guide, en particulier l'instruction de bloc BEGIN ... END.

Le code du traitant doit être considéré comme équivalent au code des instructions qu'il protège. Il peut donc accéder aux variables d'état de l'objet. Dans le cas où il est déclaré dans une méthode ou une procédure, l'accès aux variables de travail devrait aussi être possible ; cependant, cela est interdit dans la version actuelle (pour des raisons d'implémentation). Si le traitant contient l'instruction RETURN, celle-ci est équivalente à un RETURN exécuté par la méthode ou la procédure dans laquelle est défini le traitant. Il faut donc que le paramètre (s'il y en a un) retourné par le RETURN soit conforme au paramètre formel de retour défini dans l'en-tête de la méthode ou de la procédure. Pour cette raison, il n'est pas possible que l'instruction RETURN apparaisse dans un traitant défini au niveau de la classe.

Le traitant peut enfin utiliser les deux instructions privilégiées REPLACE et RETRY expliquées en détail en 7.14.2.

Bloc de traitement

Le bloc de traitement est un ensemble de traitants commençant par le mot-clé EXCEPT, chaque traitant étant associé à une liste de masques d'exceptions. Le bloc est associé à un ensemble d'instructions, chacune pouvant signaler une ou plusieurs exceptions. Les masques associés à un traitant du bloc précisent parmi toutes les exceptions possibles celles que le traitant peut traiter. On dit que le traitant protège le code auquel il est associé des exceptions qu'il peut traiter.

Un masque d'exception comprend une partie obligatoire donnant un nom d'exception 'nom_exc', et deux parties facultatives donnant un nom de type 'nom_type' et un nom d'opération 'nom_op'. Il indique que le traitant peut être appelé pour traiter une exception du nom 'nom_exc', cette exception ayant été signalée par un appel d'une opération du nom 'nom_op' sur un objet du type 'nom_type' (ou d'un de ses sous-types). Le mot-clé ALL peut être utilisé pour chacune de ces trois parties. ALL est la valeur par défaut pour les deux parties facultatives.

Le mot-clé SYSTEM peut être utilisé à la place du nom de type. Il permet de traiter les exceptions signalées par le système. ALL pour le type inclut les exceptions du système.

```
Exemple :
EXCEPT
 exc1,exc2 : <traitant commun aux exceptions exc1 et exc2>;
 exc3 FROM Atype : <traitant pour l'exception exc3 signalée
                    par un appel d'opération effectué sur un
                    objet de type (ou sous-type de) Atype>;
 exc4 FROM M1.Atype : <traitant pour exc4 signalée
                   par un appel de l'opération M1 effectué
                   sur un objet de type (ou sous-type de) Atype>;
 ALL : <traitant commun à toutes les exceptions>;
END;
```

Association

On peut associer un bloc de traitement d'exception à l'ensemble des instructions d'une structure de bloc, d'une méthode ou procédure, ou d'une classe. Les traitants du bloc sont alors associés à chacun des appels d'opération de cet ensemble. Le bloc associé à un IF, FOR, ..., gère également l'expression de test.

Si l'appel d'une opération signale une exception et que plusieurs traitants, avec des masques adaptés, protègent cet appel, alors on choisit le premier des traitants définis dans un même bloc, ou le traitant du bloc le plus interne dans le cas de blocs imbriqués.

Si aucun traitant n'est prêt à traiter une exception signalée, alors l'opération appelante est terminée et il y a signalement de l'exception système UNCAUGHT_EXCEPTION. Cette exception pourra être rattrapée dans l'appelant de l'appelant par un traitant associé au masque UNCAUGHT_EXCEPTION (FROM SYSTEM).

7.14.2 Instructions particulières

Après l'exécution du traitant, l'appelant reprend à l'opération qui suit l'appel de l'opération signalante. Si l'opération signalante doit rendre une valeur utilisée dans une expression, l'opération suivante est l'évaluation de l'expression. Le traitant peut alors fournir

une valeur de remplacement à la valeur de retour prévue grâce à l'instruction REPLACE. Cette instruction a la même syntaxe que l'instruction RETURN.

La valeur de remplacement fournie doit être conforme au type de la valeur de retour de chacune des opérations susceptibles de signaler une exception traitée par le traitant. La vérification de cette conformité n'est pas faite dans la version 1.5 du compilateur.

Le traitant a également à sa disposition l'instruction RETRY qui demande à l'appelant de reexécuter l'opération signalante au lieu de reprendre à l'instruction suivante.

Exemple

```
CLASS C IMPLEMENTS S IS
toto : REF C;
y : Integer;
METHOD M : Integer; SIGNALS erreur1, erreur2;
...
END M;

// méthode appelant la méthode M
METHOD N;
x : Integer;
BEGIN
  x := toto.M(y);
  EXCEPT
  erreur1 : BEGIN y := 20; RETRY; END; // réexécution de l'appel;
  erreur2 : REPLACE 0; // la valeur de retour de M est 0;
END N;
```

7.14.3 Exceptions dans un traitant

Le traitant peut signaler une exception ou déclarer des blocs de traitement de la même façon que le code qu'il protège. Le signalement d'une exception par le traitant est équivalent au signalement de l'exception par la méthode ou la procédure dans laquelle elle est déclarée, c'est-à-dire, on sort non seulement du traitant mais aussi de l'opération en cours d'exécution. L'exception doit donc être déclarée dans la signature de cette opération.

Dans la version actuelle, il n'est pas possible de signaler une exception dans le bloc de traitant associé à la classe (il faudrait vérifier que l'exception a été déclarée dans la signature de chaque opération au cours de l'exécution de laquelle le traitant est susceptible d'être appelé, ce qui n'est pas fait dans la version 1.5 du compilateur.

Les instructions du traitant bénéficient de la même protection que l'opération signalante, mis à part le traitant lui-même (pour éviter la récursion). Il peut en outre définir des blocs de traitement supplémentaires, prioritaires par rapport aux premiers. En particulier l'instruction RETRY peut ainsi bénéficier de la protection de traitants supplémentaires appelés prioritairement.

7.14.4 Cas particuliers de signalement

Dans le cas où une méthode utilisée dans une expression sous forme de fonction (c'est-à-dire ayant un paramètre retour) signale une exception, l'exécution de l'expression continue après l'exécution du traitant associé à l'exception. Ceci est aussi vrai dans le cas où un paramètre effectif d'un appel de méthode est lui-même un appel de méthode (x.op1(y.op2(

...), ...)). Ceci implique que la valeur de retour rendue après une exception est non significative (l'instruction RETURN n'ayant pu être exécutée avant le RAISE) et qu'il est nécessaire que le traitant associé à cette exception comporte l'instruction REPLACE pour rendre valide la suite de l'exécution de l'expression.

7.14.5 Exceptions et CO_BEGIN

Si une opération d'une branche du bloc parallèle signale une exception, alors l'activité est arrêtée, mais on n'arrête pas pour autant systématiquement les autres branches.

En effet la fonction de terminaison du bloc parallèle décrite au niveau du CO_END exprime la condition de terminaison correcte du bloc à partir de la terminaison correcte de chacune des branches, avec les opérateurs booléens AND et OR. Cette construction permet par exemple de chercher un même résultat en parallèle par deux procédés différents, et de s'arrêter dès qu'une branche l'a trouvé.

Une branche se termine correctement si et seulement si elle ne signale (propage) pas d'exception. Si la terminaison correcte d'une branche amène à satisfaire la fonction de terminaison du bloc, alors les branches encore actives sont arrêtées et on sort normalement du CO_BEGIN. Si la terminaison incorrecte (en exception) d'une branche interdit la terminaison correcte du bloc parallèle (l'expression booléenne reste toujours fausse), alors les branches encore actives sont arrêtées et on sort du CO_BEGIN en signalant l'exception système JOIN_FAILED. Cette exception peut être rattrapée dans l'appelant par un traitant associé au masque JOIN_FAILED FROM SYSTEM.

Remarque : le corps de chaque branche d'un CO_BEGIN (constitué d'un appel de méthode) n'est protégé que par les traitants définis dans le bloc de traitement de la classe..

7.14.6 Restauration

Un bloc de restauration peut être associé à une méthode ou procédure, ou à une classe. C'est une suite d'instructions Guide, sans contrainte particulière, commençant par le mot-clé RESTORE. Le bloc associé à la méthode peut utiliser les variables de travail de la méthode, le bloc associé à la classe ne peut utiliser que les variables d'état de la classe.

Lorsqu'une méthode signale ou propage une exception, avant de rendre le contrôle à la méthode appelante elle exécute l'éventuel bloc de restauration qui lui est associé, puis l'éventuel bloc associé à la classe. Seulement ensuite, le contrôle est rendu à la méthode appelante qui exécute alors le traitant associé à l'exception.

Dans le cas particulier d'une procédure, seul le bloc de restauration associé à la procédure est exécuté, celui associé à la classe ne l'est pas.

Le bloc de restauration associé à une méthode n'est pas hérité. Celui associé à une classe est implicitement hérité si la sous-classe n'a pas de bloc de restauration associé.

Les exceptions sont manipulées dans un bloc de restauration comme elles le sont en dehors. Les traitants de classe protègent les instructions des blocs de restauration associés à la méthode ou à la classe. Toutefois si une exception est signalée ou propagée par le bloc de restauration lui-même, alors il y a propagation de l'exception système RECOVERY_ FAILED. On suppose que l'objet n'a pas pu être remis dans un état cohérent.

7.14.7 Interruptions

Il existe deux manières d'arrêter de l'extérieur un programme Guide. Elles correspondent aux deux signaux Unix SIGINT et SIGQUIT. Le premier est généré par un ^C, il amène à un arrêt immédiat et brutal du processus. Le second est généré par un ^|, mais peut être redéfini par exemple par l'instruction Unix "stty quit ^A". Il amène à un éventuel arrêt en douceur de l'activité, provoqué par le signalement de l'exception système QUIT (FROM SYSTEM).

L'exception système QUIT est traitée comme toutes les autres exceptions. Si elle n'est pas traitée elle se propage en l'exception système UNCAUGHT_EXCEPTION, qui devrait normalement terminer l'activité après avoir dépilé chaque appel imbriqué, et donc après avoir exécuté tous les blocs de restauration définis.

De par son implémentation, l'exception QUIT n'est pas immédiatement générée. C'est le cas en particulier si le programme boucle sur des instructions de calcul ne faisant pas d'appel au système (pas de GuideCall).

Exemple

```
CLASS C IMPLEMENTS C IS
fini : Boolean;
METHOD M;
BEGIN
  fini := FALSE;
  REPEAT
  output.WriteString("taper ^| pour arrêter\n");
  EXCEPT
    QUIT FROM SYSTEM : fini := TRUE;
  UNTIL fini;
END M;
```

7.14.8 Exceptions système

Mises à part les exceptions UNCAUGHT_EXCEPTION, CONFORMANCE_ERROR, RECOVERY_FAILED, JOIN_FAILED et QUIT dont on a déjà parlé, il existe d'autres exceptions qui peuvent être signalées par le système. Toutes les exceptions signalées par le système peuvent être collectivement désignées par le masque ALL FROM SYSTEM.

7.14.9 Syntaxe

Signalement

<instruction_RAISE> = RAISE <ident_exception> ';'

Traitement

<bloc_traitement_exception> = EXCEPT
{ <masque_d'exception> : <traitant> }*
<masque_d'exception> = (<ident_exception> | ALL)
[FROM (<ident_type_construit>["."<ident_méthode>] | SYSTEM | ALL)]
<traitant> = <instruction> ';'

Restauration

[<bloc_de_restauration>] = RESTORE <liste_d'instructions>

7.14.10 Exemple

```
CLASS Buffer IMPLEMENTS Buffer IS
CONST max : Integer = 100;
buffer : Array[max] OF REF Element;
n : Integer = 0;
METHOD Put (IN elt : REF Element); SIGNALS Full;
BEGIN
  IF (n < max)
  THEN
    buffer[n] := elt;
    n := n+1;
  ELSE
    RAISE Full;  // signalement de l'exception
  END;
END Put;
METHOD Get : REF Element; SIGNALS Empty;
BEGIN
  IF (n > 0)
  THEN
    n := n-1;
    RETURN buffer[n];
  ELSE
    RAISE Empty;  // signalement de l'exception
  END;
END Get;
END Buffer.

// exemple de méthode appelant Get :
METHOD Vider_buffer (IN b : REF Buffer);
x : REF Element;
BEGIN
   REPEAT
     x := b.Get;
     x.Destroy;
   UNTIL false;
   EXCEPT               // bloc de traitement associé à la méthode;
     Empty : RETURN;
END Vider_buffer;
```

Lorsque le signal Empty est rendu par l'appel de méthode b.Get, le traitant associé dans la clause EXCEPT est exécuté, c'est-à-dire ici l'instruction RETURN. Elle a pour effet de terminer la méthode Vider_buffer.

8 Généricité

L'introduction de la généricité dans un langage offre au programmeur un moyen de factoriser des traitements en les paramétrant avec les types des données sur lesquels ils s'appliquent. Dans le langage Guide, la généricité permet de définir des constructeurs de types et de classes paramétrés qui seront utilisés pour définir des types et des classes Guide.

Le langage Guide offre un mécanisme de généricité non-contrainte (le paramètre formel de généricité n'est pas typé) et un mécanisme de généricité contrainte (le paramètre formel est typé de façon à imposer et contrôler statiquement que le paramètre effectif est conforme au type du paramètre formel). L'intérêt de la généricité contrainte est de permettre d'utiliser des méthodes sur les variables de type génériques.

Définition : on appelle variable de type générique, une variable déclarée dans un type ou une classe générique et dont le type est un des paramètres de généricité.

Dans les constructeurs de classes, on permet en outre l'utilisation d'un paramètre de dimensionnement constant de type <Integer> initialisé lors de la création d'un objet de cette classe.

La déclaration d'un type générique ressemble à celle d'un type non générique. Pour pouvoir les différencier, on place le mot-clé CONSTRUCTOR juste après le mot-clé TYPE dans la déclaration des types génériques. On doit aussi indiquer les différents paramètres de généricité portant sur des types (voir la section 8.1).

De manière identique, la déclaration d'une classe générique ressemble à celle d'une classe non générique. Pour pouvoir les différencier, on place le mot-clé CONSTRUCTOR juste après le mot-clé CLASS dans la déclaration des classes génériques. On doit aussi indiquer les différents paramètres de généricité portant soit sur la constante de dimensionnement (voir la section 8.2), soit sur des types (voir la section 8.1).

Attention : le super-type (la super-classe) de tout type (toute classe) générique est toujours le type (la classe) Top, il n'y a pas de hiérarchie de types et de classes génériques.

8.1 Généricité sur les paramètres de Type

8.1.1 Généricité non-contrainte

Dans ce type de généricité, le type du paramètre formel de généricité n'est pas indiqué et le type du paramètre effectif de généricité est un des types offerts par le langage.

Le langage Guide permet la généricité non contrainte sur les types suivants : REF aType, Integer, LongInt, ShortInt, Char, Boolean, mais les types structurés ne peuvent pas être utilisés comme paramètres effectifs de généricité.

Dans cette forme de généricité, on ne peut pas appliquer d'appel de méthode sur les variables de type générique (l'appel de méthode n'est pas défini sur le type Integer par exemple, et l'on ne connaît pas a priori le type du paramètre effectif), mais on peut appliquer sur ces variables les opérations définies sur tous les types : affectation, égalité, différence, comparaison (:=, =, #, <, >, <=, >=).

Lors de la déclaration d'une variable référence typée par un constructeur de types, on doit préciser les paramètres effectifs de généricité. Par contre, on ne rappelle pas ces derniers lors de la création de l'instance.

Exemple

```
TYPE CONSTRUCTOR List OF [ T ] IS
  // T est l'identificateur de substitution
  METHOD Insert (IN o : T);
  METHOD Delete : T;
  ...
END List.

CLASS CONSTRUCTOR List OF [ T ] IMPLEMENTS List IS
  // T est l'identificateur de substitution
  CONST max : Integer = 100;
  list : Array[max] OF T;
  dernier : Integer = 0;
  METHOD Insert (IN o : T);
  BEGIN
    list[dernier] := o;    // o est une variable de type générique
    dernier := dernier+1;
  END Insert;

  METHOD Delete : T;
  ...
END List.


/* exemple d'utilisation */
r : REF List OF Integer;
s : REF List OF REF Document;
r := List.New;
s := List.New;
```

8.1.2 Généricité contrainte

Un type générique contraint ne pourra être instancié que par un type conforme au paramètre formel de généricité. Une des conséquences de ce choix est que l'on peut appliquer sur les variables de type générique les méthodes définies pour ce type. Comme il n'est pas possible de définir des sous-types pour les types élémentaires, cette forme de généricité ne s'applique que pour les types construits (les paramétres de généricité sont de la forme REF aType).

Exemple

```
TYPE Document IS
nom : String[20];
END Document.

TYPE CONSTRUCTOR Collection OF [ T : REF Document ] IS
  // T est l'identificateur de substitution ;
  METHOD Search (IN titre : String[20]) : T;
  METHOD Delete : T;
END Collection.
```

```
CLASS CONSTRUCTOR Collection OF [ T : REF Document ] IMPLEMENTS Collection
IS
  // T est l'identificateur de substitution
  CONST max : Integer = 100;
  contenu : Array[max] OF T;
  dernier : Integer = 0;
  METHOD Search (IN titre : String[20]) : T;
  i : Integer;
  BEGIN
    i := 0;
    WHILE (titre # contenu[i].Titre AND i<dernier) DO
/* connaissant le type du paramètre formel de généricité, on peut
appliquer sur la variable une méthode implémentée sur ce type. Titre est
une méthode du type document qui renvoie le titre du document */
      i := i+1;
    END;
    IF i # dernier THEN
      RETURN contenu[i];
    END;
  END Search;
  METHOD Delete : T;
  ...
END Collection.


/* exemple d'utilisation */
r : REF Collection OF REF Rapport; /* Collection est un type générique qui
utilise la généricité contrainte par le type Document. Le type du
paramètre effectif peut être n'importe quel type qui lui est conforme
comme ici Rapport */
r := Collection.New;// création d'un objet collection de rapports.
```

Remarque : Il n'y a aucun rapport entre la généricité et la relation de conformité : par exemple : si le type Rapport est conforme au type Document, on ne peut pas déduire de la relation de conformité que le type List OF Rapport est conforme au type List OF Document.

8.2 Généricité sur les paramètres de dimensionnement

Le paramètre de dimensionnement apparaît uniquement dans la déclaration de la classe générique et il est de type Integer.

Une classe générique peut implémenter soit un type non générique, et dans ce cas, le seul paramètre de généricité est un paramètre de dimensionnement, soit un type générique (avec des paramètres de généricité portant sur les types).

Exemple

```
TYPE Tableau OF [ T ] IS
  METHOD Put (IN i : Integer; o : T);SIGNALS dépassement_de_borne;
  METHOD Get (IN i : Integer) : T; SIGNALS dépassement_de_borne;
END Tableau.
```

```
CLASS CONSTRUCTOR Tableau [size] OF [ T ] IMPLEMENTS Tableau IS
 // size est un identificateur de substitution ;
  contenu : Array[size] OF T;
  METHOD Put (IN i : Integer; o : T);SIGNALS dépassement_de_borne;
  BEGIN
    IF (i < 0) AND (i >= size) then RAISE dépassement_de_borne;
    ELSE contenu[i] := o ;
    END;
  END Put;
  METHOD Get (IN i : Integer) : T; SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= size THEN RAISE dépassement_de_borne;
    ELSE RETURN contenu[i];
    END ;
  END Get;
END Tableau.

TYPE Chaine IS
  METHOD Put (IN i :Integer;o :Char);SIGNALS dépassement_de_borne;
  METHOD Get (IN i : Integer) : Char;SIGNALS dépassement_de_borne;
END Chaine.
CLASS CONSTRUCTOR Chaine [size] IS
 // size est un identificateur de substitution ;
  contenu : String[size];
  METHOD Put (IN i :Integer;o :Char);SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= size then RAISE dépassement_de_borne;
    ELSE contenu[i] := o;
    END;
  END Put;
  METHOD Get (IN i : Integer) : Char;SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= size then RAISE dépassement_de_borne;
    ELSE RETURN contenu[i];
    END;
  END Get;
END Chaine.


/* exemple d'utilisation */
r : REF Tableau OF Integer;
r := Tableau[100].New;

s : REF Chaine;
s := Chaine[100].New;
```

8.3 Syntaxe

<en-tête_type_générique> =
TYPE CONSTRUCTOR <ident_type> OF
’[’ <param_générique> { ’,’ <param_générique> } ’]’ IS
<param_générique> = <param_générique_contraint>
| <param_générique_non_contraint>
<param_générique_contraint> = <ident> ’:’ REF <ident_type_construit>
<param_générique_non_contraint> = <ident>

<en-tête_classe_générique> =
CLASS CONSTRUCTOR <ident_classe> [’[’ <param_dim> ’]’] OF
’[’ <param_générique> { ’,’ <param_générique> } ’]’ IS
<param_dim> = <ident>

9 Règle de conformité

Pour assurer à l’exécution la validité de tout appel de méthode sur un objet, le compilateur effectue un contrôle sur les types pour les instructions suivantes :

- affectation (et passage de paramètres dans les procédures et les méthodes),
- instruction RETURN.

Deux types de contrôle, dépendant de la nature des variables, sont effectués par le compilateur:

- pour les types élémentaires, l’égalité (ou la compatibilité, cf Annexe B 1.6 et 2.4) des types est contrôlée ;
- pour les affectations entre variables référence, la relation de conformité est vérifiée ;

Dans Guide, nous avons choisi de ne pas restreindre le contrôle de types des références à un test d’égalité pour utiliser au mieux les possibilités du modèle à objets, notamment celle du sous-typage : avec une référence de type T, on veut pouvoir désigner tous les objets de type T ainsi que ceux dont le type est sous-type de T, puisque, par définition, ces derniers offrent au moins l’interface définie par le type T. De façon plus générale, on autorise l’accès à un objet avec toute référence dont le type offre l’interface d’accès de l’objet ; si ce dernier possède d’autres opérations que celles définies dans le type de la référence d’accès, ces opérations ne peuvent pas être utilisées avec cette référence. On obtient ainsi un mécanisme de filtre d’accès.

L’expression de la relation de conformité syntaxique est la suivante (enrichie pour prendre en compte les signaux et les paramètres à la fois arguments/résultats) :

Soient deux types Tr et To ; To est conforme à Tr, c’est-à-dire que tout contrôle syntaxique effectué sur Tr est valide pour To, si et seulement si :

- pour chaque opération définie par Tr, il existe une (et seulement une) opération dans To qui lui est conforme ;

- pour chaque signal associé au type Tr, il existe un signal de même nom associé au type To.

Soient deux opérations opo et opr ; opo est conforme à opr si et seulement si :

- opo et opr ont le même nom, le même nombre de paramètres IN, de paramètres OUT et de paramètres IN_OUT ;
- pour chaque signal déclaré dans To, il existe un signal de même nom déclaré dans Tr ;
- le type des paramètres IN de opr est conforme au type des paramètres IN correspondants de opo ;
- le type des paramètres OUT de opo est conforme au type des paramètres OUT correspondants de opr ;
- le type des paramètres IN_OUT de opo est égal au type des paramètres IN_OUT correspondants de opr.

Exemple 1

```
Conformité issue de la relation de sous-type : nous reprenons l'exemple
donné dans la section 5.4.3 :
TYPE Compte IS
  ...
END Compte.

TYPE Livret_A SUBTYPE OF Compte IS
  METHOD Calcul_interets (IN taux : Integer);
END Livret_A.

// Exemple d'utilisation de la conformité entre le type Compte et Livret_A
METHOD Imprimer_solde (IN cpt : REF Compte);
               // la méthode Imprimer_solde accepte en entrée tout
               // paramètre de type Compte ou conforme à Compte
BEGIN
  output.WriteInteger(cpt.solde);
END Imprimer_solde;

// exemple d'appel
METHOD M ;
mon_compte : REF Compte;
mon_livret_A : REF Livret_A;
BEGIN
  mon_compte : = Compte.New;
  mon_livret_A := Livret_A.New;
  ...
  ref.Imprimer_solde(mon_compte);
  ref.Imprimer_solde(mon_livret_A);
   // appel correct car la référence mon_livret_A est d'un type
   // conforme au type du paramètre formel cpt (de type Compte) de
   // la méthode M
END M;
```

Exemple 2

```
utilisation de la conformité dans le cas d'une relation de
restriction.
TYPE Bibliotheque IS
  METHOD Emprunter (IN doc :REF Document;IN qui :REF Emprunteur;
                    OUT status : Integer);
  METHOD Rendre (IN doc : REF Document; IN qui : REF Emprunteur;
                 OUT status : Integer);
  METHOD Ajouter_doc (IN titre, auteur, edition : String;
                      OUT doc : REF Document);
END Bibliotheque.

CLASS Bibliotheque IMPLEMENTS Bibliotheque IS
  // état et code des méthodes
END Bibliotheque.

TYPE Usager_bibliotheque IS
  METHOD Emprunter (IN livre :REF Livre; IN qui :REF Emprunteur;
                    OUT status : Integer);
  METHOD Rendre (IN livre : REF Livre; IN qui : REF Emprunteur;
                 OUT status : Integer);
END Usager_bibliotheque.

(On suppose que le type Livre est un sous-type du type Document).
```

Le type Usager_bibliotheque est une restriction du type Bibliotheque : on ne permet aux usagers que les opérations d'emprunt et de remise. De plus, les seuls documents qu'ils peuvent emprunter sont des livres. D'après la définition de la relation de conformité, le type Bibliotheque est conforme au type Usager_bibliotheque. Ce qui permet d'accéder à la même bibliothèque avec des interfaces différentes.

```
CLASS Administrateur IMPLEMENTS Administrateur IS
  bib : REF Bibliotheque;
  // l'administrateur accède à la bibliothèque avec une référence
  // de type Bibliotheque (toutes les opérations sont permises)
  METHOD Init;
  commande : String[15];
  BEGIN
    bib := ASSERTYPE(catal.Search("bibliotheque_Guide");
    IF bib = NIL THEN
      bib := Bibliotheque.New;
      catal.Insert("bibliotheque_Guide", bib);
    END;
    REPEAT
      input.ReadString(commande);
      CASE commande OF
```

```
        "nouveau_doc" : ...; bib.Ajouter_doc (...); END;
        "consulter_doc" : ... ; bib.Emprunter(...); END;
        ...;
      END;
    UNTIL commande = "fini";
  END Init;
END Administrateur.

CLASS Usager IMPLEMENTS Usager IS
  u_bib : REF Usager_bibliotheque;
      // un usager accède à la bibliothèque avec une référence
      // de type Usager_bibliotheque (accès restreint)
  METHOD Init ;
  commande : String[15];
  bib : REF Bibliotheque;
  BEGIN
    bib := ASSERTYPE(catal.Search("bibliotheque_Guide");
    IF bib = NIL THEN RETURN;
    u_bib := bib;   // affectation valide car le type de la référence
                  // bib est conforme à celle de u_bib
    REPEAT
       input.ReadString(commande);
       CASE commande OF
         "consulter_livre" : ...; u_bib.Emprunter(...); END;
         ...;
       END;
    UNTIL commande = "fini";;
  END Init;
END Usager.
```

Annexe A
Syntaxe du langage Guide

1 Type

<type> = <type_base> | <type_construit>
<type_base> = <type_élémentaire> | type_structuré>
<type_élémentaire> = 'Integer' | 'ShortInt' | 'LongInt' | 'Char' | 'Boolean'
<type_structuré> = type_record | type_array | type_string

<type_construit> = <en-tête_type> | <en-tête_type_générique_non_contraint>
{ <synonymes> }
{ <variable d'état visible> }
{ <signature_méthode> }
<fin_type>

<en-tête_type> = TYPE <ident_type_construit> [SUBTYPE OF <ident>] IS
<en-tête_type_générique> =
TYPE CONSTRUCTOR <ident_type> OF
'[' <param_générique> { ',' <param_générique> } ']' IS
<param_générique> = <param_générique_contraint>
<param_générique_non_contraint>
<param_générique_contraint> = <ident> : REF <ident>
<param_générique_non_contraint> = <ident>
<fin_type> = END [<ident_type_construit>] '.'

<synonymes> = { SYNONYM <ident> '='
(REF <ident_type_construit>) | <type_base> ';' }

<variable d'état visible> = (<ident> { ',' <ident> } ':'
<type_base> | REF <ident_type_construit> ['=' <expr_constante>])
| CONST <ident> { ',' <ident> } ':'
(<type_base> | REF <ident_type_construit>) '=' <expr_constante>

<signature_méthode> = METHOD <ident>
['(' <liste_param> ')']
[':' <type_retour>] ';'
[<liste_signaux>]
<liste_param> = (IN | OUT | IN_OUT) <décl_param>

```
  { ';' ( IN | OUT | IN_OUT ) <décl_param> }
<décl_param> =
      [ <ident> { ',' <ident> } ':' ] (<type_base> | REF <ident_type_construit>)
<type_retour> = (<type_base> | REF <ident_type_construit>)
<liste_signaux> = SIGNALS <ident> { ',' <ident> } ';'
```

2 Classe

```
<classe> = <en-tête_classe>
   [ <caractéristiques> ]
   { <synonymes> }
   { <variable_d'état> }
   { <méthode> | <procédure> }
   [ <bloc_traitement_exception> ]
   [ <bloc_de_restauration> ]
   [ <contrôle> ]
   <fin_classe>
<en-tête_classe> = CLASS <ident_classe> [ SUBCLASS OF <ident_classe> ]
   IMPLEMENTS <ident_type_construit> IS
<fin_classe> = END [<ident_classe>] '.'

<en-tête_classe_générique> =
    CLASS CONSTRUCTOR <ident_classe> ['[' <param_dim> ']']
    [ OF '[' <param_générique> { ',' <param_générique> } ']']
    IMPLEMENTS <ident_type_construit> IS
<param_dim> = <ident>
<param_générique> = <param_générique_non_contraint>
                  | <param_générique_contraint>
<param_générique_non_contraint> = <ident>
<param_générique_contraint> = <ident> ':' REF <ident_type_construit>

<caractéristiques> = ATTRIBUTE <caractéristique> { ',' <caractéristique> } ';'
<caractéristique> = UNMOVABLE

<variable_d'état> = <ident> { ',' <ident> } ':'
     ( <type_base> | REF <ident_type_construit> ) [ '=' <expr_constante> ]
    | CONST <ident> { ',' <ident> } ':'
                ( <type_base> | REF <ident_type_construit>) '=' <expr_constante>
```

```
<méthode> = <en-tête_méthode>
  { <variable_de_travail> ';' }
  BEGIN
   <liste_d'instructions>
   [ <bloc_traitement_exception> ]
   [ <bloc_de_restauration> ]
  END [<ident>] ';'
<en-tête_méthode> = METHOD <ident>
   [ '(' <liste_param> ')' ]
   [ ':' <type_retour> ] ';'
   [ <liste_signaux> ]
<variable_de_travail> = <variable_d'état>

<procédure> = <en-tête_procédure>
   { <variable_de_travail> ';' }
   BEGIN
   <liste_d'instructions>
   [ <bloc_traitement_exception> ]
   [ <bloc_de_restauration> ]
   END [<ident>] ';'
<en-tête_procédure> = PROCEDURE <ident>
   [ '(' <liste_param> ')' ]
   [ ':' <type_retour> ] ';'
   [ <liste_signaux> ]

<liste_param> = ( IN | OUT | IN_OUT ) <param>
   { ';' ( IN | OUT | IN_OUT ) <param> }
<param> = <ident> { ',' <ident> }
                                    ':' (<type_base> | REF <ident_type_construit>)
<type_retour>  = (<type_base> | REF <ident_type_construit>)

<contrôle> = CONTROL
            { <condition_prédéfinie> | <condition_non_préd'finie> ';' }*
<condition_non_prédéfinie> =
          <ident> { ',' <ident> }* ':' < expression_booléenne>
<condition_prédéfinie> =
    EXCLUSIVE '(' <liste_méthode> ')'
    | WRITER'(' <liste_méthodes> '),' READER '(' <liste_méthodes> ')'
    | READER '(' <liste_méthodes> '),' WRITER'(' <liste_méthodes> ')'

<liste_d'instructions> = <instruction> { ';' <instruction> }
```

```
<instruction> = ( <affectation>
      | <appel de méthode>
      | <appel de procédure>
      | <instruction_IF>
      | <instruction_CASE>
      | <instruction_TYPECASE>
      | <instruction_ASSERTYPE>
      | <instruction_FOR>
      | <instruction_WHILE>
      | <instruction_REPEAT>
      | <instruction_CO_BEGIN/CO_END>
      | <instruction_BLOCK>
      | <instruction_RETURN>
      | <instruction_RAISE>
      | BEGIN <liste_d'instructions> [ <bloc_traitement_exception> ] END
     ) ';'

<affectation> = <variable> ':=' <expression>
<expression> = <ident>
      | <appel de méthode>
      | <appel de procédure>

<appel de méthode> = <variable_référence>'.'<ident>
        [ '('<expression> { ',' <expression> } ')' ]

<appel de procédure> = <ident> [ '('<expression> { ',' <expression } ')' ]

<instruction_IF> = IF <expression_booléenne> THEN
       <liste_d'instructions>
       [ ELSE <liste_d'instructions> ]
       [ <bloc_traitement_exception> ]
       END

<instruction_CASE> = CASE <expression> OF
        { <liste_d'étiquettes> ':' <liste_d'instructions> END ';'  }*
        [ OTHERS ':' <liste_d'instructions> ]
        [ <bloc_traitement_exception> ]
       END

<instruction_TYPECASE> = TYPECASE <variable_référence> OF
        { <ident_type_construit> ':' <liste_d'instructions> END ';' }*
```

```
      [ OTHERS ':' <liste_d'instructions> ]
      [ <bloc_traitement_exception> ]
      END

<instruction_ASSERTYPE> = <variable_référence> ':='
                        "ASSERTYPE" '(' <expression> ')'

<instruction_FOR> = FOR <ident> ':=' <exp_entière> TO <exp_entière>
          [ BY <step> ] DO
      <liste_d'instructions>
      [ <bloc_traitement_exception> ]
      END

<instruction_WHILE> = WHILE <expression_booléenne> DO
       <liste_d'instructions>
       [ <bloc_traitement_exception> ]
       END

<instruction_REPEAT> = REPEAT
        <liste_d'instructions>
        [ <bloc_traitement_exception> ]
        UNTIL <expression_booléenne>

<instruction_CO_BEGIN/CO_END> = CO_BEGIN
       { <ident> ':' <appel de méthode> ';' }*
        [ <bloc_traitement_exception> ]
       CO_END [ <expression_booléenne> | ALL | FIRST ]

<instruction_BLOCK> = <ident> ':=' BLOCK
         {<variable_de_travail>}
         BEGIN
         <liste_d'instructions>
         [<bloc_traitement_exception>]
         END

<instruction_RETURN> = RETURN [ <expression> ]
```

<instruction_RAISE> = RAISE <ident_exception> ’;’

<bloc_traitement_exception> = EXCEPT
    { <masque_d’exception> ’:’ <traitant> }*
    END
<masque_d’exception> = (<ident_exception> | ALL)
    [FROM (<ident_type_construit> ["." <ident_méthode>] | SYSTEM | ALL)]
<traitant> = <instruction> ’;’

Annexe B
Types de base Guide

Les types de base sont constitués d'une part des types élémentaires et d'autre part des types structurés.

Les objets dont le type est un type de base (appelés aussi objets de base) ne peuvent pas être permanents, Ils sont toujours créés à l'intérieur d'un autre objet et ne sont pas gérés par le système. Il est donc interdit de déclarer des variables de la forme x : REF Integer ; un objet de base est donc toujours interne à un autre ou objet de travail. A partir des types (et des classes) de base, il n'est pas possible de définir d'autres types (et classes) par le biais des mécanismes de sous-typage et d'héritage.

1 Types élémentaires

Un objet Guide est toujours construit par composition d'autres objets sauf les objets élémentaires dont les types sont décrits ci-dessous. Les méthodes des types élémentaires sont définies à l'aide d'opérateurs, ce qui rend la syntaxe d'appel plus simple et similaire à celle de la programmation classique. Cette syntaxe n'est pas autorisée ailleurs que pour les types et les classes élémentaires ou prédéfinis.

Par exemple, on écrira z := x + y et non z = x.+(y)

1.1 Boolean

```
TYPE Boolean IS
 PREFIX OPERATOR NOT : Boolean ;
 OPERATOR AND (IN Boolean) : Boolean ;
 OPERATOR OR  (IN Boolean) : Boolean ;
 OPERATOR XOR (IN Boolean) : Boolean ;
 OPERATOR = (IN Boolean) : Boolean ;
 OPERATOR # (IN Boolean) : Boolean ;
END Boolean.
```

Deux objets constants de classe Boolean sont prédéfinis ; leurs valeurs sont respectivement 'vrai' et 'faux'. Les noms symboliques prédéfinis 'TRUE' et 'FALSE' permettent au programmeur de les désigner.

1.2 Char

Le type Char permet de déclarer des objets caractères codés en ASCII sur 7 bits. Une constante caractère est désignée entre apostrophes.

```
TYPE Char IS
 OPERATOR = (IN Char) : Boolean ;
```

```
  OPERATOR # (IN Char) : Boolean ;
  OPERATOR < (IN Char) : Boolean ;
  OPERATOR > (IN Char) : Boolean ;
  OPERATOR <= (IN Char) : Boolean ;
  OPERATOR >= (IN Char) : Boolean ;
  OPERATOR + (IN Integer) : Char ; // successeur selon l'ordre ASCII
  OPERATOR - (IN Integer) : Char; // prédécesseur selon l'ordre ASCII
             // ces 2 opérateurs rendent NIL hors des bornes
END Char.
```

1.3 Integer

Le type Integer permet de définir des entiers codés sur 32 bits.

```
TYPE Integer IS
  OPERATOR + (IN Integer) : Integer ; // addition
  OPERATOR - (IN Integer) : Integer ; // soustraction
  OPERATOR POWER (IN Integer) : Integer ; // puissance
  OPERATOR * (IN Integer) : Integer ; // multiplication
  OPERATOR / (IN Integer) : Integer ; // i.e. DIV
  OPERATOR DIV (IN Integer) : Integer ; // division entière
  OPERATOR MOD (IN Integer) : Integer;// reste de la division entière
  OPERATOR = (IN Integer) : Boolean ;
  OPERATOR # (IN Integer) : Boolean ;
  OPERATOR < (IN Integer) : Boolean ;
  OPERATOR > (IN Integer) : Boolean ;
  OPERATOR <= (IN Integer) : Boolean ;
  OPERATOR >= (IN Integer) : Boolean ;
  OPERATOR | (IN Integer) : Integer ;  // opération 'ou' bit à bit
  OPERATOR & (IN Integer) : Boolean ;  // opération 'et' bit à bit
END Integer.
```

1.4 ShortInt

Ce type offre les mêmes opérateurs que le type Integer, mais la taille de ses objets est de 16 bits.

1.5 LongInt

Ce type offre les mêmes opérateurs que le type Integer et la taille de ses objets est de 32 bits.

1.6 Règles d'affectation entre les types élémentaires

Le contrôle de types effectué par le compilateur sur les types élémentaires est basé sur l'identité des types. Cependant, les affectations définies dans le tableau ci-dessous sont autorisées.

Par contre, les affectations entre booléens et caractères ou entiers sont interdites.Il est enfin possible d'affecter un caractère (ou un entier) à un élément de chaîne.

type de la variable	type de l'expression	effet de l'affectation
Integer/LongInt/Short Int	Char	Affectation du code ASCII du caractère à la variable entière.
Char	Integer/LongInt /ShortInt	Les 7 bits de poids faible de l'entier définissent le code ASCII de la variable caractère.
Integer	ShortInt	Affectation des 16 bits du ShortInt à la variable entière, et extension de signe.
ShortInt	Integer	Troncature des bits de poids fort de l'expression entière.
Integer/LongInt	LongInt/Integer	pas de problème car même longueur.

2 Les types structurés

2.1 Array[n] OF ...

Le type structuré Array permet de construire des tableaux multidimensionnels de taille fixe. Les paramètres de taille précisent le nombre d'éléments de chaque dimension (trois dimensions au maximum sont autorisées), les bornes étant toujours de 0 à taille - 1. La syntaxe de déclaration est :

<type_array> = 'Array' '[' <const_entière> { ',' <const_entière> } ']' OF
(<type_base> | REF <nom_type_construit>)

Les opérations définies sont l'affectation de tableaux et l'accès aux éléments de tableaux en lecture et écriture avec la syntaxe classique sur les tableaux.

Exemple

```
tab : Array[20,20] OF Integer;
tab[5,5] := 10;   // opération d'écriture
elem := tab[5,5];  // opération de lecture
```

2.2 Record

Il est très facile de définir des types et des classes Record dans le langage Guide, sans syntaxe supplémentaire. En effet, on peut déclarer un type (et la classe correspondante) ne comportant que des variables visibles (pas de méthodes explicites). Les seules opérations définies sur les objets de ce type sont des lectures et des écritures sur ces variables visibles : cette définition correspond bien à celle d'une structure de Record classique.

Cependant, comme le langage Guide ne permet pas de définir des types et des classes à l'intérieur de types et de classes (notion de types et de classes cachées), on offre le type Record pour déclarer des variables structurées enregistrement. Ce type est réalisé dans la version actuelle par la structure Struct du langage C.

Le format d'instantiation du constructeur Record est celui de Pascal.

<type_record> = 'Record'
{ <ident> ':' (<type_base> | REF <nom_type_construit>) ';' }*
END

Si l'on désire définir une structure Record interne à une classe, pour en générer ensuite plusieurs instances, il faut déclarer cette structure avec la clause SYNONYM.

Exemple

```
SYNONYM livre =  Record
            titre : String [15] ;
            auteur : String [10] ;
            contenu : REF List OF REF Chapitre;
          END ;
x, y : livre;
z : Record
    a : Integer;
    b : String[10];
END;
```

Règle d'affectation

Il est possible d'affecter un Record à un Record si l'affectation champ à champ est correcte du point de vue du contrôle de types : conformité pour les champs de type référence vers un type construit, et égalité pour les champs de type élémentaire.

2.3 String[n]

Le type structuré String[n] permet de créer et d'utiliser des chaînes de caractères de taille fixe. Le paramètre dans les crochets en précise la taille, mais il n'est pas obligatoire dans les paramètres formels des méthodes et des procédures.

Il est possible d'accéder aux caractères d'une variable de type String en sélectionnant le rang du caractère dans la chaîne. Les bornes d'une chaîne sont 0 et n-1 (la longueur n'est pas codée dans la structure).

Les constantes chaînes de caractères sont désignées entre guillemets et ne doivent pas dépasser la taille d'une ligne. Un guillemet à l'intérieur d'une chaîne doit être précédé du caractère '\'.

La syntaxe du type String est :

<type_string> = 'String' [' ['<expression_entière_constante> ']']
<constante_string> = ' " ' <suite_de_caractères> ' " '

La définition de ce constructeur de types est :

```
TYPE CONSTRUCTOR String IS
 OPERATOR = (IN String) : Boolean ;
 OPERATOR # (IN String) : Boolean ;
 OPERATOR < (IN String) : Boolean ;
 OPERATOR > (IN String) : Boolean ;
 OPERATOR <= (IN String) : Boolean ;
```

```
    OPERATOR >= (IN String) : Boolean ;
    OPERATOR +  (IN s1, s2 : String) : String;
                // Concaténation de chaînes
  END String.
```

Exemple

```
      string1 : String[20] ;
      string2 : String[10] = "toto";
      c : Char;
      CONST string3 : String[4] = "tata";
      c := string2[0]; // premier caractère de string2, soit  't';
      METHOD Search (IN nom : String) : REF Element;
```

Dans la version actuelle du compilateur, cette classe est réalisée avec les tableaux de caractères du langage C.

2.4 Règles d'affectation entre les types structurés

Il est possible d'affecter des tableaux (d'éléments de même type) ou des chaînes de dimensions différentes : si l'affectation a lieu dans le sens du plus grand vers le plus petit, il y a troncature. Un message d'avertissement est affiché par le compilateur lorsque les dimensions ne sont pas égales.

Il est possible d'affecter un Array OF Char à une variable de type String et vice versa.

Annexe C
Bibliothèque des Types Guide

Dans cette annexe, nous présentons les différents types construits Guide définis dans la bibliothèque actuellement disponible et qui sont accessibles par le programmeur. Nous donnerons pour chaque type une description des méthodes accessibles à l'utilisateur.

1 Block

Le type Block, utilisé dans l'instruction BLOCK (cf. 7.12) est défini de la façon suivante :

```
TYPE Block IS
  METHOD Value;
END Block.
```

2 Activity

```
TYPE Activity IS
  METHOD Kill;
  // destruction de l'activité
  METHOD Sleep (IN time : Integer);
  // suspend l'activité pendant une durée de 'time' secondes
  METHOD Suspend;
  // arrêt de l'activité
  METHOD Resume;
  // réveil de l'activité
  METHOD Status : Boolean;
  // état de l'activité. Si TRUE, l'activité est active ;
  // si FALSE, l'activité est suspendue ou détruite.
END Activity.
```

3 Domain

Le type Domain permet de rendre visible dans un programme Guide, certaines primitives de la machine virtuelle. Cela permet notamment de détruire son propre domaine (fin d'exécution de son application) par l'instruction : MyDomain.Kill.

```
TYPE Domain IS
  chanIn : REF ChanIn;
  chanOut : REF ChanOut;

  METHOD Kill;
  // destruction du domaine
  METHOD Start(IN block : REF Block);
  /* création de l'activité initiale du domaine. Celle-ci a comme
    canaux d'entrée/sortie les canaux d'entrée/sortie de l'activité qui
    exécute la méthode start. L'activité principale créée exécute le
    bloc passé en paramètre */
  METHOD Suspend;
  // arrêt du domaine
```

```
  METHOD Resume;
  // réveil du domaine
  METHOD Status : Boolean;
  // état du domaine. Si TRUE, le domaine est actif ; FALSE sinon
END Domain.
```

La constante prédéfinie MyDomain contient en permanence une référence de type Domain.

4 Catal

Le type Catal et la constante référence catal permettent d'accéder au catalogue des objets : ce catalogue a pour fonction d'associer à tout objet permanent un nom symbolique fourni par le programmeur, de durée de vie indépendante des programmes.

```
TYPE Catal IS
  father : REF Catal = SELF;
  // donne le répertoire de niveau supérieur ('..' pour Unix)
  root : REF Catal = SELF;
  // donne le répertoire racine ('/' pour Unix)
  nbEntries : Integer = 0;
  // donne le nombre d'objets insérés dans le répertoire

  METHOD ChangeOwner(IN newownername, localname : String) : Boolean;
  // changer le propriétaire d'un objet ou d'un répertoire

  METHOD CreateDir (IN localname, ownername : String; OUT dir : REF
    Catal) : Boolean;
  // créer un nouveau répertoire dans le répertoire courant

  METHOD Delete(IN localname : String; OUT object : REF Top) :
    Boolean;
  METHOD Delete(IN localname : String; OUT object : REF Top);
  METHOD Delete(IN localname : String);
  // détruire l'association entre un objet et un nom symbolique

  METHOD DeleteDir(IN localname : String; OUT dir : REF Catal) :
    Boolean;
  // détruire le répertoire de nom "localname" dans le répertoire
  // courant

  METHOD Find(IN object : REF Top) : String;
  METHOD Find(IN object : REF Top; OUT name : String) : Boolean;
  METHOD RecursiveFind(IN object : REF Top; pathname : String;
    pathtable : Array[10] OF Catal): String;
  // retrouver le nom symbolique de l'objet dont la référence est
  // passée en paramètre

  METHOD Init(IN dad, slash : REF Catal);
  // initialiser ou modifier le père et le répertoire du catalogue

  METHOD Insert(IN localname : String; object : REF Top);
  METHOD Insert(IN localname : String; object : REF Top) : Boolean;
  METHOD Insert(IN localname : String; owner : String; object : REF
    Top);
  METHOD Insert(IN localname : String; owner : String; object : REF
    Top) : Boolean;
  // insérer une nouvelle entrée dans le catalogue

  METHOD List;
```

```
  METHOD List(IN filter : String);
  // lister le contenu du répertoire courant
  // le filtre permet d'en sélectionner un sous-ensemble

  METHOD SearchClass(IN localname : String) : REF Top;
  // rechercher la référence de la classe dont le nom symbolique
  // est passé en paramètre

  METHOD Search(IN localname : String) : REF Top;
  // rechercher la référence de l'objet dont le nom symbolique est
  // passé en paramètre

  METHOD RecursiveSearch(IN localname, pathname : String ; pathtable :
    Array[10] OF REF Catal);
  // lister tous les objets d'une arborescence qui ont le même nom
  // symbolique

  METHOD SearchDir(IN localname : String) : REF Catal;
  // rechercher la référence d'un répertoire local dont le nom
  // symbolique est passé en paramètre.

END Catal.
```

5 Chan

Le type Chan permet d'accéder aux entrées/sorties du système depuis les programmes écrits en Guide. Les méthodes définissent les accès d'entrées/sorties des types élémentaires. Suivant l'implémentation du type Chan choisie, certaines des methodes suivantes n'ont aucun effet.

```
TYPE Chan IS
  /* Les méthodes d'initialisation renvoient le signal NOT_EXIST, si
    le fichier n'existe pas */
  METHOD Init (IN fichier : String);
  // ouvre le fichier spécifié en lecture/ecriture
  METHOD Init(IN fichier : String; beginRead, beginWrite : Boolean);
  // ouvre le fichier spécifié selon le mode suivant :
  //  si beginRead = TRUE, le pointeur de lecture est positionné
  //  au début. A la fin sinon.
  //  si beginWrite = TRUE, le pointeur d'écriture est positionné
  //  au début. A la fin sinon.

  /* toutes les méthodes de lecture renvoient le signal EOF, si l'on
    est en fin de fichier */
  METHOD ReadInteger : Integer;
  // lire le prochain entier rencontré sous forme décimale
  METHOD ReadString : String;
  /* lire la prochaine chaîne de caractères (jusqu'au caractère '\n').
    Celui-ci n'est pas délivré */
  METHOD ReadChar : Char;
  // lire le prochain caractère rencontré
  METHOD ReadBool : Boolean;
  // lire le prochain booléen rencontré
  /* On renvoie TRUE si la chaine de caractères rencontrée est égale
    "TRUE" ou "true" ; FALSE sinon

  METHOD Sync;
  // vider les buffers d'entrée et de sortie
  METHOD Clear;
  // effacer l'écran si celui-ci est compatible vt100
```

```
  METHOD Goto ( IN Integer; Integer );
  /* positionner le curseur à la ligne i et la colonne j si l'écran
    est compatible vt100 */


  METHOD WriteRef( IN r : REF Top );
  // écrire une reference sous la forme : <numéro container>.<oid>
  METHOD WriteInteger( IN i : Integer );
  // écrire un entier sous forme décimale
  METHOD WriteString( IN s : String );
  // écrire la chaîne de caractère
  METHOD WriteChar( IN c : Char );
  // écrire un caractère
  METHOD WriteBool( IN b : Boolean );
  // écrire un booléen.

END Chan.
```

6 ChanIn

Le type ChanIn est une restriction du type Chan, ne comportant que les méthodes d'entrée. La variable d'environnement input est déclarée de ce type.

```
TYPE ChanIn IS
  /* toutes les méthodes de lecture renvoient le signal EOF, si l'on
    est en fin de fichier */
  METHOD ReadInteger : Integer;
  // lire le prochain entier rencontré sous forme décimale.
  METHOD ReadString : String;
  /* lire la prochaine chaîne de caractères (jusqu'au caractère '\n').
    Celui-ci n'est pas délivré. */
  METHOD ReadChar : Char;
  // lire le prochain caractère rencontré
  METHOD ReadBool : Boolean;
  // lire le prochain booléen rencontré.
  // On renvoie TRUE si la chaine de caractères rencontrée est
  //  égale "TRUE" ou "true" ; FALSE sinon.
END ChanIn.
```

7 ChanOut

Le type ChanOut est une restriction du type Chan, ne comportant que les méthodes de sortie. La variable d'environnement output est déclarée de ce type.

```
TYPE ChanOut IS
  METHOD Sync;
  // vider les buffers d'entrée et de sortie
  METHOD Clear;
  // effacer l'écran si celui-ci est compatible vt100
  METHOD Goto ( IN Integer; Integer );
  /* positionner le curseur à la ligne i et la colonne j si l'écran
    est compatible vt100 */
  METHOD WriteRef( IN r : REF Top );
  // écrire une référence sous la forme : <numéro-container>.<oid>
  METHOD WriteInteger( IN i : Integer );
  // écrire un entier sous forme décimale
  METHOD WriteString( IN s : String );
  // écrire la chaîne de caractère
  METHOD WriteChar( IN c : Char );
```

```
  // écrire un caractère
  METHOD WriteBool( IN b : Boolean );
  // écrire un booléen.
END ChanOut.
```

8 Class

```
TYPE Class IS
  /* toutes les méthodes renvoient le signal NOT_EXIST si la classe
    spécifiée n'existe pas */
  METHOD New : REF Top;
  /* créer un objet de la classe spécifiée sur le site d'exécution de
    l'activité */
  METHOD New (IN site : String) : REF Top;
  /* créer un objet de la classe spécifiée sur le site spécifié */
  METHOD New (IN site : String; taille : Integer) : REF Top;
  /* créer un objet de la classe spécifiée sur le site spécifié et de
    la taille donnée (utilisé essentiellement par le compilateur). Si
    la taille spécifiée est inférieure à la taille d'une instance de la
    classe on prend la taille de la classe */
  METHOD New (IN site : String; taille : Integer; flag : Integer) :
    REF Top;
  /* créer un objet de la classe spécifiée sur le site spécifié, de la
    taille donnée (utilisé essentiellement par le compilateur) avec les
    caractéristiques spécifiées (cf. 6.1). Si la taille spécifiée est
    inférieure à la taille d'une instance de la classe on prend la
    taille de la classe */
  METHOD NewOnSite (IN site : String) : REF Top;
  /* créer un objet de la classe spécifiée sur le site spécifié */
  METHOD NewWithSize (IN taille : Integer) : REF Top;
  /* créer un objet de la classe spécifiée de la taille donnée
    (utilisé essentiellement par le compilateur). Si la taille
    spécifiée est inférieure à la taille d'une instance de la classe,
    on prend la taille de la classe */
  METHOD NewWithFlag (IN flag : Integer) : REF Top;
  /* créer un objet de la classe spécifiée avec les caractéristiques
    spécifiéee */
END Class.
```

9 Top

Le type Top est le super-type implicite de tous les autres.

```
TYPE Top IS

  SYNONYM Predicat = Record
    caller : Top ;
    methode : String ;
  END;

  METHOD InitOnNew;
  METHOD GetRef (OUT nb : Integer; listref :  Array OF REF Top);
    // rend le nombre et la liste des références apparaissant dans
    // l'état de l'objet
  METHOD PrintState;
    // affiche l'état de l'objet
```

```
  METHOD PrintCounter;
    // affiche la valeur des compteurs de synchronisation associés
    // à l'objet
  METHOD ObjectSize : Integer;
  METHOD ObjectTypeName : String;
  METHOD ObjectClassName : String;
  METHOD ObjectTypeRef : REF Top;
  METHOD ObjectClassRef : REF Top;
  METHOD Destroy;

END Top.
```

Annexe D
Compilation et exécution

1 Compilation

Une application Guide est constituée d'un ensemble d'objets interagissant les uns avec les autres. La construction d'une application correspond à la définition des types et des classes de ces objets ainsi qu'à la désignation de l'objet initial et de la méthode initiatrice de l'application.

Le compilateur impose que les fichiers des types et les classes à compiler lui soient fournis dans l'ordre dans lequel ils sont utilisés. Un fichier à compiler peut contenir autant de types et/ou de classes que désiré.

Les différents fichiers générés lors de la compilation sont sauvegardés par le compilateur dans le directory .GUIDE/nom_utilisateur/nom_type (ou nom_classe). Le directory .GUIDE se trouve dans le directory défini par la variable d'environnement $GUIDESYS (en général /usr/local/GUIDE).

Dans ce qui suit, nous décrivons brièvement comment utiliser le compilateur GUIDE. Le lecteur trouvera une description précise du fonctionnement du compilateur dans le Manuel utilisateur du compilateur [4].

1.1 Principes de compilation d'un Type

Un type est compilé en deux passes. La première consiste en une analyse lexicale et syntaxique du source. La seconde vérifie la description de ce type par rapport à sa place dans la hiérarchie des types (respect de la règle de surcharge pour un sous-type) ; à l'issue de cette passe, un fichier de description du type est généré et est stocké dans le fichier nom_type/TYPE.

Pour compiler un type, il suffit d'appeler le compilateur Guide de la façon suivante :

glc nom_fichier où nom_fichier est le nom du fichier qui contient le code source du type (ou des types) à compiler.

1.2 Principes de compilation d'une Classe

Une classe est compilée en cinq passes. La première consiste en une analyse lexicale et syntaxique du source. La seconde contrôle si les règles de programmation Guide sont respectées (règle de construction d'une sous-classe, règle de conformité). La troisième produit un ensemble de fichiers qui contiennent du code C : description de la classe, description de son état et code des méthodes ou des procédures. La quatrième passe est la compilation de ce code C et la cinquième le chargement de la classe compilée en MPO.

Les principales options sont :

glc nom_fichier // compilation jusqu'à la passe 3 (production du code C).

glc -p5 nom_fichier // compilation totale (chargement de la classe en MPO).

1.3 Les options du compilateur Guide

Appel du compilateur : glc [option] nom_fichier.

La commande glc sans argument produit la liste des options de compilation.

Options communes pour la compilation des types et des classes

- -p1 : le compilateur n'effectue que la passe 1 (analyse lexicale) sur un fichier contenant une suite de descriptions de types et de classes. Le fichier source est désigné par rapport au répertoire courant. Aucun fichier n'est généré.
- -p2 : le compilateur effectue les passes 1 et 2 (analyse syntaxique et vérification de la conformité) sur un fichier contenant une suite de descriptions de types et de classes. Le fichier source est désigné par rapport au répertoire courant. Pour chaque type, un fichier contenant sa description est généré et il a pour nom .GUIDE/nom_type/TYPE ; pour les classes, le fichier a pour nom : .GUIDE/nom_classe/CLASS

Options pour la compilation des classes

- -p3 le compilateur effectue les passes 1, 2 et 3 (génération du code C) sur un fichier contenant une suite de descriptions de types et de classes. Le fichier source est désigné par rapport au répertoire courant.

  Pour chaque classe compilée, un ensemble de fichiers est généré :

  1. .GUIDE/nom_classe/header.c : contient les directives d'include pour tous les fichiers *.h utilisés (include du système), Il contient en outre la description de la structure d'exécution de la classe.
  2. .GUIDE/nom_classe/D : contient la description de l'état des objets de la classe.
  3. .GUIDE/nom_classe/M : contient le code des méthodes définies dans la classe.
- -p4 : le compilateur effectue les passes 1, 2, 3 et 4 (compilation du code C et résolution des références externes) sur un fichier contenant une suite de descriptions de types et de classes. Pour chaque classe, le fichier généré a pour nom : .GUIDE/nom_classe/g.
- -p5 : le compilateur effectue les passes 1, 2, 3 ,4 et 5 (chargement des classes dans la M.P.O et insertion de celle-ci dans le catalogue de classes) sur un fichier contenant une suite de descriptions de types et de classes. Pour chaque classe, le fichier généré, qui a pour nom : .GUIDE/nom_classe/e, est la copie de celui inséré dans la M.P.O.
- -r : le compilateur garde les fichiers intermédiaires.
- -F : le compilateur force la compilation des types et des classes (même si ils avaient été compilés précédemment).

Par défaut le compilateur est appelé avec l'option -p3.

2 Exécution

Le lancement d'une application se fait à partir de la désignation d'une classe (ou d'une classe et d'un nom de méthode) ; le système crée le domaine correspondant à l'application, crée dans le domaine un objet de la classe désignée puis crée l'activité principale du domaine en lui donnant pour charge d'exécuter la méthode spécifiée (ou la méthode de nom Init si aucune n'est spécifié) sur ce premier objet du domaine. Les types et les classes nécessaires à l'application doivent avoir été compilés avant cette exécution.

Dans la version actuelle, la méthode à exécuter lors du lancement d'une application est une méthode sans paramètre.

L'activité permettant de démarrer une exécution se nomme :

guide

Cette activité permet les commandes suivantes :

- ls : liste des classes et des objets présents dans le catalogue,
- nom_classe : création d'un objet instance de cette classe et activation de la méthode Init.
- nom_classe.nom_méthode : création d'un objet instance de cette classe et activation de la méthode nom_méthode.

Bibliographie

[1] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, ''Design and Implementation of an object-oriented, strongly typed language for distributed applications'', *JOOP*, 3(3), pp. 11-22, September/October 1990.

[2] S. Lacourte, ''Note 61 : Le modèle d'exception de Guide et son implémentation'', *Note interne Guide*, (61), Juillet 1990.

[3] H. Nguyen Van, *Compilation et environnement d'exécution d'un langage à base d'objets*, Thèse, Institut National Polytechnique de Grenoble, Janvier 1991.

[4] M. Riveill, ''Note 53 : Manuel utilisateur du compilateur Guide'', *Note interne Guide*, (53), Janvier 1990.

Manuel du langage Guide (V1.5)

1 Introduction . . . 1
2 Règles de programmation . . . 2
2.1 Jeu de caractères . . . 2
2.2 Identificateurs . . . 2
2.3 Mots-clés . . . 2
2.4 Symboles . . . 3
2.5 Mots réservés . . . 3
2.6 Commentaires . . . 4
3 Programme Guide . . . 4
4 Objets, variables et références . . . 5
4.1 Définitions . . . 5
4.2 Portée d'une variable . . . 6
4.3 Objets et références . . . 6
4.3.1 Création d'un objet . . . 6
4.3.2 Destruction explicite d'un objet . . . 7
4.3.3 Désignation de l'objet courant : SELF . . . 7
4.3.4 Référence constante NIL . . . 7
4.4 Variables d'environnement . . . 8
5 Types construits . . . 8
5.1 Synonymes . . . 9
5.2 Variables d'état visibles . . . 9
5.2.1 Déclaration et initialisation . . . 9
5.2.2 Variable d'état visible constante . . . 10
5.2.3 Syntaxe . . . 10
5.2.4 Restrictions . . . 10
5.3 Méthodes . . . 10
5.4 Construction d'un sous-type . . . 12
5.4.1 Surcharge des variables d'état visibles . . . 12
5.4.2 Surcharge d'une signature . . . 12
5.4.3 Exemple . . . 12
6 Classes construites . . . 13
6.1 Caractéristiques des objets d'une classe . . . 14
6.2 Synonymes . . . 15
6.3 Variables d'état . . . 15
6.3.1 Variables d'état initialisées . . . 15
6.3.2 Variables d'état constantes . . . 16
6.3.3 Syntaxe . . . 16
6.4 Méthodes . . . 16
6.5 Procédures . . . 17

6.6 Passage de Paramètres . . . 17
6.6.1 Paramètres IN . . . 18
6.6.2 Paramètres OUT . . . 18
6.6.3 Paramètres IN_OUT . . . 18
6.6.4 Paramètre de retour . . . 19
6.7 Conditions d'activation . . . 19
6.8 Construction d'une sous-classe . . . 21
6.8.1 Surcharge du modèle d'état . . . 21
6.8.2 Surcharge d'une signature . . . 22
6.8.3 Surcharge du code . . . 22
6.8.4 Surcharge de la clause de contrôle . . . 22
6.8.5 Surcharge des blocs EXCEPT et RESTORE . . . 23
7 Instructions . . . 23
7.1 Affectation et expression . . . 24
7.2 Appel de méthode . . . 24
7.3 Appel de procédure . . . 25
7.4 Instruction IF . . . 25
7.5 Instruction CASE . . . 25
7.6 Instruction TYPECASE . . . 26
7.7 Instruction ASSERTYPE . . . 26
7.8 Instruction FOR . . . 27
7.9 Instruction WHILE . . . 28
7.10 Instruction REPEAT . . . 28
7.11 Instruction CO_BEGIN . . . 28
7.12 Instruction BLOCK . . . 29
7.13 Instruction RETURN . . . 31
7.14 Instructions relatives aux exceptions . . . 32
7.14.1 Description du mécanisme d'exception . . . 32
7.14.2 Instructions particulières . . . 33
7.14.3 Exceptions dans un traitant . . . 34
7.14.4 Cas particuliers de signalement . . . 34
7.14.5 Exceptions et CO_BEGIN . . . 35
7.14.6 Restauration . . . 35
7.14.7 Interruptions . . . 36
7.14.8 Exceptions système . . . 36
7.14.9 Syntaxe . . . 36
7.14.10 Exemple . . . 37
8 Généricité . . . 38
8.1 Généricité sur les paramètres de Type . . . 38
8.1.1 Généricité non-contrainte . . . 38
8.1.2 Généricité contrainte . . . 39
8.2 Généricité sur les paramètres de dimensionnement . . . 40
8.3 Syntaxe . . . 42
9 Règle de conformité . . . 42

Annexe A Syntaxe du langage Guide

1 Type . . . 1
2 Classe . . . 2

Annexe B Types de base Guide

1 Types élémentaires . . . 1
1.1 Boolean . . . 1
1.2 Char . . . 1
1.3 Integer . . . 2
1.4 ShortInt . . . 2
1.5 LongInt . . . 2
1.6 Règles d'affectation entre les types élémentaires . . . 2
2 Les types structurés . . . 3
2.1 Array[n] OF 3
2.2 Record . . . 3
2.3 String[n] . . . 4
2.4 Règles d'affectation entre les types structurés . . . 5

Annexe C Bibliothèque des Types Guide

1 Block . . . 1
2 Activity . . . 1
3 Domain . . . 1
4 Catal . . . 2
5 Chan . . . 3
6 ChanIn . . . 4
7 ChanOut . . . 4
8 Class . . . 5
9 Top . . . 5

Annexe D Compilation et exécution

1 Compilation . . . 1
1.1 Principes de compilation d'un Type . . . 1
1.2 Principes de compilation d'une Classe . . . 1
1.3 Les options du compilateur Guide . . . 2
2 Exécution . . . 3