
Université de Rennes I

ENSSAT Lannion

LSI 2

---

# COMPILATION nilNovi

## SUPPORT DE COURS

Marc Guyomard (& Damien Lolive)

---

Février 2016

*...Comment, alors, améliorer un programme ?*
*les grandes méthodes [...] diviser pour régner,*
*compromis espace-temps, compilation/interprétation*
*sont [...] la clé des améliorations possibles. ...*

Bertrand Meyer – Claude Baudoin

**Résumé**

Ce document est destiné à servir de support à un cours de compilation. Ce cours aborde progressivement les concepts de variable et de structure de contrôle, de procédure et de fonction et enfin d'objet. Il s'appuie successivement sur trois niveaux de langages. Le premier, nilNovi Algorithmique vise la compréhension de la compilation et de l'exécution des expressions, des affectations et des structures de contrôle. Le second, nilNovi Procédural sur-ensemble du premier est concerné par les notions de procédure, de fonction et de passage de paramètre. Le troisième, nilNovi Objet également sur-ensemble du premier illustre la notion d'objet. Une partie du document est consacrée à chaque niveau de langage.

# Première partie
# nilNovi Algorithmique

nilNovi Algorithmique est un langage algorithmique rudimentaire destiné à permettre l'initiation aux techniques élémentaires de la compilation. C'est un langage fortement typé orienté vers la compilation. On trouvera à l'annexe A sa grammaire sous forme BNF. Cette partie du document présente le langage, sa compilation et son exécution.

## 1 Introduction : nilNovi Algorithmique par l'exemple

Un programme nilNovi Algorithmique se compose d'un nom permettant son identification (située entre les mots-clés *procedure* et *is*), d'une partie déclarative (située entre les mots-clés *is* et *begin*) et d'une partie impérative (située entre les mots-clés *begin* et *end*). La partie déclarative comprend la déclaration des variables. Les seuls types autorisés sont les entiers (*integer*) et les booléens (*boolean*).

### 1.1 Exemple 1

```
01 : procedure pp is
02 :    i,som : integer ;
03 : begin
04 :    som :=0 ;
05 :    get(i) ;
06 :    while i/=0 loop
07 :       //som est la somme des valeurs paires lues
08 :       if i-(i/2)*2=0 then
09 :          som :=som+i
10 :       end ;
11 :       get(i)
12 :    end ;
13 :    put(som)
14 : end.
```

**Commentaires**

1. Le programme *pp* ci-dessus lit une liste de valeurs s'achevant par 0, calcule la somme des valeurs paires de la liste puis affiche le résultat.

2. La partie déclarative se limite à la ligne 02. La partie impérative est constituée des lignes 04 à 13.
3. La ligne 02 déclare deux variables entières $i$ et *som*.
4. La ligne 04 constitue une instruction d'affectation. La ligne 05 est un exemple d'utilisation de l'instruction de lecture (*get*) de la variable $i$ sur le périphérique standard. Le paramètre de l'instruction *get* est toujours une variable entière.
5. Dans la partie déclarative, on note que la séquentialité entre deux instructions est symbolisée par l'opérateur « ; ».
6. Les lignes 06 à 12 représentent une boucle *while*. La sémantique de ce type de boucle n'est pas rappelée ici. C'est le seul type de boucle existant en nilNovi Algorithmique.
7. Les lignes 08 à 10 représentent une alternative simple. La sémantique de cette structure de contrôle n'est pas rappelée ici. Il existe une seconde forme d'alternative : l'alternative double, qui sera illustrée plus tard.
8. La ligne 07 contient un commentaire « ligne ». Ce type de commentaire débute par le délimiteur // et s'achève à la première fin de ligne rencontrée. Un commentaire ne peut débuter à l'intérieur d'un identificateur ni à l'intérieur d'un mot-clé ou d'un délimiteur.
9. Les lignes 06 et 08 contiennent des expressions booléennes :

   $$i/=0 \text{ et } i-(i/2)*2=0$$

   Il s'agit d'expressions relationnelles : elles sont chacune constituées de deux expressions arithmétiques séparées par un opérateur relationnel ($/=$ et $=$). Les opérateurs $/=$ et $=$ sont disponibles pour tous les types (*integer* et *boolean*). Dans le cas de booléens, ces deux opérateurs sont définis par :

   $$a = b \Leftrightarrow (a \textbf{ and } b) \textbf{ or } (\textbf{not } a \textbf{ and } \textbf{not } b)$$

   et par :

   $$a/=b \Leftrightarrow (a \textbf{ and } \textbf{not } b) \textbf{ or } (\textbf{not } a \textbf{ and } b)$$

   par contre les autres opérateurs relationnels ($<, <=, >, >=$) ne sont autorisés que pour le type *integer*. Dans une expression arithmétique la priorité des opérateurs binaires est donnée par le tableau suivant :

| opérateur | priorité |
|---|---|
| * | 2 |
| / | 2 |
| + | 1 |
| - | 1 |

   En présence de deux opérateurs de même priorité, l'évaluation se fait de gauche à droite. Il est possible de forcer la priorité en utilisant des parenthèses.

10. La ligne 13 contient une instruction d'écriture (*put*). Le paramètre d'une telle instruction est une expression entière.
11. *pp* n'est une véritable procédure dans la mesure où il est impossible d'appeler *pp.*

**Ce qu'il faut retenir de cet exemple**

1. Le langage nilNovi Algorithmique permet de construire des algorithmes en composant des constructions élémentaires (affectations et opérations d'entrée/sortie) à l'aide de la séquentialité, de boucles et d'alternatives.
2. Les seuls types de données sont les types entier (*integer*) et booléen (*boolean*). Les variables sont déclarées et typées.
3. Un identificateur de variable doit être déclaré avant d'être utilisé.
4. Un identificateur est constitué de lettres (majuscules ou minuscules) ou de chiffres. Il débute obligatoirement par une lettre. La casse est significative.

## 1.2 Exemple 2

```
01 : procedure pp is
02 :    b : boolean ;
03 :    i, j : integer ;
04 : begin
05 :    get(i) ;
06 :    get(j) ;
07 :    b:=(i<=23)=(j>12) ;
08 :    if b and i>2 then
09 :       b :=not b or i=20 ;
10 :       if b then
11 :          put(i*3)
12 :       end
13 :    else
14 :       put(-34)
15 :    end
16 : end.
```

**Commentaires**

1. À la ligne 02 est déclarée la variable booléenne $b$, à la ligne 03 sont déclarées les variables entières $i$ et $j$.
2. Aux lignes 07 et 09 se trouvent deux affectations de la variable booléenne $b$ par une expression booléenne.

3. Aux lignes 08 à 15 on note l'imbrication de deux alternatives, la plus externe est une alternative double, la plus interne une alternative simple.
4. À la ligne 07 on trouve une expression booléenne construite à partir de l'opérateur = . Celui-ci, ainsi que l'opérateur /= peuvent être utilisés pour comparer deux valeurs booléennes. Ces deux opérateurs relationnels permettent de construire une expression booléenne à partir soit de deux expressions booléennes soit de deux expressions entières. Par contre les autres opérateurs relationnels (<, <=,...) sont réservés aux opérandes entiers.
5. Dans une expression booléenne l'opérateur *and* est prioritaire par rapport à l'opérateur *or*.

# 2 Rattachement d'un identificateur à une entité

## 2.1 Introduction

Dans nilNovi Algorithmique un identificateur est associé par sa déclaration à une entité. Cette entité est en général une variable, mais il existe un cas particulier : l'identificateur de la procédure principale. Sur quelle portion de texte peut-on utiliser l'identificateur pour désigner l'entité en question ? C'est la notion de portée.

**Exemple :**

```
01 : procedure p is
02 :     p,i : integer;
03 :     i : boolean;
04 : begin
05 :     p :=0;
06 : end.
```

À la ligne 02 l'identificateur $p$ est autorisé car il n'y a pas de conflit possible avec le nom de la procédure principale. À la ligne 05 $p$ désigne la variable déclarée à la ligne 02. La ligne 03 présente une erreur : $i$ ne peut désigner simultanément deux entités différentes.

## 2.2 Cas de l'identificateur de la procédure principale

**Portée.** Compte tenu qu'il n'est jamais nécessaire de désigner l'entité programme principal (l'appel de cette procédure est interdite dans le langage), on considère que la portée de l'identificateur de la procédure principale est vide.

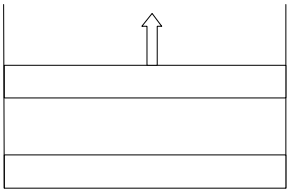
### 2.3 Cas des identificateurs de variables

**Portée.** La portée d'un identificateur de variable est le texte du programme à partir de l'occurrence de la déclaration.

**Conflit.** Un conflit est possible (déclaration multiple) avec une autre déclaration de variable. Dans ce cas le programme est erroné.

## 3 Attribution d'adresses

En préambule à cette section il faut savoir que dans un langage tel que nilNovi Algorithmique, les variables sont gérées dans une pile : la pile d'exécution. Celle-ci est également utilisée pour mémoriser les résultats intermédiaires lors de l'évaluation d'expressions.




Lors de la rencontre de la déclaration d'un identificateur de variable, le compilateur associe une adresse à l'identificateur. Compte tenu du caractère statique du schéma d'exécution du langage nilNovi Algorithmique, cette adresse est une adresse absolue dans la pile[1]. L'attribution se fait séquentiellement en partant de zéro (adresse de l'emplacement de la base de la pile). Lors de l'occurrence d'un identificateur déjà déclaré, il est facile de retrouver son adresse.

**Exemple 1.**

```
01 : procedure pp is
02 :    i,som : integer ;
03 :    b : boolean ;
04 : begin
05 :    som :=0 ;
06 :    i :=23 ;
07 :    b :=i+som = som+i
08 : end.
```

1. Nous verrons qu'il n'en est pas toujours ainsi. Dès que l'on introduit la possibilité d'opérations (procédures ou fonctions), il est en général impossible de connaître à la compilation l'adresse qu'aura une variable dans la pile. Il faut alors passer par une adresse intermédiaire (l'adresse statique) entre l'adresse symbolique (l'identificateur) et l'adresse d'exécution (dite adresse « dynamique »).

Les trois variables $i$, *som* et $b$ sont respectivement associées aux trois adresses 0, 1, 2.

# 4 La machine nilNovi Algorithmique

La machine nilNovi Algorithmique est constituée de deux parties que allons présenter successivement : la partie « données » et la partie « programme ».

La partie « données » se compose d'une pile (*pile*) et de son pointeur (le registre *ip*) qui constituent la pile d'exécution. Elle est destinée à mémoriser les variables, elle permet également d'évaluer les expressions.

La partie « programme » est destinée à recevoir le programme objet à des fins d'exécution. Elle est composée d'un tableau *po* constituant la mémoire de programme et d'un registre *co* : le compteur ordinal.

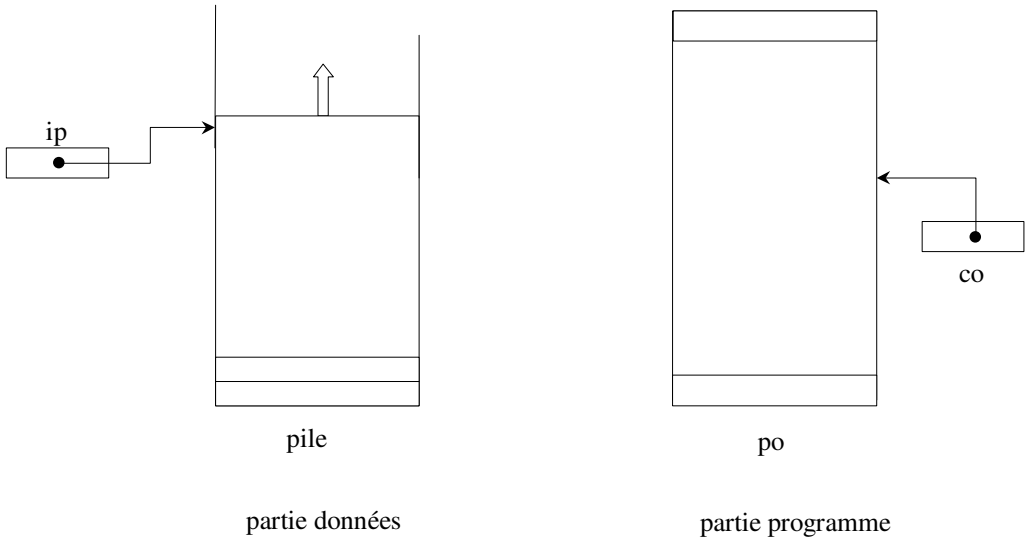



Figure 1 – Structure de la machine nilNovi Algorithmique

# 5 Jeu d'instructions de la machine nilNovi Algorithmique

Afin de ne pas alourdir la description, aucun contrôle (débordement de pile, division par 0, etc.) n'est spécifié.

Le nombre noté entre parenthèses (par exemple (1) après *debutProg*) désigne le code machine de l'instruction. Dans la suite ni l'initialisation ni l'évolution du compteur ordinal ne sont prises en compte excepté dans les instructions de contrôle.

La suite de cette section spécifie les instructions de la machine nilNovi Algorithmique sous la forme pré/post. Les instructions sont classées par famille (entrée/sortie, variables et affectation, etc.).

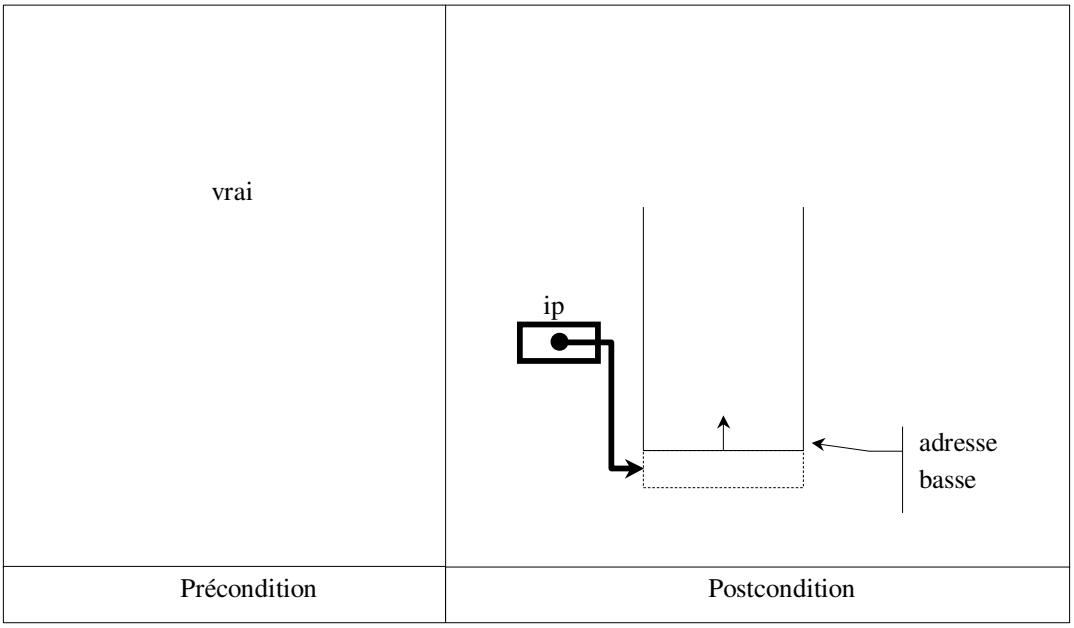
L'initialisation de la machine place l'adresse de la première instruction machine dans le compteur ordinal.

## 5.1 Unité de compilation

### 5.1.1 débutProg (1)

**procedure** debutProg() ;

Cette instruction est générée au début d'un programme nilNovi Algorithmique. Elle permet d'initialiser la structure de données (pile, registres).




### 5.1.2 finProg (2)

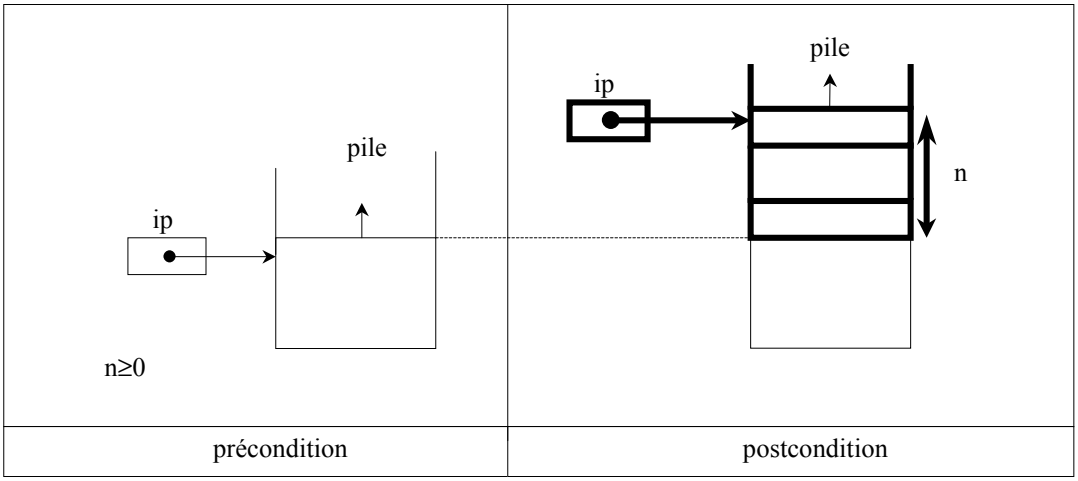
**procedure** finProg() ;
Cette instruction arrête la machine nilNovi Algorithmique.

## 5.2 Variables et affectation

### 5.2.1 reserver (3)

**procedure** reserver(n : integer) ;

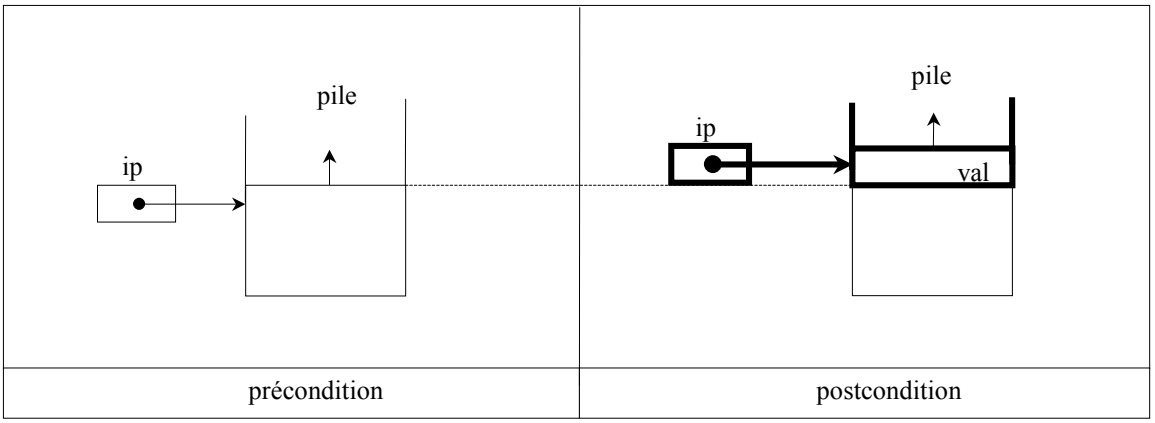
Cette instruction a pour but de réserver $n$ emplacements pour $n$ ($n > 0$) variables au sommet de la pile d'exécution.




### 5.2.2 empiler (4)

**procedure** empiler(val : integer) ;

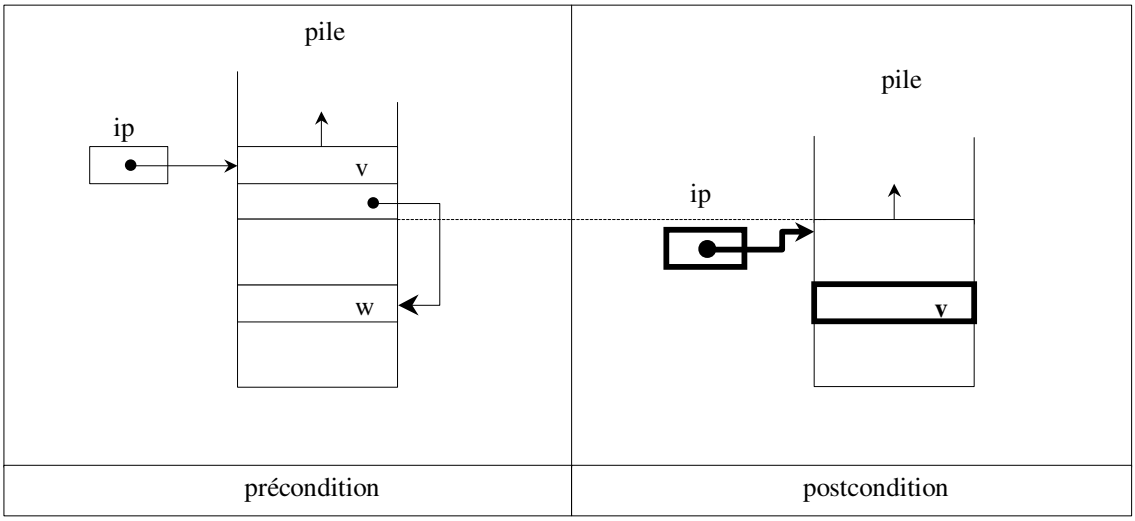
Cette instruction est utilisée pour empiler les adresses des variables ainsi que les valeurs présentes dans une expression.




### 5.2.3 affectation (6)

**procedure** affectation() ;

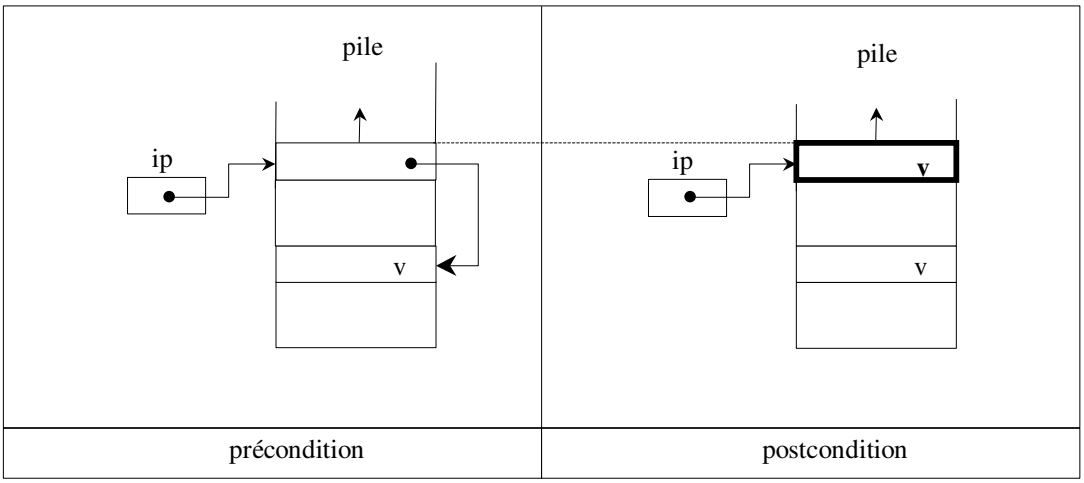
Cette instruction place la valeur située en sommet de pile à l'adresse désignée par l'emplacement sous le sommet.




### 5.2.4 valeurPile (7)

**procedure** valeurPile() ;

Cette instruction remplace le sommet de pile par le contenu de l'emplacement désigné par le sommet.



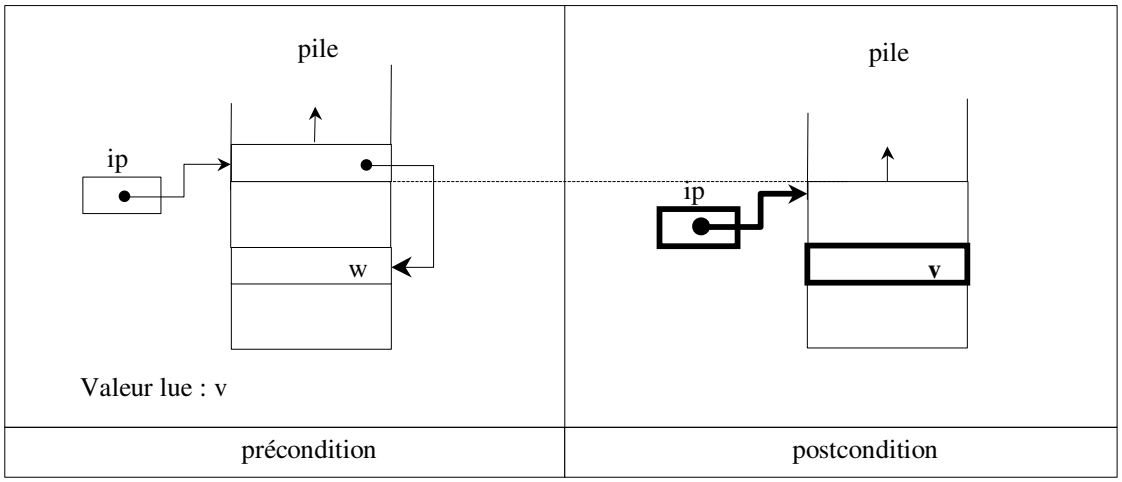

## 5.3 Entrées-Sorties

Il existe deux instructions dans cette famille : une pour les entrées (clavier) et une pour les affichages écran.

### 5.3.1 get (8)

**procedure** get() ;

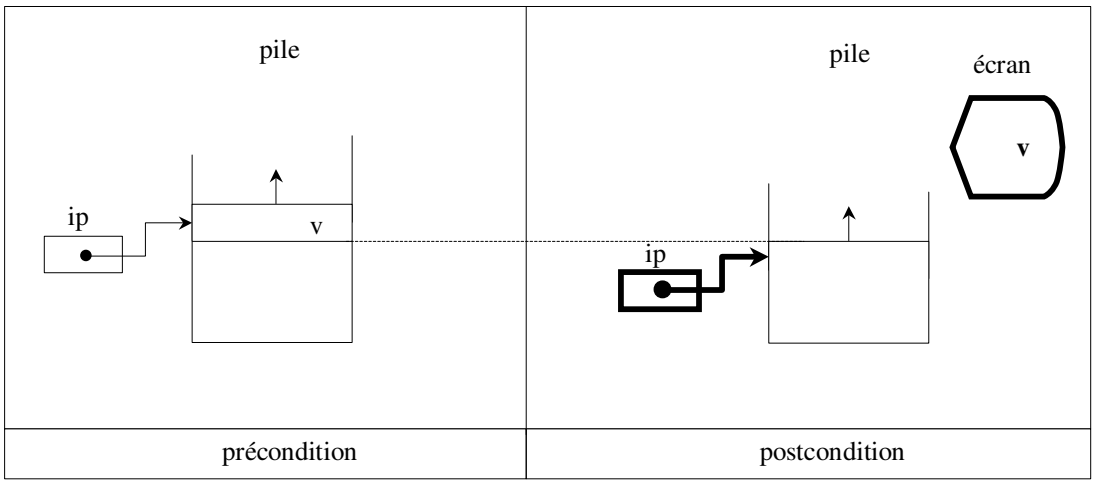
Cette instruction permet de placer la valeur lue sur le clavier dans la variable qui est désignée par le sommet de pile.




### 5.3.2 put (9)

**procedure** put() ;

Cette instruction permet d'afficher la valeur présente au sommet de la pile.



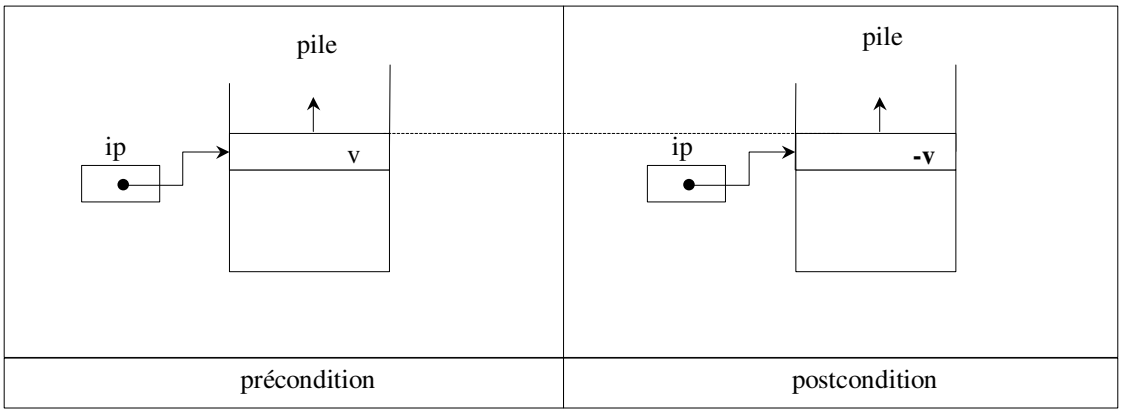

## 5.4 Expressions arithmétiques

Il existe une instruction machine unaire, *moins,* et quatre instructions machine binaires : *mult, add, sous* et *div*. Les instructions binaires opèrent sur les deux éléments en sommet de pile.

### 5.4.1 moins (10)

**procedure** moins() ;

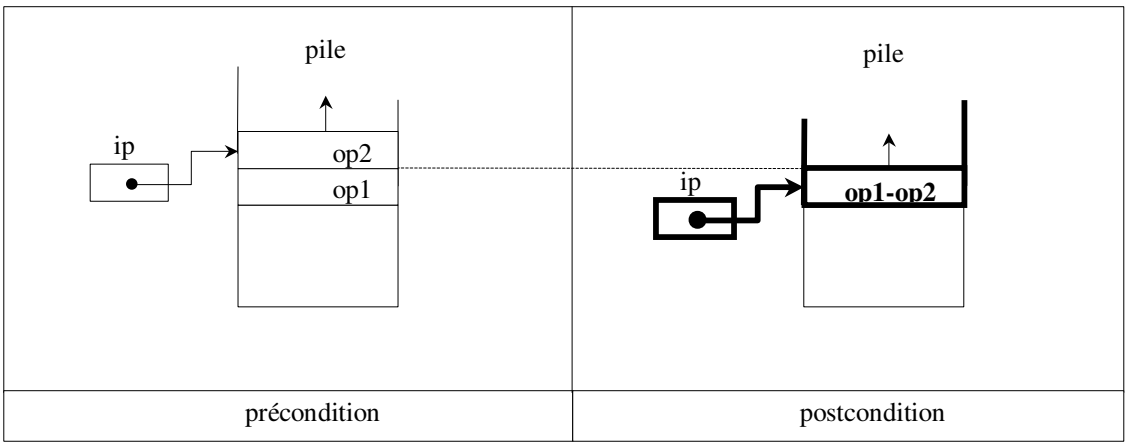
Cette instruction permet de calculer l'opposé de la valeur entière en sommet de pile.




### 5.4.2 sous (11)

**procedure** sous() ;

Cette instruction permet de calculer la différence entre les deux valeurs au sommet de pile.




### 5.4.3 add (12), mult (13), div(14)

**procedure** add() ;
**procedure** mult() ;
**procedure** div() ;

Ces instructions sont similaires à l'instruction *sous* pour respectivement les opérations d'addition, de multiplication et de division. Leur sémantique pré/post n'est pas présentée ici.

## 5.5 Expressions relationnelles et booléennes

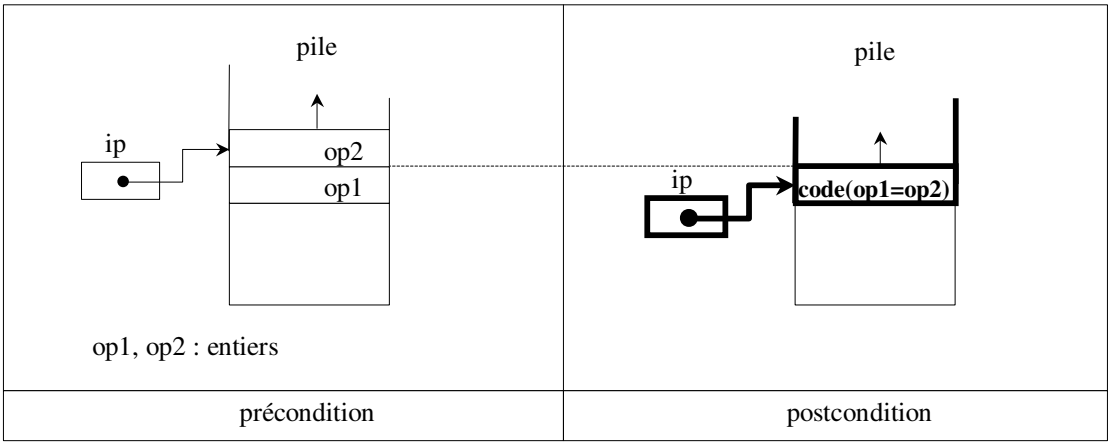
Il existe six instructions relationnelles : *egal, diff, inf, infeg, sup* et *supeg.* Elles correspondent respectivement aux opérateurs $=, / =, <, <=, >, >=$. Leurs opérandes sont des entiers. Il existe deux instructions booléennes binaires : *et* et *ou.* Elles correspondent aux opérations *and* et *or*. On rappelle ici que ces opérations n'ont pas une sémantique « court-circuit » : elles évaluent *toujours* leurs deux opérandes. Il existe une instruction booléenne unaire : *non,* qui correspondent à l'opérateur *not*.

Les constantes booléennes *vrai* et *faux* sont respectivement codées par les entiers 1 et 0.

### 5.5.1 egal (15)

Cette instruction compare les deux valeurs *op1* et *op2* en sommet de pile et empile le code de l'expression booléenne *op1=op2*.

**procedure** egal() ;




### 5.5.2 diff (16), inf (17), infeg(18), sup (19), supeg (20)

Ces instructions sont similaires à l'instruction *egal* pour les opérations / =, <, <=, >, >= . Leur sémantique pré/post n'est pas présentée ici.

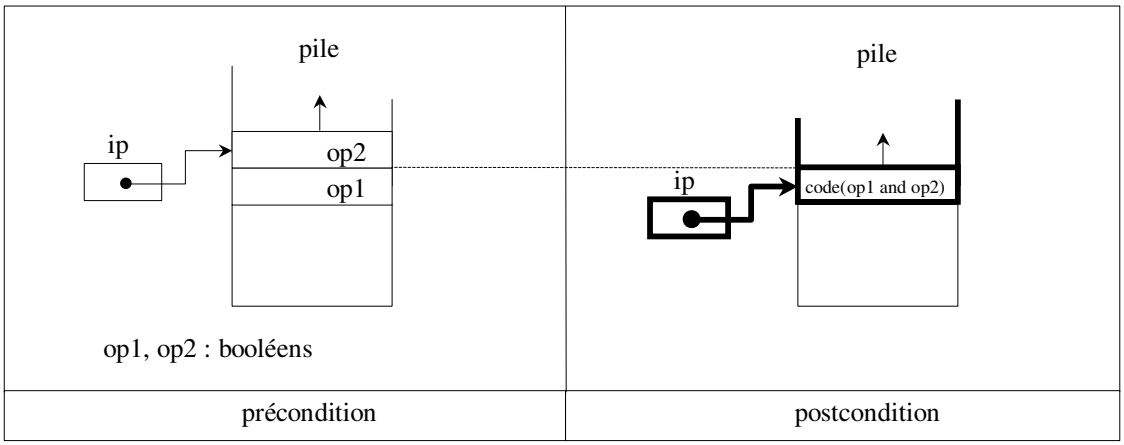
**procedure** diff() ;
**procedure** inf() ;
**procedure** infeg() ;
**procedure** sup() ;

**procedure** supeg() ;

### 5.5.3 et (21)

Cette instruction prend en compte deux booléens *op1* et *op2* en sommet de pile et place dans la pile le code de l'expression booléenne (*op1* **and** *op2*).

**procedure** et() ;




### 5.5.4 ou (22)

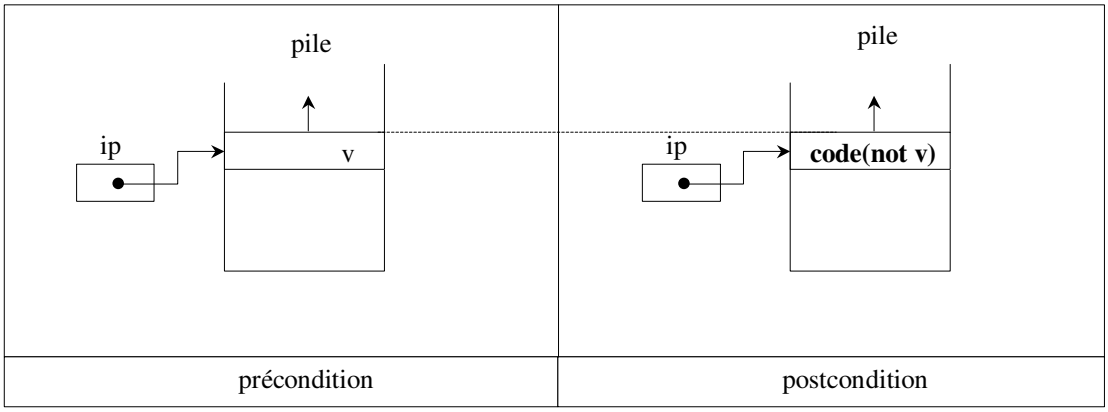
Cette instruction prend en compte deux booléens *op1* et *op2* en sommet de pile, et place dans la pile le code de l'expression booléenne (*op1* **or** *op2*). Cette instruction est similaire à l'instruction *et*. Sa sémantique pré/post n'est pas présentée ici.

**procedure** ou() ;

### 5.5.5 non (23)

Cette instruction permet de calculer la négation du booléen en sommet de pile.

**procedure** non() ;



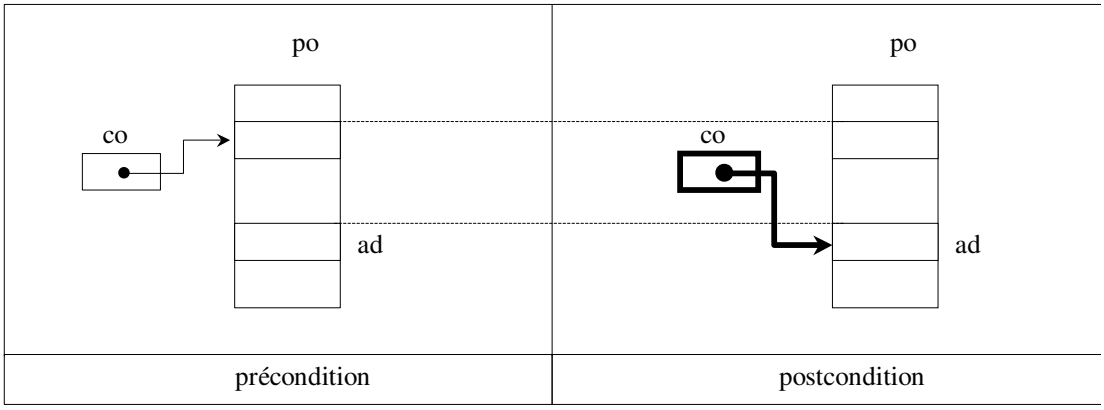

## 5.6 Contrôle

Nous étudions ici trois instructions de contrôle : *tra, tze, erreur.* Ces opérations agissent sur la progression du compteur ordinal.

### 5.6.1 tra (24)

**procedure** tra(ad : integer) ;

Cette instruction donne le contrôle à l'instruction située à l'adresse *ad.* Il s'agit d'un branchement inconditionnel.

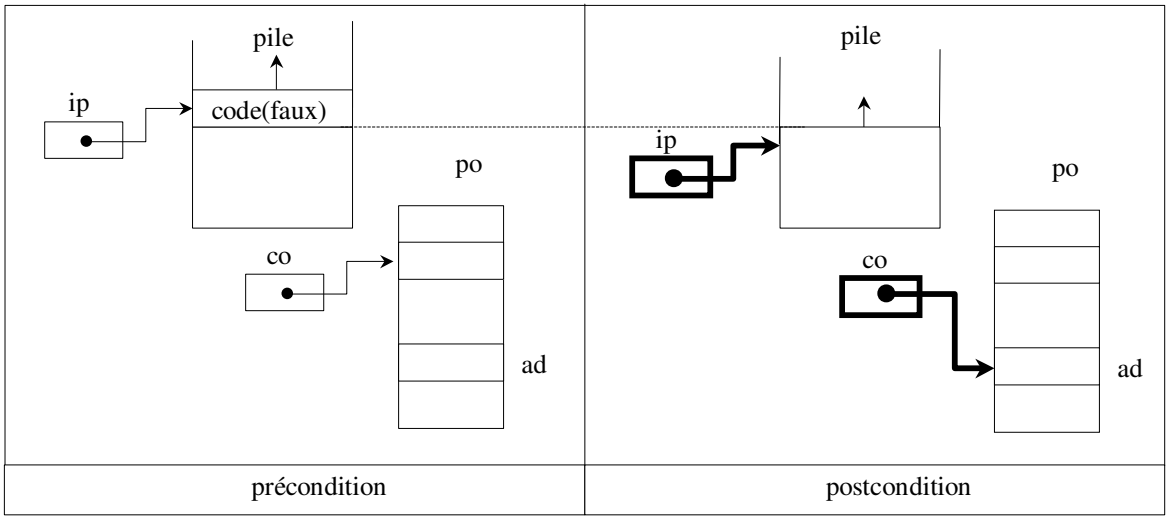



### 5.6.2 tze (25)

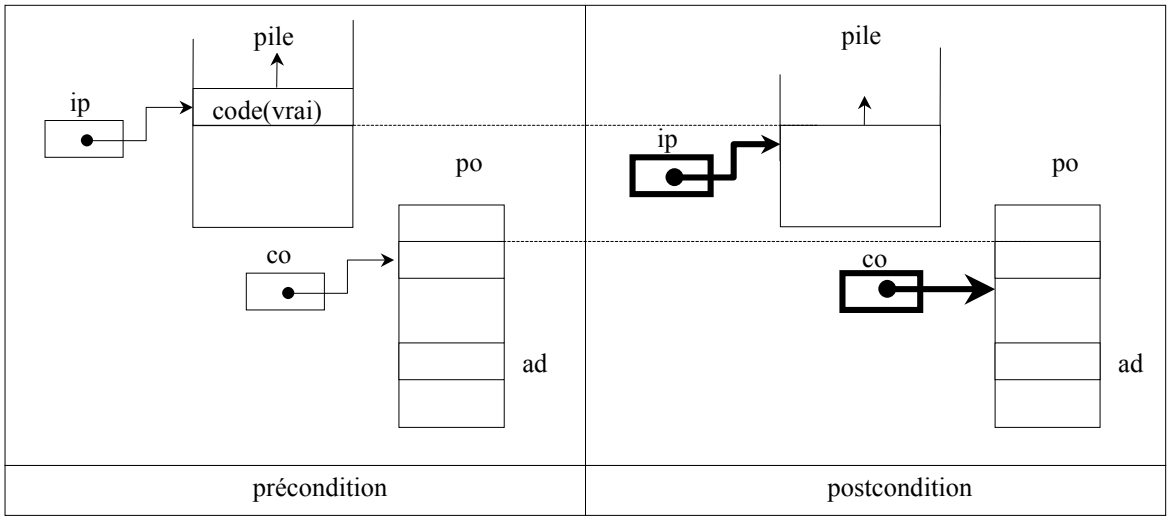
**procedure** tze(ad : integer) ;

Cette instruction donne le contrôle à l'instruction située à l'adresse *ad* si le sommet de pile contient *faux,* continue en séquence sinon.

**Premier cas :** 0 (*faux*) en sommet de pile.




**Second cas :** 1 (*vrai*) en sommet de pile.




### 5.6.3 erreur (26)

**procedure** erreur() ;

Cette instruction machine est compilée à la rencontre de l'instruction nilNovi Algorithmique *error(exp)*. Elle affiche la valeur de l'expression *exp* passée en paramètre et arrête la machine nilNovi Algorithmique.

# 6 Schéma de compilation et d'exécution

## 6.1 Introduction

Nous présentons ici la compilation des différentes constructions d'un programme NIL-NOVI ALGORITHMIQUE. La compilation y est vue comme un processus inductif : on distingue donc les cas de base (déclaration de variables, expressions réduites à une constante, une variable, un paramètre, un attribut, ordre de lecture) et les cas inductifs, pour lesquels la compilation d'une structure $A$ constituée des sous-structures $X$ et $Y$ est composée de la compilation de $X$ et de $Y$, ce que l'on formalisera de la manière suivante, en notant $\{P\}$ la compilation d'un programme source $P$ :

$$\{A\} = \Phi(\{X\}, \{Y\})$$

Dans la suite $ad(< V >)$ signifie « adresse de l'identificateur $< V >$ ».

## 6.2 Unité principale

$$\left\{\begin{array}{l}\textbf{procedure}\ \text{p}\ \ \textbf{is}\\ \quad < \text{A} >\\ \textbf{begin}\\ \quad < \text{B} >\\ \textbf{end}.\end{array}\right\} = \begin{array}{l}\texttt{debutProg}();\\ \{< \texttt{A} >\};\\ \{< \texttt{B} >\};\\ \texttt{finProg}()\end{array}$$

## 6.3 Ordre d'écriture

$$\{\text{put}(< \texttt{E} >)\} = \begin{array}{l}\{< \texttt{E} >\};\\ \texttt{put}()\end{array}$$

On rappelle que le paramètre $< \texttt{E} >$ est une expression entière.

## 6.4 Variables

Nous considérons ici que l'on sait compiler des expressions. Cette lacune sera comblée dans la section suivante.

### 6.4.1 Déclarations

On considère la déclaration successive de $n, (n > 0)$ variables.

$$\{\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}\ :< \texttt{type} >\} = \texttt{reserver(n)}$$

### 6.4.2 Affectation

Soit $< E >$ une expression.

$$\{< \texttt{V} >:=< \texttt{E} >\} = \begin{array}{l}\texttt{empiler}(\texttt{ad}(< \texttt{V} >));\\ \{< \texttt{E} >\};\\ \texttt{affectation}()\end{array}$$

## 6.5 Expressions

### 6.5.1 Expressions arithmétiques

Quatre cas de base sont développés. Par induction structurelle, et en prenant en considération la priorité des opérateurs et le parenthèsage, il sera possible d'appliquer la démarche à toute expression arithmétique.

#### Expression réduite à une constante entière

$$\{< \texttt{C} >\} = \texttt{empiler}(< \texttt{C} >)$$

#### Expression avec un opérateur binaire

$$\{< \texttt{E1} > \textbf{op} < \texttt{E2} >\} = \begin{array}{l}\{< \texttt{E1} >\};\\ \{< \texttt{E2} >\};\\ \texttt{op}()\end{array}$$

où *op* représente l'un des opérateurs arithmétiques $+$, $-$, $*$, $/$ et *op()* est l'instruction correspondante (*add(), sous(), mult(), div()*) de la machine nilNovi Algorithmique.

#### Expression avec l'opérateur moins unaire

$$\{- < \texttt{V} >\} = \begin{array}{l}\{< \texttt{V} >\};\\ \texttt{moins}()\end{array}$$

#### Expressions réduites à une variable

Les expressions réduites à une variable se compilent de la manière suivante :

$$\{< \texttt{V} >\} = \begin{array}{l}\texttt{empiler}(\texttt{ad}(< \texttt{V} >));\\ \texttt{valeurPile}()\end{array}$$

### 6.5.2 Expressions booléennes

Trois cas sont considérés : les constantes, les opérateurs binaires et l'opérateur **not**. Ensuite, à l'instar des expressions arithmétiques, par induction structurelle, et en prenant en considération la priorité des opérateurs et le parenthèsage, il sera possible d'appliquer la démarche à toute expression booléenne.

#### Expression réduite à une constante booléenne

On rappelle que *vrai* se code 1 et *faux* 0.

$$\boxed{\{\texttt{true}\} = \texttt{empiler}(1)}$$

$$\boxed{\{\texttt{false}\} = \texttt{empiler}(0)}$$

#### Expression avec un opérateur booléen binaire

La démarche s'applique tant aux expressions booléennes pures qu'aux expressions relationnelles.

$$\boxed{\begin{array}{rl} & \{< \texttt{E1} >\}; \\ \{< \texttt{E1} > \textbf{ op } < \texttt{E2} >\} = & \{< \texttt{E2} >\}; \\ & \texttt{op}() \end{array}}$$

où *op* représente l'un des opérateurs booléens **and, or** ou relationnel =, / =, <, <=, >, >= et *op()* est l'instruction correspondante (*et(), ou(), egal(), diff(), inf(), infeg(), sup(), supeg()*) de la machine nilNovi Algorithmique.

#### Expression avec un opérateur not

$$\boxed{\{\textbf{not } < \texttt{E} >\} = \begin{array}{l} \{< \texttt{E} >\}; \\ \texttt{non}() \end{array}}$$

## 6.6 Séquentialité

La composition séquentielle se traduit trivialement de la manière suivante :

$$\boxed{\{< \texttt{S1} >; < \texttt{S2} >\} \;=\; \{< \texttt{S1} >\}; \{< \texttt{S2} >\}}$$

## 6.7 Ordre de lecture

L'ordre de lecture *get* permet de lire une variable entière depuis le clavier.

$$\boxed{\{\text{get}(< \texttt{V} >)\} = \begin{array}{l} \texttt{empiler}(\texttt{ad}(< \texttt{V} >)); \\ \texttt{get}() \end{array}}$$

## 6.8 Alternatives

Deux cas d'alternatives sont à considérer : les alternatives simples et les alternatives doubles.

### Alternatives simples

$$\left\{\begin{array}{l}\textbf{if} < \texttt{C} > \ \textbf{then} \\ \quad < \texttt{A} > \\ \textbf{end}\end{array}\right\} = \begin{array}{l}\{< \texttt{C} >\}; \\ \texttt{tze(ad)}; \\ \{< \texttt{A} >\}; \\ \texttt{ad} : \ldots\end{array}$$

### Alternatives doubles

$$\left\{\begin{array}{l}\textbf{if} < \texttt{C} > \ \textbf{then} \\ \quad < \texttt{A} > \\ \textbf{else} \\ \quad < \texttt{B} > \\ \textbf{end}\end{array}\right\} = \begin{array}{l}\{< \texttt{C} >\}; \\ \texttt{tze(ad1)}; \\ \{< \texttt{A} >\}; \\ \texttt{tra(ad2)}; \\ \texttt{ad1} : \{< \texttt{B} >\}; \\ \texttt{ad2} : \ldots\end{array}$$

## 6.9 Boucle

$$\left\{\begin{array}{l}\textbf{while} < \texttt{C} > \ \textbf{loop} \\ \quad < \texttt{A} > \\ \textbf{end}\end{array}\right\} = \begin{array}{l}\texttt{ad1} : \{< \texttt{C} >\}; \\ \texttt{tze(ad2)}; \\ \{< \texttt{A} >\}; \\ \texttt{tra(ad1)}; \\ \texttt{ad2} : \ldots\end{array}$$

# 7 Exemple

On considère un exemple simple. On note en commentaires les adresses associées aux variables et on présente le code objet correspondant.

```
01 : procedure p is
02 :    v, som : integer ;                          // v : 0; som : 1
03 : begin
04 :    get(v) ;
05 :    som :=0 ;
06 :    while v/=0 loop
07 :       som:=som+v;
08 :       get(v)
```

```
09 :     end ;
10 :     put(som)
11 : end.
```

Le code objet correspondant à la compilation de ce programme se présente de la manière suivante (le numéro de la ligne source est rappelé à droite de la ligne objet) :

```
00 :   debutProg()                    01
01 :   reserver(2)                    02
02 :   empiler(0)                     04
03 :   get()                          04
04 :   empiler(1)                     05
05 :   empiler(0)                     05
06 :   affectation()                  05
07 :   empiler(0)                     06
08 :   valeurPile()                   06
09 :   empiler(0)                     06
10 :   diff()                         06
11 :   tze(22)                        06
12 :   empiler(1)                     07
13 :   empiler(1)                     07
14 :   valeurPile()                   07
15 :   empiler(0)                     07
16 :   valeurPile()                   07
17 :   add()                          07
18 :   affectation()                  07
19 :   empiler(0)                     08
20 :   get()                          08
21 :   tra(08)                        09
22 :   empiler(1)                     10
23 :   valeurPile()                   10
24 :   put()                          10
25 :   finProg()                      11
```

# Deuxième partie
# nilNovi Procédural

nilNovi Procédural est un langage de programmation rudimentaire destiné à permettre l'initiation aux techniques élémentaires de la compilation. C'est un langage fortement typé orienté vers la compilation. On trouvera à l'annexe B sa grammaire sous forme BNF. Cette partie du document présente le langage, sa compilation et son exécution.

## 8 Introduction : nilNovi Procédural par l'exemple

Le langage nilNovi Procédural est un sur-ensemble du langage nilNovi Algorithmique permettant en particulier de déclarer et d'exécuter des opérations (procédures et fonctions). Un programme nilNovi Procédural se compose d'un nom permettant son identification (située entre les mots-clés *procedure* et *is*) d'une partie déclarative (située entre les mots-clés *is* et *begin*) et d'une partie impérative (située entre les mots-clés *begin* et *end*). La partie déclarative comprend la déclaration des opérations suivie de la déclaration des variables [2]. Les seuls types autorisés sont les entiers (*integer*) et les booléens (*boolean*). Une opération est elle-même composée d'une partie déclarative et d'une partie impérative. La partie déclarative d'une opération est réservée aux variables locales, elle ne peut contenir de déclaration d'opérations.

### 8.1 Exemple 1

```
01 : procedure pp is
02 :     function fact(i: integer) return integer is
03 :         // pré : i>=0
04 :         // post : résultat : i!
05 :     begin
06 :         if i=0 then
07 :             return 1
08 :         else
09 :             return i*fact(i-1)
10 :         end
11 :     end;
12 :     procedure p(i: integer) is
13 :         // pré : i>=0
14 :         // post : fact(0) à fact(i-1) sont affichés
15 :         j: integer;
```

2. Il n'existe que des variables scalaires.

```
16 :     begin
17 :        j:=0;
18 :        while j/ =i loop
19 :           // fact(0) à fact(j-1) sont affichés
20 :           put(fact(j));
21 :           j:=j+1
22 :        end
23 :     end;
24 :     k: integer;
25 : begin
26 :     get(k) ;
27 :     if k>=0 then
28 :        p(k)
29 :     end
30 : end.
```

**Commentaires**

1. Le programme *pp* permet de lire une valeur entière $k$ et, si elle est positive ou nulle d'afficher la valeur des factorielles depuis 0 jusqu'à $k-1$. Ce programme est organisé autour d'une fonction *fact* (lignes 02 à 11) qui calcule la factorielle du paramètre et d'une procédure $p$ (lignes 12 à 23) qui effectue l'affichage. Le bloc impératif (lignes 26 à 29) effectue la lecture de la variable $k$ et invoque la procédure $p$ si la précondition de $p$ est vraie.
2. En nilNovi Procédural toute utilisation d'un identificateur doit être précédée de sa déclaration. On note que la fonction *fact* est déclarée avant la procédure $p$, il est donc possible d'utiliser *fact* dans $p$. C'est ce qui est fait (ligne 20). Ce principe interdit l'utilisation croisée d'opérations (et donc la récursivité croisée). Cette limitation est rédhibitoire pour un usage industriel mais n'a pas de conséquences sur notre objectif pédagogique. Voir le § 9 pour un développement de cet aspect.
3. L'appel d'une procédure constitue une instruction (au même titre qu'une affectation par exemple). La ligne 28 en est un exemple. L'appel d'une fonction possède le statut d'une expression (ligne 20).
4. On note dans la déclaration de la fonction *fact*, l'usage de l'instruction spécifique *return* (lignes 07 et 09). Cette instruction possède un double rôle : elle signale d'une part la valeur qui est délivrée par l'appel de la fonction et elle termine d'autre part l'exécution de la fonction. Cette instruction est réservée aux fonctions : elle est donc interdite dans les procédures ainsi que dans le bloc impératif du programme principal mais il doit exister au moins au moins une occurrence de cette instruction dans le bloc impératif d'une fonction.
5. Le mot clé *return* est par ailleurs utilisé dans la déclaration de la fonction (ligne 02) pour spécifier le type de la valeur délivrée.

6. Aux lignes 06 et 18 on utilise des opérateurs relationnels (resp. = et /=). De même que dans nilNovi Algorithmique, ces deux opérateurs sont disponibles pour tous les types. Par contre les autres opérateurs relationnels (<, <=, >, >=) ne sont disponibles que pour le type *integer*.

**Ce qu'il faut retenir de cet exemple**

1. nilNovi Procédural est un langage qui permet de définir et d'exécuter des procédures et des fonctions. C'est un sur-ensemble du langage nilNovi Algorithmique.
2. Les identificateurs (variables, opérations) doivent être déclarés avant d'être utilisés.
3. Les seuls types existants sont les entiers (*integer*) et les booléens (*boolean*).
4. Les opérations peuvent déclarer des variables locales mais ne peuvent déclarer d'autres opérations.
5. nilNovi Procédural permet de déclarer des opérations récursives.
6. Compte tenu des règles concernant la portée des identificateurs, il est impossible qu'une opération accède à une variable globale. Elle ne peut mentionner que les paramètres et les variables locales. La notion de portée est développée plus loin.

## 8.2 Exemple 2

Cet exemple nous permet de nous intéresser aux modes de passage de paramètres.

```
01 : procedure pp is
02 :    procedure p(v: in integer; s: in out integer) is
03 :       // pré : s=S
04 :       // post : s=S+v
05 :    begin
06 :       s:=s+v
07 :    end;
08 :    procedure q(w: in out integer) is
09 :    begin
10 :       p(w+1,w)
11 :    end;
12 : begin
13 :    i:=12;
14 :    q(i,i);
15 :    put(i)
16 : end.
```

**Commentaires**

1. La procédure $p$ (lignes 02 à 07) utilise deux paramètres formels : $v$, de mode *in*, et $s$, de mode *in out*. Les paramètres formels servent à décrire le calcul. Ils seront associés, lors du calcul effectif, à des paramètres effectifs. Un paramètre formel de mode *in* possède le statut d'une valeur (il est en particulier interdit d'utiliser un identificateur de paramètre de mode *in* à gauche d'un symbole d'affectation). Les paramètres effectifs correspondants sont des expressions du même type. Un paramètre formel de mode *in out* possède le statut de variable. Les paramètres effectifs correspondants sont des variables ou des paramètres de mode *in out* du même type. Notons qu'une fonction ne peut avoir de paramètres de mode *in out*[3] et que par défaut les paramètres sont de mode *in*.
2. La procédure $q$, lignes 08 à 11, effectue un appel à la procédure $p$. Elle utilise pour cela son propre paramètre (de mode *in out*) à deux occasions, une première fois dans l'expression qui tient lieu de premier paramètre effectif pour la procédure $p$, et une seconde fois en tant que second paramètre effectif de la procédure $p$.

# 9 Rattachement d'un identificateur à une entité

## 9.1 Introduction

Dans un programme nilNovi Procédural il est possible d'utiliser un même identificateur pour désigner différentes entités. Cependant un usage particulier d'un identificateur ne doit pas être ambigu. Lever les ambiguïtés possibles se fait soit en exploitant le contexte (voir l'exemple ci-dessous) soit par des conventions sur le lien entre l'identificateur et l'une des entités possibles (voir le second exemple). Dans ce dernier cas on fait usage des notions de portée et de visibilité.

**Exemples :**

```
01 : procedure pp is
02 :    function t() return integer is
03 :    begin
04 :       return 5
05 :    end;
06 :    t: integer ;
07 : begin
08 :    t:=6 ;
09 :    print(t) ;
10 :    print(t())
11 : end.
```

3. Une fonction est destinée à *observer* l'état d'un programme, pas à le modifier.

La ligne 06 déclare l'identificateur $t$ comme étant un entier. La ligne 02 déclare le même identificateur comme étant une fonction sans paramètre délivrant un entier. À la ligne 08 l'identificateur $t$ est utilisé. Le contexte permet de déterminer que l'on a affaire à la *variable* $t$. Pour la même raison on sait que la ligne 09 (resp. 10) utilise la *variable* (resp. la *fonction*) $t$. En effet à la ligne 10 la présence des parenthèses caractérise un appel d'opération.

```
01 : procedure pp is
02 :    procedure p() is
03 :       t: integer;
04 :    begin
05 :       t:= 15
06 :    end;
07 :    t: integer ;
08 : begin
09 :    t:=5 ;
10 :    p()
11 : end.
```

L'identificateur $t$ est utilisé dans deux déclarations : à la ligne 03 pour déclarer une variable *locale* entière et à la ligne 07 pour déclarer une variable *globale* entière. La ligne 09 utilise cet identificateur. L'entité désignée est-elle la variable locale ou la variable globale ? Le contexte ne permet pas de décider. Dans ce cas se sont les notions de portée et de visibilité (voir ci-dessous) qui vont permettre d'affirmer que l'affectation porte sur la variable *globale.*

Définir les notions de portée et de visibilité va permettre de répondre aux questions suivantes : étant donné une déclaration d'entité (variable globale, variable locale), dans quel fragment du texte du programme a-t-on le droit d'utiliser cet identificateur pour désigner l'entité en question (notion de portée) ? Cet usage peut-il être momentanément invalidé par une autre déclaration du même identificateur pour désigner une entité différente (notion de visibilité) ?

## 9.2 Cas de l'identificateur de la procédure principale

**Portée.** Compte tenu qu'il n'est jamais nécessaire de désigner la procédure principale, on considère que la portée de cet identificateur est vide.

## 9.3 Cas d'un identificateur d'opération

**Portée.** La portée d'un identificateur d'opération est le texte du programme à partir de l'occurrence de la déclaration d'opération.

**Visibilité.** La visibilité d'un identificateur d'opération est identique à sa portée.

**Conflit.** Un conflit est possible avec un autre identificateur d'opération. Dans ce cas le programme est erroné.

La portée de l'identificateur du programme principal est vide.

**Exemple :**

```
01 : procedure p is
02 :    procedure p() is
03 :    begin
04 :       put(56)
05 :    end;
06 :    function p() return boolean is
07 :    begin
08 :       return false
09 :    end;
10 : begin
11 :    p()
12 : end.
```

La déclaration de procédure de la ligne 02 est correcte. Cette procédure peut être appelée dans la zone qui va de la ligne 02 à la ligne 12. Par contre la déclaration de fonction de la ligne 06 est incorrecte : elle est en conflit avec celle de la ligne 02.

## 9.4 Cas d'un identificateur de variable ou de paramètre

**Portée.** La portée d'un identificateur de variable ou de paramètre est le texte de l'opération où l'identificateur est déclaré, à partir de l'occurrence de la déclaration.

**Visibilité.** La visibilité d'un identificateur d'opération est identique à sa portée.

**Conflit.** Un conflit est possible avec un autre identificateur de variable ou de paramètre dans la même opération. Dans ce cas le programme est erroné.

**Exemple :**

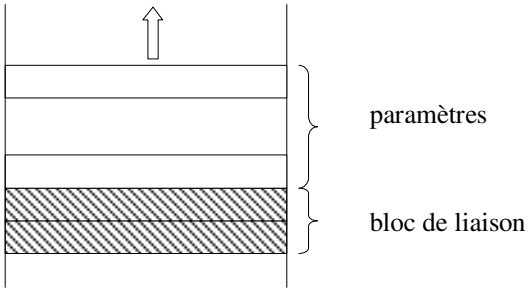
```
01 : procedure pp is
02 :    procedure p(i : in out integer) is
03 :       i: boolean;
04 :       j: integer;
05 :    begin
06 :       ...
07 :    end;
08 :    function f(i: integer) return boolean is
```

```
09 :        j : integer;
10 :        j : boolean;
11 :    begin
12 :        ...
13 :    end;
14 :    i, j : integer;
15 : begin
16 :    p()
17 : end.
```

La déclaration de la variable locale $i$ de la ligne 03 est incorrecte : elle est en conflit avec la déclaration de paramètre $i$ de la ligne 02. La déclaration de la variable locale $j$ (ligne 10) est incorrecte : elle est en conflit avec la déclaration de la ligne 09. Les déclarations de la ligne 14 sont correctes.

# 10 Attribution d'adresses statiques

En préambule à cette section il faut savoir (cela sera détaillé dans les sections postérieures) que dans un langage tel que nilNovi Procédural, les entités (variables et paramètres) sont gérées dans une pile (la pile d'exécution). Notons également que lors de l'exécution d'une opération, un bloc de données (le bloc de liaison) est créé, il permet d'effectuer correctement le retour. Les paramètres effectifs sont placés au-dessus du bloc de liaison.




Différentes raisons obligent, ou incitent, dans un langage tel que nilNovi Procédural, à gérer dynamiquement la mémoire. Citons :

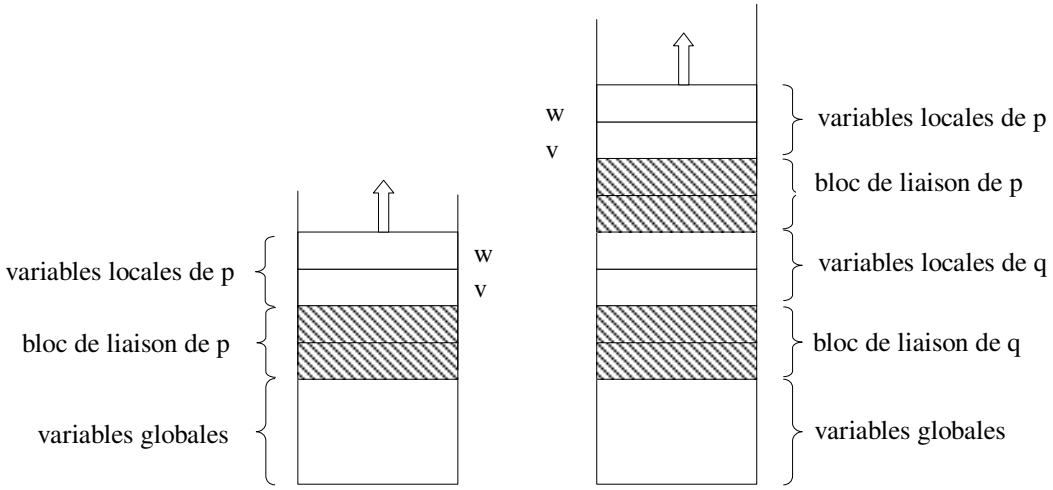
— une meilleure gestion de la place libre (seules les entités utiles existent au moment souhaité),
— la récursivité qui, du point de vue des variables locales, va conduire à associer au cours du temps *différents* emplacements à une *même* entité.

Une difficulté surgit alors : à l'exception des variables globales, le compilateur n'est pas capable de calculer l'adresse définitive d'une variable. Pour lever cette difficulté on utilise une adresse intermédiaire entre l'adresse symbolique et l'adresse définitive (l'adresse

« dynamique ») : c'est ce que nous appelons l'adresse *statique*. Cette adresse statique se présente comme un déplacement par rapport à une base.

**Exemple.** Dans l'exemple qui suit nous allons montrer que des variables locales (celles de la procédure $p$) peuvent se voir attribuer des adresses dynamiques différentes au cours de l'exécution.

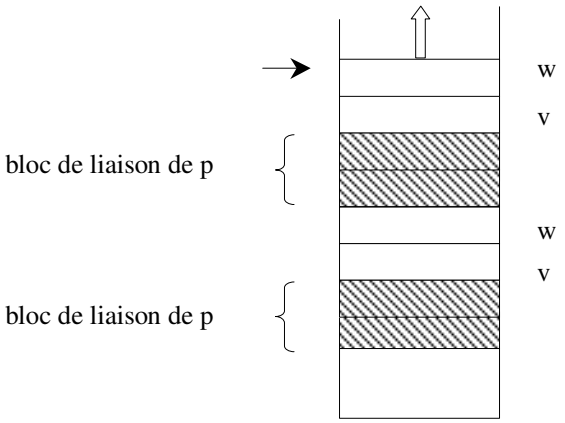
```
01 : procedure pp is
02 :     procedure p() is
03 :         v, w: integer;
04 :     begin
05 :         ···
06 :     end;
07 :     procedure q() is
08 :         i, j: integer;
09 :     begin
10 :         ···
11 :         p()
12 :     end;
13 : begin
14 :     ···
15 :     p() ;
16 :     q()
17 : end.
```




Appel de la ligne 15 Appel de la ligne 16

On note effectivement que d'un appel à l'autre l'adresse d'implantation des variables locales $v$ et $w$ est différente.

Le cas de figure suivant est encore plus caractéristique. Il fait l'hypothèse que la procédure $p$ est récursive. Plusieurs instances des variables locales $v$ et $w$ coexistent, seule celle qui est en sommet de pile est accessible.




Dans la suite de la section nous montrons comment s'effectue l'attribution des adresses statiques des variables et des paramètres.

**Adresses statiques des variables locales et des paramètres des opérations.** Pour une opération donnée (y compris la procédure principale) elles sont attribuées séquentiellement à partir de 0.

**Exemple**

```
01 : procedure p is
02 :    procedure p(k,l : in integer) is                  // k:0;l:1
03 :       i, s: integer;                                 //i:2;s:3
04 :    begin
05 :       i:= k;                                         //i:2;k:0
06 :       s:=l+3;                                        //s:3;l:1
07 :       ...
08 :    end;
09 :    function f(a,b : integer) return integer is       //a:0;b:1
10 :       t, l: integer;                                 //t:2;l:3
11 :    begin
12 :       ...
13 :    end;
14 :    i,j,s: integer;                                   //i:0;j:1;s:2
15 : begin
```

```
16 :    p(f(12,23),15)
17 : end.
```

**Commentaire.** Pour chaque opération, les adresses statiques des paramètres et des variables locales sont attribuées dans l'ordre d'apparition à partir de 0.

# 11 La machine nilNovi Procédural

Nous allons dans cette section étudier le « matériel » de la machine nilNovi Procédural. Le langage machine sera décrit dans une section spécifique en utilisant une sémantique pré/post.

## 11.1 Le matériel de la machine nilNovi Procédural

La machine nilNovi Procédural est constituée de deux parties que nous allons présenter successivement : la partie « données » et la partie « programme ».

La partie « données » se compose d'une pile (la pile d'exécution) et de deux registres. La pile d'exécution est destinée à mémoriser les variables ainsi que les informations nécessaires à l'exécution d'une opération. Elle permet également d'évaluer les expressions. Elle est gérée par deux registres : *ip* et *base*. *ip* désigne le sommet de pile tandis que *base* désigne le premier élément du bloc de liaison [4] de l'opération en cours d'exécution.

La partie « programme » est destinée à recevoir le programme objet à des fins d'exécution. Elle est composée d'un tableau *po* constituant la mémoire de programme et d'un registre *co* : le compteur ordinal. L'architecture est présentée à la figure 2.

# 12 Jeu d'instructions de la machine nilNovi Procédural

Afin de ne pas alourdir la description, aucun contrôle (débordement de pile, division par 0, etc.) n'est spécifié.

Le nombre noté entre parenthèses (par exemple (1) après *debutProg*) désigne le code machine de l'instruction. Dans la suite ni l'initialisation ni l'évolution du compteur ordinal ne sont pas prises en compte excepté dans les instructions de contrôle.

La suite de cette section spécifie les instructions de la machine nilNovi Procédural sous la forme pré/post. Les instructions sont classées par famille (entrée/sortie, variables et affectation, etc.) bien que certaines d'entre elles puissent appartenir à plusieurs familles.

4. Comme nous l'avons vu ci-dessus, un bloc de liaison est constitué de l'ensemble des informations nécessaires au retour d'une opération. En nilNovi Procédural il s'agit de deux informations élémentaires : l'adresse de retour et l'adresse du bloc de liaison appelant.

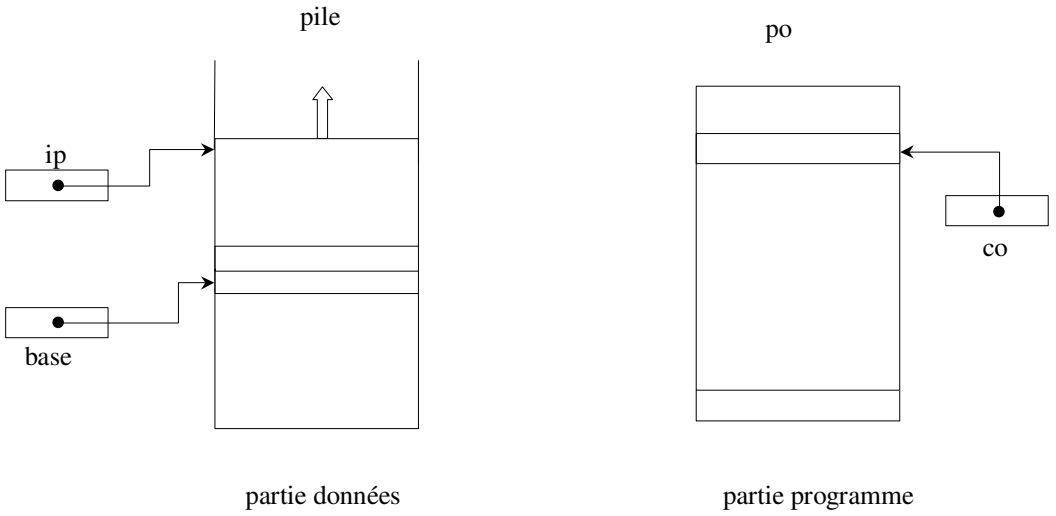
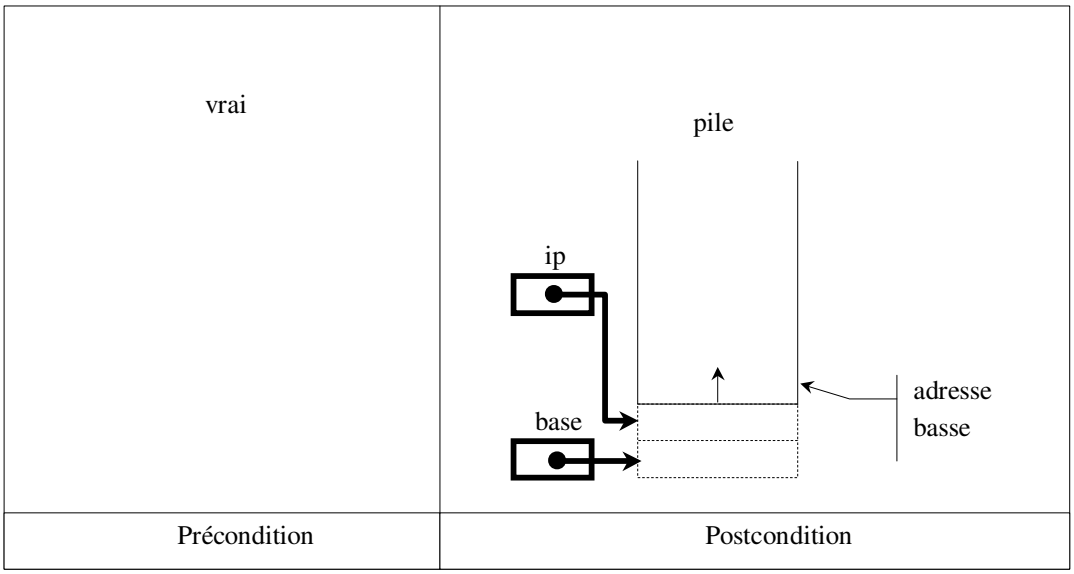



FIGURE 2 – Structure de la machine NILNOVI PROCÉDURAL

## 12.1 Unité de compilation

### 12.1.1 debutProg (1)

**procedure** debutProg() ;




### 12.1.2 finProg (2)

**procedure** finProg() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

## 12.2 Variables et affectation

Nous avons vu ci-dessus que le compilateur ne peut attribuer une adresse absolue (adresse « dynamique ») à une variable locale et qu'il est nécessaire de passer par une adresse intermédiaire : l'adresse statique. Celle-ci représente à l'exécution un déplacement par rapport à l'information contenue dans le registre *base*. Ces deux informations (valeur du registre *base* et adresse statique) permettent de calculer l'adresse dynamique à chaque accès à la variable.

### 12.2.1 reserver(3)

**procedure** reserver(n : integer) ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.2.2 empiler (4)

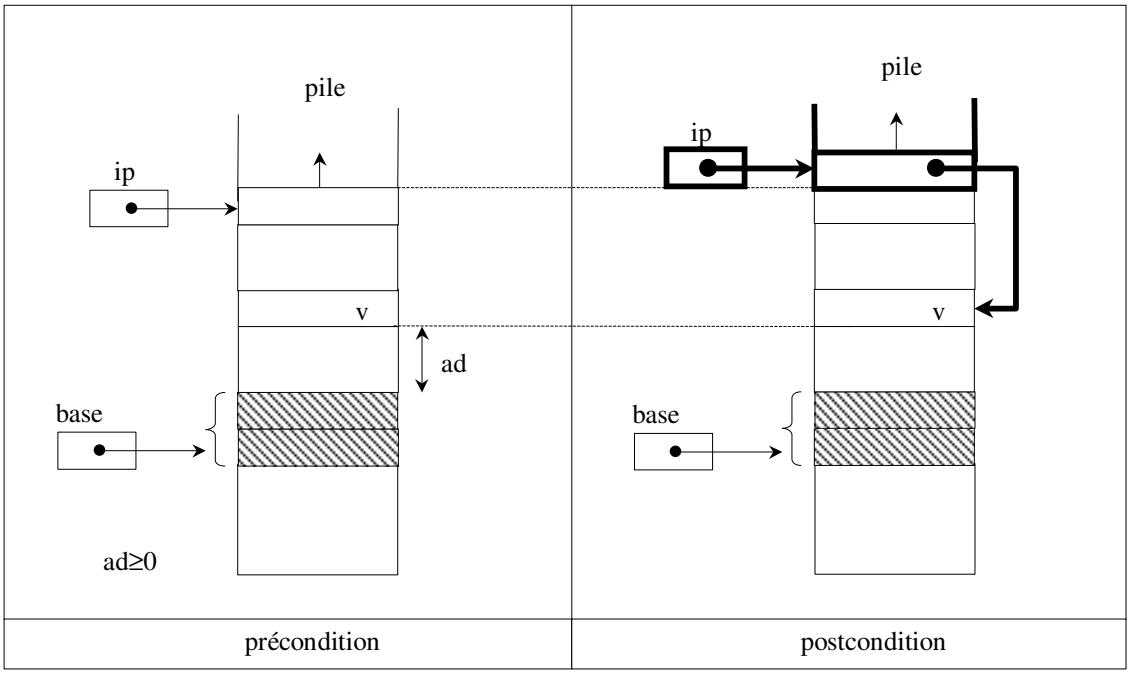
**procedure** empiler(val : integer) ;

Cette instruction est utilisée pour empiler les adresses des variables globales ainsi que les valeurs présentes dans une expression. Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.2.3 empilerAd (5)

**procedure** empilerAd(ad : integer) ;

Cette instruction est utilisée dans le cas de variables locales pour transformer l'adresse statique en adresse dynamique. La valeur *ad* désigne le $ad^e$ emplacement au-dessus du bloc de liaison courant.




### 12.2.4 affectation (6)

**procedure** affectation() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.2.5 valeurPile (7)

**procedure** valeurPile() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

## 12.3 Entrées-Sorties

Il existe deux instructions dans cette famille : une pour les entrées (clavier) et une pour les affichages écran.

### 12.3.1 get (8)

**procedure** get() ;

L'ordre de lecture *get* permet de lire une variable ou un paramètre d'entrée-sortie depuis le clavier. Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.3.2 put (9)

**procedure** put() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

## 12.4 Expressions arithmétiques

Il existe une instruction unaire, *moins,* et quatre instructions binaires : *mult, add, sous* et *div.* Les instructions binaires opèrent sur les deux éléments en sommet de pile.

### 12.4.1 moins (10)

**procedure** moins() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.4.2 sous (11)

**procedure** sous() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.4.3 add (12), mult (13), div(14)

**procedure** add() ;
**procedure** mult() ;
**procedure** div() ;

Voir les mêmes instructions dans la machine nilNovi Algorithmique.

## 12.5 Expressions relationnelles et booléennes

Les constantes booléennes *true* et *false* sont respectivement codées par les entiers 1 et 0.

### 12.5.1 egal (15)

**procedure** egal() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.5.2 diff (16), inf (17), infeg(18), sup (19), supeg (20)

**procedure** diff() ;
**procedure** inf() ;
**procedure** infeg() ;
**procedure** sup() ;
**procedure** supeg() ;

Voir les mêmes instructions dans la machine nilNovi Algorithmique.

### 12.5.3 et (21)

**procedure** et() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.5.4 ou (22)

**procedure** ou() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.5.5 non (23)

**procedure** non() ;

Voir la même instruction dans la machine nilNovi Algorithmique.

## 12.6 Contrôle

### 12.6.1 tra (24)

**procedure** tra(ad : integer) ;

Voir la même instruction dans la machine nilNovi Algorithmique.

### 12.6.2 tze (25)

**procedure** tze(ad : integer) ;

Voir la même instruction dans la machine nilNovi Algorithmique.

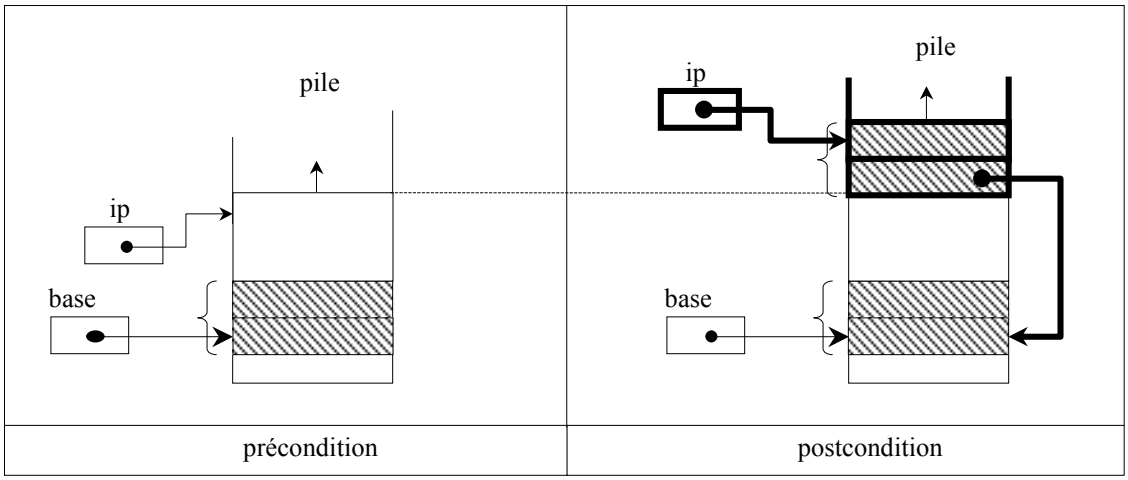
## 12.7 Opérations

Cette famille regroupe les instructions spécifiques à la compilation des procédures et des fonctions.

### 12.7.1 reserverBloc (30)

**procedure** reserverBloc() ;

Cette instruction permet, lors de l'appel d'une opération, de réserver les emplacements du futur bloc de liaison et d'initialiser la partie « pointeur vers le bloc de liaison de l'appelant ».



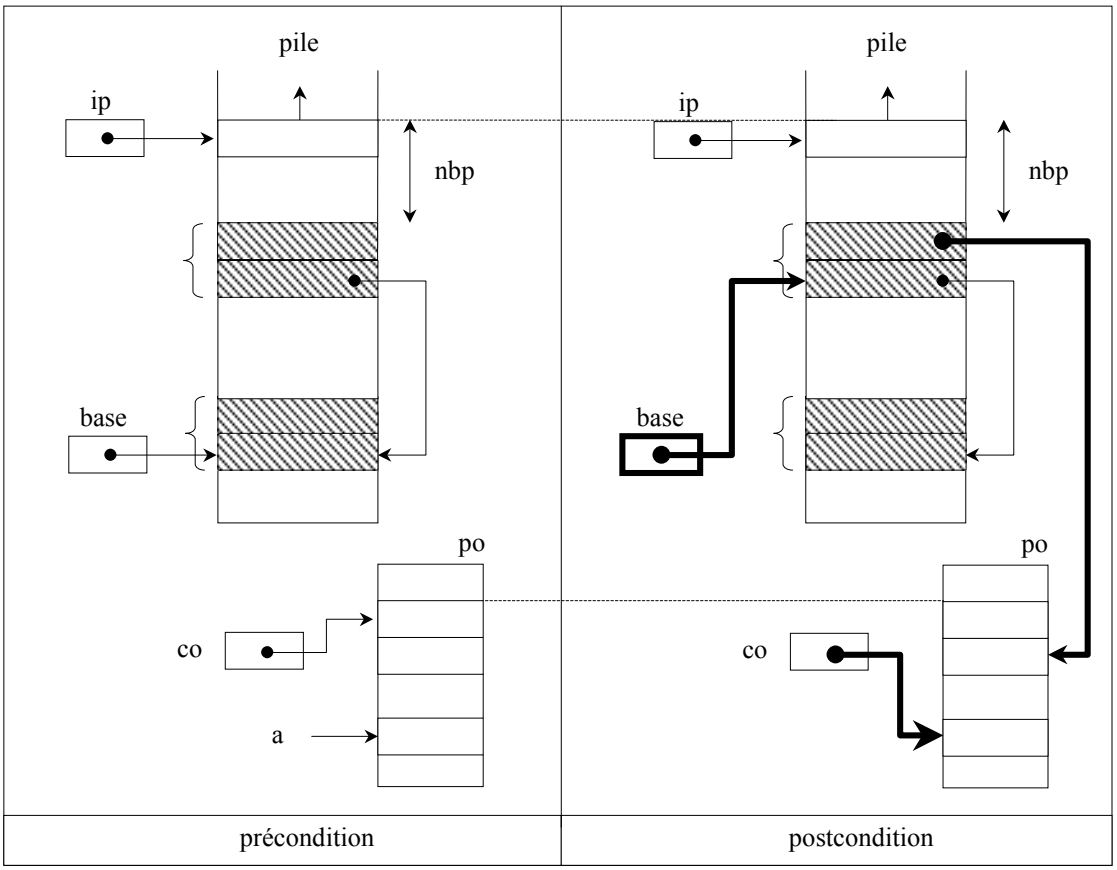

On note que l'opération ne provoque pas de changement de bloc de liaison [5].

### 12.7.2 traStat (32)

**procedure** traStat(a : integer ;nbp : integer) ;

Cette instruction est produite à la fin de la compilation d'un appel à une opération. Lors de l'exécution, elle complète la structure créée par l'exécution de l'instruction *reserverBloc* (structure qui se trouve à *nbp* positions sous le sommet de pile) en introduisant l'adresse de retour (qui est l'adresse suivant l'adresse de cette instruction). Cette structure est promue (nouveau) bloc de liaison. Enfin cette structure affecte le compteur ordinal avec la valeur *a*.

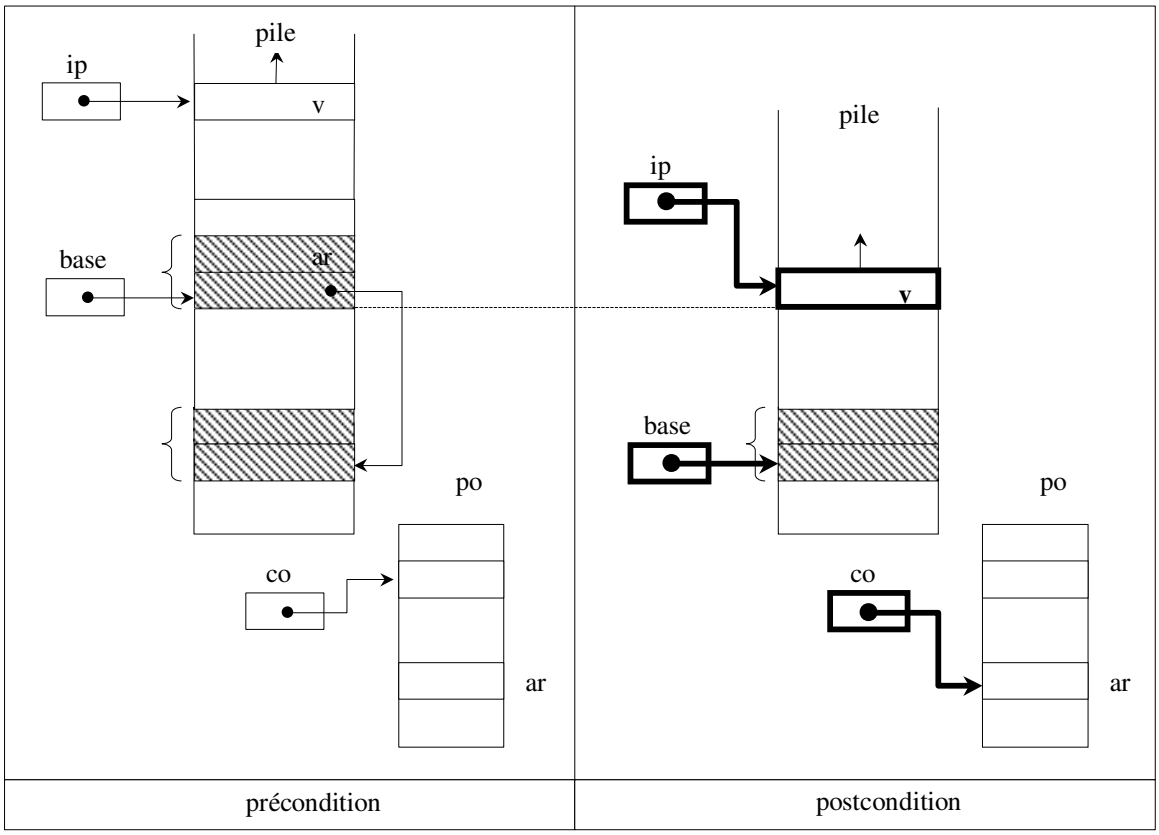
5. En effet il faut pouvoir utiliser le bloc courant pour la création des paramètres effectifs ainsi que pour traiter correctement l'imbrication des appels de fonctions.




### 12.7.3 retourFonct (33)

**procedure** retourFonct() ;

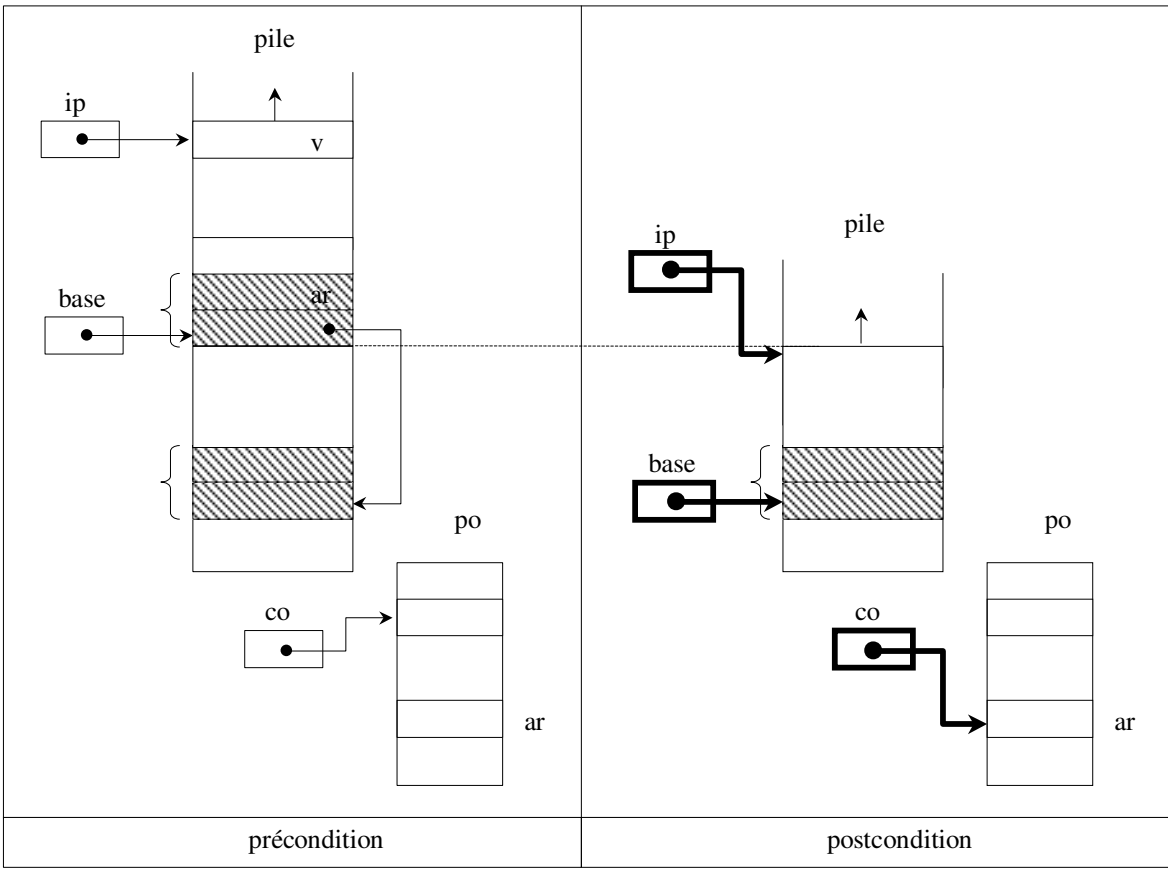
Cette instruction est produite à la fin de la compilation d'une instruction *return* dans une fonction. Outre son rôle dans le retour, elle assure que la valeur en sommet de pile sera le résultat de l'appel.




### 12.7.4 retourProc (34)

**procedure** retourProc() ;

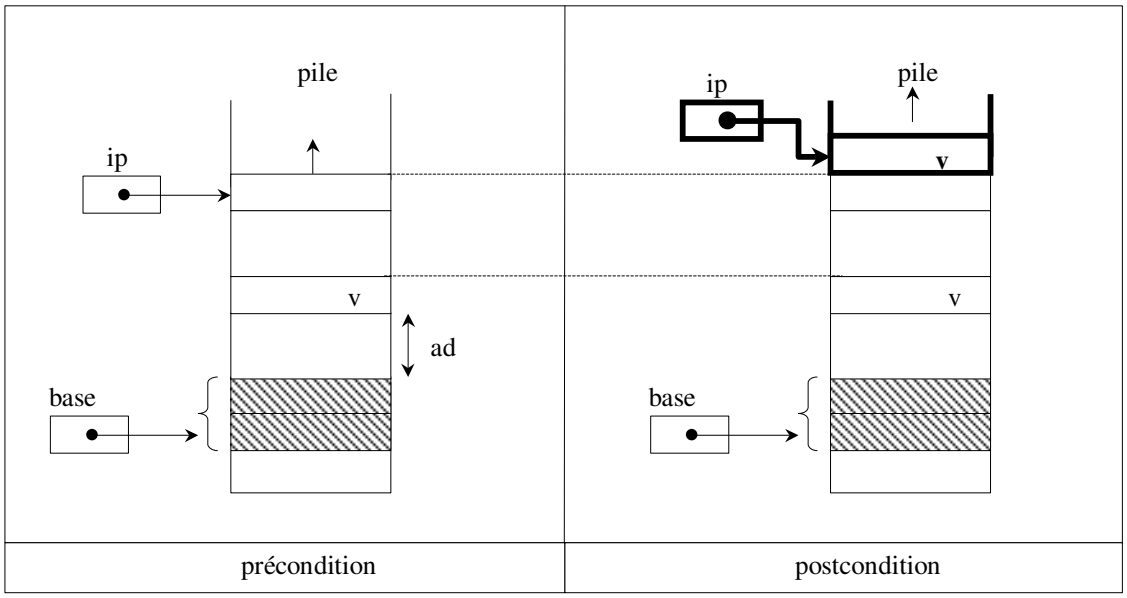
Cette instruction est produite à la fin de la compilation d'une procédure. Elle assure le retour à l'appelant.




### 12.7.5 empilerParam (35)

**procedure** empilerParam(ad : integer) ;

Cette instruction permet de gérer les paramètres effectifs.




# 13 Schéma de compilation et d'exécution

## 13.1 Introduction

Nous présentons ici la compilation des différentes constructions d'un programme NIL-NOVI PROCÉDURAL. La compilation y est vue comme un processus inductif : on distingue donc les cas de base (déclaration de variables, expressions réduites à une constante, une variable, un paramètre, un attribut, ordre de lecture) et les cas inductifs, pour lesquels la compilation d'une structure $A$ constituée des sous-structures $X$ et $Y$ est composée de la compilation de $X$ et de $Y$, ce que l'on formalisera de la manière suivante, en notant $\{P\}$ la compilation d'un programme source $P$ :

$$\{A\} = \Phi(\{X\}, \{Y\})$$

Dans la suite $as(< V >)$ signifie « adresse statique de l'identificateur $< V >$ » et $ai(< O >)$ « adresse d'implantation de l'opération $< O >$ ».

## 13.2 Unité principale

Soit $< O >$ la déclaration des opérations et $< V >$ la déclaration des variables.

$$\left\{\begin{array}{l} \textbf{procedure } \mathrm{p} \textbf{ is} \\ \quad < \mathtt{O} > \\ \quad < \mathtt{V} > \\ \textbf{begin} \\ \quad < \mathtt{B} > \\ \textbf{end}. \end{array}\right\} = \begin{array}{l} \mathtt{debutProg}(); \\ \mathtt{tra(ad1)}; \\ \{< \mathtt{O} >\}; \\ \mathtt{ad1} : \{< \mathtt{V} >\}; \\ \{< \mathtt{B} >\}; \\ \mathtt{finProg}() \end{array}$$

## 13.3 Ordre d'écriture

Voir la machine nilNovi Algorithmique.

## 13.4 Variables

Nous considérons ici que l'on sait compiler des expressions. Cette lacune sera comblée dans la section suivante.

### 13.4.1 Déclaration

On considère la déclaration successive de $n, (n > 0)$ variables.

$$\boxed{\texttt{v}_1, \ldots, \texttt{v}_\texttt{n} : < \texttt{type} > = \texttt{reserver(n)}}$$

### 13.4.2 Affectation

Soit $< E >$ une expression. Dans le cas d'une variable locale le schéma de compilation est :

$$\boxed{\begin{array}{lll} & & \texttt{empilerAd(as(}< \texttt{V} >\texttt{))}; \\ \{< \texttt{V} >:=< \texttt{E} >\} & = & \{< \texttt{E} >\}; \\ & & \texttt{affectation()} \end{array}}$$

Dans le cas d'une variable globale on a par contre

$$\boxed{\begin{array}{lll} & & \texttt{empiler(as(}< \texttt{V} >\texttt{))}; \\ \{< \texttt{V} >:=< \texttt{E} >\} & = & \{< \texttt{E} >\}; \\ & & \texttt{affectation()} \end{array}}$$

## 13.5 Expressions

Voir la machine nilNovi Algorithmique.

## 13.6 Séquentialité

Voir la machine nilNovi Algorithmique.

## 13.7 Ordre de lecture

L'ordre de lecture *get* permet de lire une variable ou un paramètre effectif d'entrée-sortie depuis le clavier. *get* ne permet de lire que des valeurs entières. Dans le cas de variables locales nous avons :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empilerAd}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

Dans le cas de variables globales nous avons :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empiler}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

Et dans le cas de paramètres formels d'entrée-sortie (cf. § 13.10.2) :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empilerParam}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

## 13.8 Alternatives

Voir la machine nilNovi Algorithmique.

## 13.9 Boucle

Voir la machine nilNovi Algorithmique.

## 13.10 Procédures

Dans une première étape nous étudions la compilation d'une procédure sans paramètre et de son appel. Ensuite nous considérons les cas des procédures *avec* paramètres.

### 13.10.1 Procédure sans paramètre

**Compilation de la définition**

$$\left\{\begin{array}{l}\textbf{procedure}\ \text{p}\ \ \textbf{is}\\ \quad < \texttt{A} >\\ \textbf{begin}\\ \quad < \texttt{B} >\\ \textbf{end}.\end{array}\right\} = \begin{array}{l}\{< \texttt{A} >\};\\ \{< \texttt{B} >\};\\ \texttt{retourProc}()\end{array}$$

**Compilation de l'appel**

Soit $p$ une procédure sans paramètre.

$$\{\texttt{p}()\} = \begin{array}{l}\texttt{reserverBloc}();\\ \texttt{traStat}(\texttt{ai}(\texttt{p}), 0);\end{array}$$

### 13.10.2 Procédure avec paramètres

On rappelle que le langage nilNovi Algorithmique distingue deux types de paramètres : les paramètres d'entrée (mode *in*) et les paramètres d'entrée-sortie (mode *in out*). Dans le premier cas le paramètre effectif transmis est une *expression* qui est évaluée. Le paramètre est considéré dans le corps de l'opération comme une constante, que l'on peut consulter mais pas modifier. Dans le second cas (paramètre d'entrée-sortie) le paramètre effectif est une *variable* ou un paramètre d'entrée/sortie qui peut, lors de l'appel, contenir une valeur. Le paramètre peut être modifié et consulté dans le corps de l'opération. Le principe du traitement des paramètres découle de ces éléments et s'articule autour des conventions suivantes :

- dans le cas d'un paramètre formel de mode *in*, le paramètre effectif est empilé lors de l'appel (au-dessus du bloc de liaison),
- dans le cas d'un paramètre formel de mode *in out,* l'*adresse dynamique* du paramètre effectif est empilée à l'appel (au-dessus du bloc de liaison).

La compilation de l'en-tête de la définition de la procédure ne produit pas de code :

$$\{\textbf{procedure } \mathtt{p}(\mathtt{p}_1, \ldots, \mathtt{p}_n : \ldots) \textbf{ is}\} \;=\; (\mathtt{rien})$$

**Paramètres de mode *in***

Considérons la procédure $p$ suivante :

```
procedure p(...,<X> : in <type>;...) is
    ...
begin
    ...
    ... <X>...
    ...
end;
```

et l'appel :

```
p(...,<E>,...)
```

où $< E >$ est le paramètre effectif correspondant à $< X >$ .

**Paramètre formel :** le $< X >$ de l'en-tête ne produit aucun code objet. Le $< X >$ du bloc impératif de la procédure joue le rôle d'une expression et se compile comme une variable locale de la manière suivante :

$$\{< \mathtt{X} >\} = \begin{array}{l} \mathtt{empilerAd}(\mathtt{as}(< \mathtt{X} >)); \\ \mathtt{valeurPile}() \end{array}$$

**Paramètre effectif :** dans l'appel, $< E >$ est une expression. Elle se compile comme toute expression.

**Paramètres de mode *in out***

Considérons la procédure $p$ suivante :

```
procedure p(...,<X> : in out <type>;...) is
    ...
begin
    ...
    ... <X>:=<E>;
    ... <X> ...
    ...
end;
```

où la première occurrence de $< X >$ dans le corps de la méthode en fait un usage de type variable et la seconde est une sous-expression. Soit l'appel :

```
p(...,<V>,...)
```

où $< V >$ est une variable (ou un paramètre formel de mode *in out*) qui constitue le paramètre effectif correspondant à $< X >$.

**Paramètre formel :** le $< X >$ de l'en-tête ne produit aucun code objet. Utilisé comme une variable $< X >$ se compile de la manière suivante :

$$\{< \texttt{X} >:=< \texttt{E} >\} \quad = \quad \begin{array}{l} \texttt{empilerParam}(\texttt{as}(< \texttt{X} >)); \\ \{< \texttt{E} >\}; \\ \texttt{affectation}() \end{array}$$

Utilisé comme une expression $< X >$ se compile de la manière suivante :

$$\{< \texttt{X} >\} \quad = \quad \begin{array}{l} \texttt{empilerParam}(\texttt{as}(< \texttt{X} >)); \\ \texttt{valeurPile}() \end{array}$$

**Paramètre effectif :** trois cas sont à considérer : $< V >$ est une variable globale, $< V >$ est une variable locale, $< V >$ est un paramètre formel.
Cas d'une variable globale :

$$\{< \texttt{X} >\} \quad = \quad \texttt{empiler}(\texttt{as}(< \texttt{V} >))$$

Cas d'une variable locale :

$$\{< \texttt{X} >\} \quad = \quad \texttt{empilerAd}(\texttt{as}(< \texttt{V} >))$$

Cas d'un paramètre formel :

$$\{< \texttt{X} >\} \quad = \quad \texttt{empilerParam}(\texttt{as}(< \texttt{V} >))$$

## 13.11 Fonctions

L'étude de la compilation des fonctions vient compléter l'étude de la compilation des expressions. La compilation et l'exécution d'une fonction ne sont pas fondamentalement différentes de celles d'une procédure. Un point l'en distingue : les fonctions délivrent une valeur. Cette particularité est prise en compte par l'instruction de retour *retourFonct.*

### Compilation de la définition

Soit une fonction dotée de $n$ paramètres formels.

$$\left\{ \begin{array}{l} \textbf{function } \mathrm{f}(\mathrm{p}_1, \ldots, \mathrm{p}_n) \textbf{ return } < \texttt{t} > \textbf{ is} \\ \quad < \texttt{A} > \\ \textbf{begin} \\ \quad < \texttt{B} > \\ \textbf{end}; \end{array} \right\} = \begin{array}{l} \{< \texttt{A} >\}; \\ \{< \texttt{B} >\} \end{array}$$

Une fonction contient toujours au moins (et son exécution s'achève toujours par) une instruction *return* $< E >$. Une telle instruction se compile de la manière suivante :

$$\{\texttt{return} < \texttt{E} >\} \quad = \quad \begin{array}{l} \{< \texttt{E} >\}; \\ \texttt{retourFonct}() \end{array}$$

Concernant les paramètres effectifs (de mode *in* uniquement) ils sont traités de la même façon que dans les procédures.

### Compilation de l'appel

Soit la fonction $f$, et $p_1, \ldots, p_n$ $n$ paramètres effectifs.

$$\{\mathtt{f}(\mathtt{p}_1, \ldots, \mathtt{p}_n)\} \quad = \quad \begin{array}{l} \texttt{reserverBloc}() \\ \{\mathtt{p}_1\}; \\ \cdots \\ \{\mathtt{p}_n\}; \\ \texttt{traStat}(\texttt{ai}(\mathtt{f}), \mathtt{n}); \end{array}$$

# 14 Exemple

Cet exemple montre comment se compile une procédure, une fonction ainsi que leurs appels.

```
01 : procedure pp is
02 :    function f(i: integer) return integer is
03 :       //pré: i>=0
04 :    begin
05 :       if i=0 then
06 :          return 1
07 :       else
08 :          if i=1 then
09 :             return 2
10 :          else
11 :             return i+f(i-1)+f(i-2)
12 :          end
13 :       end
14 :    end ;
15 :    procedure p(j: in out integer) is
16 :    begin
17 :       j:=f(f(j))
18 :    end;
19 :    k: integer;
20 : begin
21 :    get(k) ;
22 :    p(k) ;
23 :    put(k)
24 : end.
```

Le code objet correspondant à la compilation de ce programme se présente de la manière suivante (le numéro de la ligne source est rappelé à droite de la ligne objet) :

```
01 :   debutProg()                    01
02 :   tra(45)                        02
03 :   empilerAd(0)                   05
04 :   valeurPile()                   05
05 :   empiler(0)                     05
06 :   egal()                         05
07 :   tze(11)                        05
08 :   empiler(1)                     06
```

```
09 :   retourFonct()                    06
10 :   tra(45)                          06
11 :   empilerAd(0)                     08
12 :   valeurPile()                     08
13 :   empiler(1)                       08
14 :   egal()                           08
15 :   tze(17)                          08
16 :   empiler(2)                       09
17 :   retourFonct()                    09
18 :   tra(35)                          09
19 :   empilerAd(0)                     11
20 :   valeurPile()                     11
21 :   reserverBloc()                   11
22 :   empilerAd(0)                     11
23 :   valeurPile()                     11
24 :   empiler(1)                       11
25 :   moins()                          11
26 :   traStat(3,1)                     11
27 :   add()                            11
28 :   reserverBloc()                   11
29 :   empilerAd(0)                     11
30 :   valeurPile()                     11
31 :   empiler(2)                       11
32 :   moins()                          11
33 :   traStat(3,1)                     11
34 :   add()                            11
35 :   retourFonct()                    11
36 :   empilerParam(0)                  17
37 :   reserverBloc()                   17
38 :   reserverBloc()                   17
39 :   empilerParam(0)                  17
40 :   valeurPile()                     17
41 :   traStat(3,1)                     17
42 :   traStat(3,1)                     17
43 :   affectation()                    17
44 :   retourProc()                     17
45 :   reserver(1)                      19
46 :   empiler(0)                       21
47 :   get()                            21
48 :   reserverBloc()                   22
49 :   empiler(0)                       22
50 :   traStat(34,1)                    22
51 :   empiler(0)                       23
```

```
52 :   valeurPile()                    23
53 :   put()                           23
54 :   finProg()                       24
```

L'application systématique du schéma de traduction des alternatives conduit à produire du code mort : c'est le cas des instructions *tra* situées aux adresses 10 et 18. Elles succèdent à une instruction *retourFonct* et ne sont pas référencées par une instruction de contrôle : elles ne seront jamais exécutées. Un compilateur industriel se devrait d'optimiser le code produit de façon (notamment) à ne pas introduire de code mort.

# Troisième partie
# nilNovi Objet

Cette partie du document concerne la définition du langage objet nilNovi Objet. nilNovi Objet est un langage objet rudimentaire destiné à faciliter la compréhension des mécanismes mis en œuvre lors de la *compilation* et de l'*exécution* de programmes objets. nilNovi Objet est un langage objet *compilé* et *fortement typé,* il s'apparente par ces aspects à des langages industriels tels que Delphi, Simula, Eiffel, Java, Ada 95, voire C++. Il s'écarte dans sa philosophie des langages interprétés tels que SmallTalk. La grammaire de nilNovi Objet est présentée à l'annexe 3. Cette partie n'est pas une introduction à la programmation objet, elle suppose en particulier que le lecteur est familier avec les techniques qui prévalent dans ce paradigme de programmation.

## 15 Types, classes et objets

Cette section peut être sautée en première lecture, il est cependant important d'y revenir une fois achevée la lecture de la troisième partie. L'objectif de cette section est de préciser les notions de type, de classe et d'objet, et les relations qu'elles entretiennent du point de vue de la mise en œuvre de nilNovi Objet.

Bien que les notions de classe et d'objet soient disjointes dans nilNovi Objet [6], elles s'apparentent par certains aspects. Nous mettons l'accent sur leurs analogies mais aussi sur leurs différences.

Tant pour les classes que pour les objets on distingue trois niveaux :

— Le niveau « identificateur » (qui sera appelé « identificateur d'objet » dans le cas des objets et « identificateur de classe » sinon). Cette notion n'a pas d'existence à l'exécution : elle n'a de sens que sur un plan symbolique (le programme source). S'ils sont typés par une classe, une variable globale, une variable locale, un attribut ou un paramètre sont des identificateurs d'objet.

— Le niveau « référence » (de classe ou – implicitement – d'objet). Au moment de l'exécution une référence est matérialisée par une cellule mémoire dont le contenu *désigne,* selon la nature de la référence, un objet ou une classe. Cette notion n'a de sens que du point de vue de l'exécution.

— Le niveau « représentation » qui, selon le cas, est un objet ou une classe. Cette notion n'a de sens que du point de vue de l'exécution.

Du point de vue qui nous intéresse ici, les différences entre les notions de classe et d'objet sont les suivantes :

1. Un identificateur d'objet peut « donner naissance » à une ou plusieurs références (objet) qui peuvent coexister. La relation – immatérielle – qui lie référence et identificateur d'objet est du type « fonction totale non injective ». Au contraire

6. Ce n'est pas le cas dans tous les langages objets, ainsi dans le langage Smalltalk, une classe est un objet.

un identificateur de classe « donne naissance » à une et une seule référence de classe. La relation - immatérielle - qui lie référence de classe et identificateur de classe est bijective et, une fois établie, constante et définitive.

2. Une référence d'objet peut être dans l'un des deux états suivants : elle ne désigne rien, ou elle désigne un objet. La relation qui (à l'exécution) lie la référence et l'objet désigné est matérialisée par un pointeur (une adresse). L'absence de désignation est matérialisée par une adresse fictive (-1). La relation de désignation est du type « fonction partielle non injective », elle est variable dans le temps mais il existe des contraintes sur la nature de l'objet désigné (ces contraintes seront explicitées à travers des exemples). Une référence de classe désigne toujours une classe. Une fois établie, cette relation est bijective, constante et définitive. Elle est matérialisée par un pointeur.
3. Un objet désigne une représentation de classe (ce qui permet en particulier à l'objet d'identifier et d'exécuter ses méthodes virtuelles). Une fois établie, cette relation est constante. À l'inverse une représentation de classe ne désigne rien [7].
4. Un objet non désigné par une référence est inutile car inaccessible. Il pourrait donc être éliminé (notion de glaneur, ou ramasse-miettes). Cette fonctionnalité n'est pas spécifiée dans nilNovi Objet où une fois créé, un objet persiste jusqu'à la fin de l'exécution.

La figure 3 symbolise les relations existant entre objets et classes. Notons qu'un objet peut contenir une référence (si l'attribut correspondant à ce champ de l'objet est d'un type classe).

Nous sommes maintenant à même de formuler des définitions concernant les types.

**Définition 1 (Type d'un identificateur d'objet)** *Étant donné un identificateur d'objet, son type est l'identificateur de classe qui a servi à sa déclaration.*

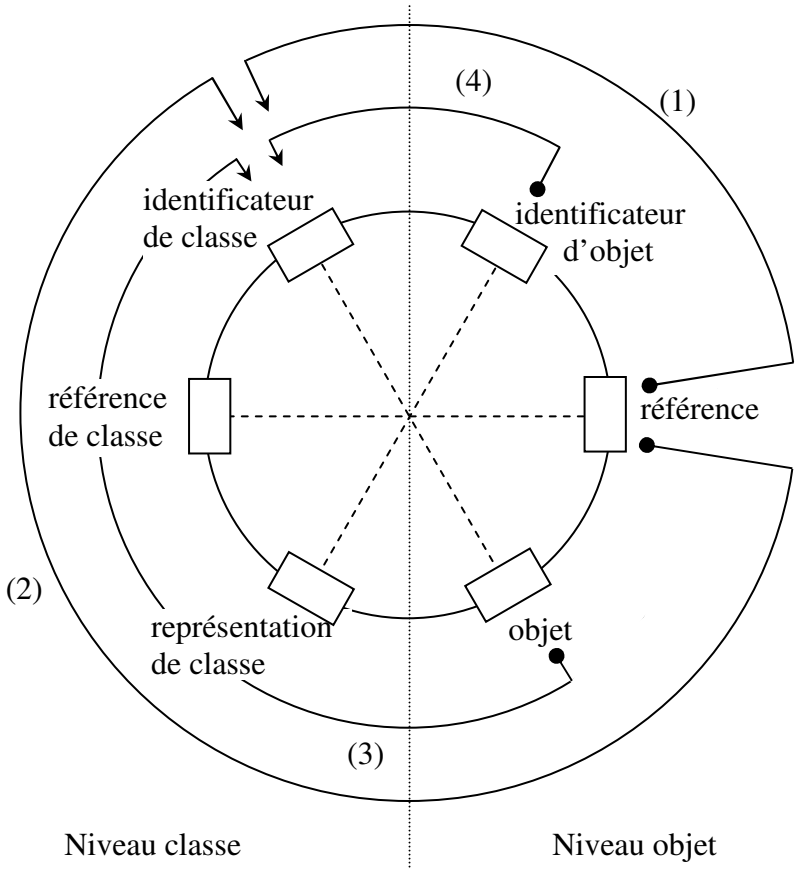
**Définition 2 (Type (statique) d'une référence)** *Le type statique d'une référence est le type de l'identificateur d'objet qui lui a donné naissance.*

**Définition 3 (Type d'un objet)** *Le type d'un objet est l'identificateur de la classe désigné par l'objet.*

**Définition 4 (Type dynamique d'une référence)** *Le type dynamique d'une référence est le type (s'il existe) de l'objet désigné par la référence. Le cas échéant la référence ne possède pas de type dynamique.*

Il faut à présent aborder, toujours sous l'aspect du typage, le concept d'expression (d'objet ou de classe). Soit $cl$ une classe, $a$ et $b$ des identificateurs d'objets du type $cl$. Soit *create* le constructeur de la classe $cl$ et $f$ et $g$ des méthode-fonctions de la classe $cl$. Quand on rédige des affectations telles que :

7. Qui soit du niveau objet, une classe désigne cependant le code de ses méthodes virtuelles.




(1) : type statique d'une référence
(2) : type dynamique d'une référence
(3) : type d'un objet
(4) : type d'un identificateur d'objet

FIGURE 3 – Typage et relations entre objets et classes

```
a:=b
a:=cl.create(5,true)
a:=nil
a:=b.f(3).g(5)
```

à gauche du symbole d'affectation se trouve un identificateur d'objet et à droite une *expression d'objet.*

**Définition 5 (Type d'une expression d'objet)** *Le type d'une expression d'objet se définit par induction sur la structure de l'expression.*

**Base.** *Si l'expression est* **nil***, son type n'est pas identifiable mais il est compatible avec tout type objet. Si l'expression est un identificateur d'objet son type est celui de l'identificateur d'objet. Si l'expression est l'invocation d'un constructeur, son type est l'identificateur de classe sur lequel s'applique le constructeur.*

**Induction.** *Si l'expression est l'invocation d'une méthode fonction sur une expression d'objet son type est le type délivré par la méthode.*

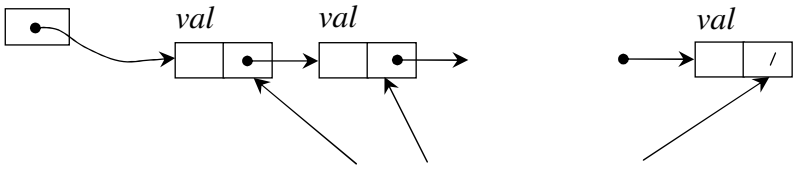
Lors de son exécution, une expression donne naissance à une référence dont le type statique est celui de l'expression et le type dynamique éventuel le type de l'objet désigné par la référence, s'il existe.

Pour ce qui concerne les classes, les expressions (de classes) se réduisent à l'utilisation d'un identificateur de classe comme préfixe de l'appel d'un constructeur.

**Exemple.**

Considérons l'exemple 1 de la page 51.

1. *liste* (l. 02) est un identificateur de classe.
2. *svt* (l. 04), *s* (l. 06), *e* et *tete* (l. 25) sont des identificateurs d'objets du type *liste.*
3. *e* (resp. *tete*) va donner naissance à une seule référence, lors de l'exécution de la ligne l. 25. Cette référence est du type statique *liste.*
4. Après exécution de la ligne 27, la référence issue de *tete* ne désigne aucun objet. Elle a toujours le type statique *liste,* elle n'a pas de type dynamique.
5. Après exécution de (au moins une fois) la ligne 32, la référence issue de *tete* désigne un objet de type *liste.* Le type dynamique de la référence est donc *liste.*
6. L'identificateur d'objet *svt* (l. 04) peut donner naissance à plusieurs références existant simultanément. Ainsi, suite à l'exécution de la boucle l. 27 – l. 34, la référence issue de l'identificateur d'objet *tete* désigne une « liste chaînée » matérialisant plusieurs références issues de *svt* :




références issues de *svt*

Dans la suite, quand il n'y a pas ambiguïté, on s'autorise l'abus de langage consistant à formuler « la référence issue de la variable objet X » par « la référence X ».

# 16 Introduction : nilNovi Objet par l'exemple

Un programme nilNovi Objet se compose d'un nom permettant son identification (situé entre les mots-clés *procedure* et *is*) d'une partie déclarative (située entre les mots-clés *is* et *begin*) et d'une partie impérative (située entre les mots-clés *begin* et *end*). Typiquement la partie déclarative est constituée de la description d'une hiérarchie de classes suivie de la déclaration des variables (scalaires ou objet). Les seuls types scalaires autorisés sont les entiers (*integer*) et les booléens (*boolean*).

Une classe décrit la *structure* et le *comportement* des objets. Elle peut comprendre :

— la mention de la classe dont elle hérite (facultative),
— les attributs propres à la classe considérée (facultatifs),
— les méthodes : un constructeur (obligatoire), des procédures et fonctions (facultatives).

## 16.1 Exemple 1

```
01 : procedure p is
02 :    type liste is class
03 :       val: integer;
04 :       svt: liste;
05 :    interface
06 :       constructor create(v: integer;s: liste);
07 :       function valeur() return integer;
08 :       function suivant() return liste;
09 :    implementation
10 :       constructor create is
11 :       begin
12 :          val:=v;
13 :          svt:=s
14 :       end;
15 :       function valeur is
```

```
16 :        begin
17 :           return val
18 :        end;
19 :        function suivant is
20 :        begin
21 :           return svt
22 :        end;
23 :     end ;
24 :     i: integer;
25 :     e, tete: liste;
26 : begin
27 :     tete:=nil ;
28 :     i:=1 ;
29 :     while i/=10 loop
30 :        //la liste des valeurs de i-1 à 1 est créée,
31 :        //elle est désignée par la variable tete
32 :        tete:=liste.create(i,tete);
33 :        i:=i+1
34 :     end;
35 :     e:=tete ;
36 :     while e/=nil loop
37 :        //les valeurs de la liste depuis tete inclus
38 :        //jusqu'à e exclus sont affichée
39 :        put(e.valeur());
40 :        e:=e.suivant()
41 :     end
42 : end.
```

**Commentaires**

1. Le programme *p* ci-dessus construit une liste chaînée par insertion en tête. Chaque cellule de la liste est une instance de la classe (un objet de type) *liste*. Chaque objet comprend deux champs. Le premier, *val*, est destiné à recevoir une valeur entière, le second, *svt*, est destiné à désigner la cellule suivante de la liste, si elle existe.
2. Les lignes 02 à 23 déclarent le type classe *liste*. Cette déclaration comprend la déclaration de deux attributs propres[8] : *val* et *svt*, et la *déclaration* de trois méthodes : *create* (un constructeur, c'est-à-dire une méthode qui crée un objet du type considéré et qui – en général – l'initialise), *valeur* et *suivant* (deux méthodes-fonctions, qui permettent de consulter l'état d'un objet, et de ses champs

8. C'est-à-dire d'attributs directement déclarés dans la déclaration de la classe et non dans une classe ancêtre.

en particulier). nilNovi Objet exige que chaque classe possède un (et un seul) constructeur. Bien que provoquant un effet de bord (la création de l'objet) un constructeur possède le statut de fonction (et non celui de procédure). L'appel d'un constructeur se comporte comme l'évaluation d'une expression. Il est donc interdit d'écrire une *instruction* telle que

```
liste.create(2,nil)
```

par contre, dans un contexte convenable, il est possible d'écrire

```
return liste.create(2,nil)
```

puisqu'une expression est exigée après le mot-clé *return.* De même il est possible d'écrire l'appel suivant à une procédure (fictive) *q*

```
q(liste.create(12,nil))
```

qui aurait un paramètre formel de type *liste* et de mode *in*.

3. Les lignes 03 et 04 définissent les attributs (propres) de la classe. Les lignes 05 à 08 constituent l'*interface* de la classe. On note que les attributs ne font pas partie de l'interface : ils ne peuvent être manipulés en dehors de la classe [9], ils sont cependant consultables par des méthodes-fonctions, et modifiables par des méthodes-procédures. Ils peuvent être initialisés par les constructeurs. Les lignes 09 à 22 constituent l'*implémentation* : elles décrivent en particulier comment les méthodes sont mises en œuvre (leurs *définitions*). On remarque que les paramètres formels des méthodes sont spécifiés dans l'interface mais ne sont pas répétés dans l'implémentation. Dans l'implémentation lorsqu'un attribut de la classe considéré est utilisé, il dénote à l'exécution le champ correspondant de l'instance. C'est par exemple le cas des attributs *val* et *svt* dans le corps du constructeur *create*, lignes 12 et 13.
4. Les lignes 03 et 04 déclarent les deux attributs de la classe liste : *val* et *svt*. Le premier, *val*, est un attribut de type entier, le second, *svt*, est un attribut du type de la classe en cours de déclaration (cette technique de référence est, dans le domaine de la programmation objet, l'équivalent des techniques « pointeur » dans le domaine de la programmation classique).
5. On note, en ligne 02, le mot clé *type*, qui met en évidence le fait que *l'on ne puisse* (contrairement à certains langages) déclarer directement des objets : il est nécessaire de passer par une déclaration de classe.
6. À la ligne 21 on note l'usage de l'instruction *return*. Cette instruction ne peut être utilisée que dans le corps d'une méthode-fonction. Son exécution d'une part délivre la valeur de l'expression qui suit (obligatoirement) le mot-clé *return* et d'autre part achève l'exécution de la fonction.

9. Et de la hiérarchie de classe à laquelle ils appartiennent. Ce point est développé ci-dessous.

7. Les lignes 24 et 25 déclarent les variables globales du programme *p*. À la ligne 24, *i* est une variable entière. À la ligne 25 se trouve la déclaration des variables objet *e* et *tete*. Une variable objet est à l'origine d'une référence destinée à *désigner* un objet (d'un point de vue interne une référence contient un pointeur, mais ce terme ne sera plus utilisé dans cette partie du document).
8. Une classe possède des *attributs*. Une instance de classe (un objet) possède des *champs*. Ainsi la classe *liste* a comme attributs *val* et *svt*, et les instances de la classe *liste* (comme par exemple les objets qui seront désignés par la variable *e*) sont dotés des champs *val* et *svt*.
9. La ligne 27 affecte à la référence *tete* la valeur de la constante objet **nil.** L'effet de cette affectation est que la référence *tete* ne désigne aucun objet.
10. Les lignes 27 à 34 constituent une boucle (avec son initialisation aux lignes 27 et 28). Il n'y a pas d'autre type de boucle en nilNovi Objet que les boucles *while*.
11. Les lignes 30, 31, 37 et 38 contiennent chacune un commentaire « ligne ». Ce type de commentaire débute par le délimiteur // et s'achève à la première fin de ligne rencontrée. Un commentaire ne peut débuter à l'intérieur d'un identificateur, ni à l'intérieur d'un mot-clé ou d'un délimiteur.
12. La ligne 32 contient un appel au constructeur sous la forme *liste.create*. L'appel d'un constructeur a pour effet d'une part de créer (par effet de bord) un objet du type de la classe préfixe et d'autre part de délivrer comme résultat une référence qui désigne l'objet qu'il a créé. Les paramètres du constructeur sont habituellement utilisés pour initialiser les champs de l'objet qui va être créé. Si le constructeur mentionné n'est pas le constructeur de la classe préfixe, une erreur de compilation est produite. Nous verrons dans les exemples qui suivent que le type (statique) de la référence associée à la variable objet affectée et l'objet créé peuvent ne pas être du même type. Un constructeur ne contient pas d'instruction *return* car le résultat est implicitement et obligatoirement une référence désignant l'instance nouvellement créée.
13. Aux lignes 32 et 33 on note que le « ; » est un opérateur de composition séquentielle de programmes (et non un terminateur d'instruction).
14. Les lignes 35 à 41 décrivent une seconde boucle. Son initialisation est située à la ligne 35.
15. La ligne 39 est une instruction de sortie (*put*). Cette instruction a comme paramètre effectif une expression entière qui est ici représentée par un appel de la méthode-fonction *valeur* appliquée à l'objet désigné par la référence *e*.
16. La ligne 40 représente une affectation à la variable *e* de l'expression représentée par un appel de la méthode-fonction *suivant* appliquée à l'objet désigné par *e* (on remarque l'usage de la notation pointée (*e.suivant*()) pour l'application d'une méthode – différente d'un constructeur – à une expression d'objet).

**Ce qu'il faut retenir de cet exemple**

1. Un programme nilNovi Objet permet de représenter des *classes,* qui sont *instanciées* par des *objets.* Les objets peuvent être *désignés* par des références issues de variables objets, de champs objets ou de paramètres objets.
2. Une classe possède des *attributs* dont les instances sont les *champs* de l'objet. Les attributs peuvent être de type scalaire (*integer* ou *boolean*) ou de type classe. Dans ce dernier cas les champs correspondants peuvent *désigner* des objets. Ce procédé permet de créer et de manipuler des structures de données complexes.
3. On distingue trois types de méthodes : les constructeurs, les méthodes-procédures et les méthodes-fonctions. Les constructeurs sont des fonctions qui permettent de créer un nouvel objet. Une classe possède un et un seul constructeur [10]. Les méthodes-procédures sont des procédures qui s'appliquent à un objet. Les méthodes-fonctions sont des fonctions qui s'appliquent à un objet et délivrent une valeur. Les appels aux constructeurs sont préfixés par une *classe,* les appels aux autres méthodes sont préfixées par une expression *d'objet.*
4. Les structures de contrôle de la programmation algorithmique classique sont disponibles en nilNovi Objet (boucle *while*, alternatives simple et double).
5. Il n'y a, en nilNovi Objet, que des attributs et des méthodes *d'instance.* Il n'y a pas d'attributs ni de méthodes de *classes* [11] : les classes ne sont pas des objets en nilNovi Objet.
6. La procédure qui tient lieu de programme principal ne peut être récursive.
7. Un identificateur est constitué de lettres (majuscules ou minuscules) ou de chiffres. Il débute obligatoirement par une lettre. La casse est significative.
8. Les opérateurs relationnels $=$ et $/=$ sont disponibles pour tous les types. Dans le cas des objets ces opérateurs représentent l'égalité et la différence « de surface » [12], (l'identité). En particulier si $eo_1$ et $eo_2$ sont deux expressions d'objets, l'expression relationnelle

   $$eo_1 = eo_2$$

   exige que $type(eo_1) \leq type(eo_2)$ ou que $type(eo_2) \leq type(eo_1)$, et délivre le résultat *true* si et seulement si les références résultant de l'évaluation des expressions $eo_1$ et $eo_2$ désignent le même objet. On note que cet opérateur ne peut être utilisé pour déterminer si deux objets différents ont la même valeur, il faudrait disposer pour cela de l'égalité profonde. On rappelle que les autres opérateurs relationnels ne sont autorisés que pour le type *integer.*

## 16.2 Exemple 2

Cet exemple revient sur la notion de méthode et illustre celles de sous-classe, d'héritage et de polymorphisme. Il précise enfin les notions de *type statique* et de *type dynamique.*

10. Ceci est une restriction propre à nilNovi Objet.
11. Dans la mesure où il s'applique à une classe, on peut cependant assimiler un constructeur à une méthode de classe.
12. Par opposition à l'égalité et la différence profonde.

```
01 : procedure pr is
02 :    type a is class
03 :       x: integer;
04 :    interface
05 :       constructor create(i: integer);
06 :       procedure p();
07 :       procedure q();
08 :    implementation
09 :       constructor create is
10 :       begin
11 :          x:=i
12 :       end;
13 :       procedure p is
14 :       begin
15 :          put(-1)
16 :       end;
17 :       procedure q is
18 :       begin
19 :          put(1)
20 :       end;
21 :    end;
22 :    type b is class(a)
23 :       y: integer;
24 :    interface
25 :       constructor create(i,j: integer);
26 :       procedure q();
27 :       procedure r();
28 :    implementation
29 :       constructor create is
30 :       begin
31 :          x:=i;
32 :          y:=j
33 :       end;
34 :       procedure q is
35 :       begin
36 :          put(2)
37 :       end;
38 :       procedure r is
39 :       begin
40 :          put(3)
41 :       end;
42 :    end;
```

```
43 :    e: a;
44 :    f: b;
45 : begin
46 :    e:=a.create(12) ;
47 :    f:=b.create(12,0) ;
48 :    e.q() ;
49 :    e:=f ;
50 :    e.q() ;
51 :    e:=b.create(3,6)
52 : end.
```

**Commentaires et définitions**

1. Les lignes 02 à 42 déclarent deux classes : la classe $a$ et la classe $b$. À la ligne 22 la qualification « *(a)* » spécifie que la classe $b$ est une sous-classe de la classe $a$. La classe $a$ n'est elle-même sous-classe d'aucune autre classe : c'est une classe *racine*. La classe $b$ est une classe fille de la classe $a$. La classe $a$ est la classe mère de la classe $b$. De manière générale le procédé permet de définir plusieurs hiérarchies de classes. Une hiérarchie de classes dote l'ensemble des classes d'une structure d'ordre partiel. On pourra donc parler de l'ensemble des classes (ou des types) {inférieures, inférieures ou égales, supérieures, supérieures ou égales} à une classe (ou un type) donnée. $a$ est une classe inférieure ou égale $a$, $b$ est une classe inférieure à $a$.
2. On note, aux lignes 05 et 25, que chacune des classes $a$ et $b$ possède un constructeur. On rappelle qu'il est obligatoire pour une classe d'avoir un et un seul constructeur.
3. On appelle *attributs propres* d'une classe les attributs qui sont déclarés dans la définition de la classe. Ainsi $y$ est un attribut propre de la classe $b$.
4. On appelle *attributs hérités* d'une classe les attributs des classes supérieures à la classe considérée. On appelle *attribut* l'union des attributs hérités et des attributs propres (l'ensemble des attributs des classes supérieures ou égale). Il est interdit de redéfinir un attribut dans une classe inférieure. Alors que $y$ est un attribut propre de la classe $b$, $x$ est un attribut hérité, tandis que $x$ et $y$ constituent les attributs de la classe $b$. Par contre si on avait déclaré $x$ : *boolean* dans la classe $b$ une erreur serait signalée à la compilation ($x$ est déjà un attribut hérité de la classe $b$).
5. Les méthodes d'une classe sont connues statiquement (à la compilation du programme). On distingue les constructeurs et les méthodes virtuelles (méthodes-fonctions et méthodes-procédures). Parmi les méthodes virtuelles on considère :
   (a) les méthodes *propres* (qui n'appartiennent pas aux classes supérieures),
   (b) les méthodes *redéfinies* (qui appartiennent à l'une des classes supérieures mais qui sont redéclarées localement dans l'interface et redéfinies avec un nouveau corps dans l'implémentation),

(c) les méthodes *héritées* (qui appartiennent aux classes supérieures et qui ne sont pas redéfinies).

Les méthodes de la classe $a$ sont *create* (le constructeur, des lignes 05, et 09 à 12), $p$ et $q$ (méthodes propres). Les méthodes de la classe $b$ sont *create* (le constructeur de la ligne 25 et des lignes 29 à 33), $q$ (méthode qui est *redéfinie* pour la classe $b$ à la ligne 26, et aux lignes 34 à 37), $p$ (méthode qui est *héritée* de la classe mère $a$) et $r$ (méthode propre). Une méthode virtuelle redéfinie doit avoir le même profil (nom, nature, type statique, mode et ordre des paramètres), que la méthode qu'elle redéfinit. Ainsi la méthode $q$ de la classe $b$ a le même profil que la méthode $q$ de la classe mère $a$. Cette forme de redéfinition ne s'applique qu'aux méthodes virtuelles (nous verrons pourquoi ci-dessous). Ainsi le constructeur *create* de $b$ possède (incidemment) le même nom que le constructeur de la classe $a$ mais n'a aucun rapport avec le constructeur homonyme de $a$. Il n'a pas le même profil que celui de $a$. Par contre on note que le constructeur de $b$ utilise l'attribut propre $y$ de $b$ de même que l'attribut $x$, hérité de $a$.

6. Les lignes 43 et 44 déclarent deux variables objets : $e$ et $f$. Le type utilisé dans la déclaration dote les variables d'un type. $e$ est du type $a$ tandis que $f$ est du type $b$.
7. À la ligne 46 l'objet obtenu par l'appel du constructeur *create* de la classe $a$ est affecté à la variable objet $e$. De même à la ligne 47 l'objet obtenu par l'appel du constructeur *create* de la classe $b$ est affecté à la variable objet $f$.
8. Les lignes 48 à 51 illustrent la notion de polymorphisme. Avant l'exécution de la ligne 48, la référence $e$ possède le type dynamique $a$ (elle désigne un objet de type $a$). La méthode $q$ invoquée à la ligne 48 sur la variable objet $e$ est celle du type dynamique de la référence $e$ c'est-à-dire $a$. À la ligne 49 l'objet désigné par $f$ est affecté à la variable objet $e$. Cette affectation est légale, elle a en particulier pour effet de modifier le type dynamique de la référence $e$ (son type statique n'étant jamais affecté) qui devient $b$. La méthode appelée à la ligne 50 sur la variable objet $e$ est déterminée par le type dynamique de la référence $e$ : c'est donc la méthode $q$ de la classe $b$ qui est invoquée. À la ligne 51 on note que l'on crée un objet de type $b$, qui est désigné par la référence $e$ (qui conserve donc son type dynamique $b$). La possibilité pour une référence de voir ainsi évoluer son type dynamique est appelé polymorphisme.
9. Que se serait-il passé si, à la place de la ligne 49, nous avions écrit

```
f:=e ;
```

Une erreur aurait été signalée à la compilation, informant qu'il est nécessaire, dans une affectation d'une variable objet de type $a := b$ que le type de l'identificateur d'objet $a$ soit supérieur ou égal au type de l'expression $b$. Cette règle s'applique également dans le cas de passage de paramètres de type classe. Voir [12], p. 270 pour une justification de cette règle.

10. Que se serait-il passé si, à la suite de la ligne 49 on avait ajouté

```
e.r()
```

On pourrait penser que, puisque à ce moment là de l'exécution la référence $e$ est du type dynamique $b$, il est légitime de lui appliquer une méthode propre à la classe $b$. Ce n'est pas le cas, cette instruction provoquera une erreur de compilation pour violation de la règle suivante : dans une application de méthode virtuelle de type $o.m$ ($o$ : expression d'objet, $m$ : méthode) le type statique de $o$ doit posséder la méthode $m$ (par appropriation : méthode propre, par héritage ou par redéfinition). Voir [12] p. 266 pour une justification de cette règle.

11. Revenons sur les lignes 46 à 51. Le constructeur qui est invoquée est *toujours* celui de la classe préfixe. Par contre dans le cas de méthodes-procédures ou de méthodes-fonctions, la méthode invoquée est toujours la méthode du *type dynamique* (lignes 48 et 50). Ceci caractérise les méthodes *virtuelles.* Alors que le code d'un constructeur peut être lié à celui de l'appel dès la compilation, dans le cas des autres méthodes ce n'est qu'à l'exécution que l'on connaît la méthode qui doit être invoquée et que l'on peut lier son code à celui de l'appel.
12. Au total, l'exécution du programme *pr* aura pour effet d'écrire successivement les valeurs 1 et 2.
13. En nilNovi Objet toute utilisation d'un identificateur doit être précédée de sa déclaration. Ainsi dans l'exemple ci-dessus il serait impossible de déclarer dans la classe $a$ un attribut de type $b$ puisque à cet endroit la classe $b$ n'est pas encore déclarée. Cette limitation présente des inconvénients rédhibitoires pour un usage industriel de nilNovi Objet. Elle ne limite néanmoins pas les objectifs didactiques. De la même façon, il est impossible d'utiliser les variables globales dans les classes, puisqu'elles ne sont pas encore déclarées.

**Ce qu'il faut retenir de cet exemple**

1. nilNovi Objet permet de définir des hiérarchies de classes (rappel).
2. Toute classe doit posséder un et un seul constructeur.
3. Les constructeurs sont des méthodes *statiques,* les autres méthodes sont *virtuelles.*
4. L'ensemble des attributs d'une classe est défini grâce à l'héritage.
5. Il existe trois façons de doter une classe d'une méthode virtuelle : par *appropriation* (en déclarant une méthode *propre* à la classe), par *héritage* (par défaut on hérite des méthodes *virtuelles* des classes supérieures) et par *redéfinition.* Dans ce dernier cas il est nécessaire de déclarer à nouveau la méthode dans la partie interface de la classe et de *définir* la méthode dans la partie implémentation.
6. On sait que les références possèdent un type statique (le type de la classe utilisée dans la déclaration) et éventuellement un type dynamique (le type de l'objet désigné à l'exécution). Le type dynamique d'une référence qui désigne effectivement un objet est obligatoirement *inférieur ou égal* au type statique. Une référence qui ne désigne pas d'objet (valeur à **nil**) n'a pas de type dynamique.

7. Dans une affectation $a := e$ d'une variable objet $a$ par une expression d'objet $e$, il est nécessaire que le type de l'identificateur d'objet $a$ soit supérieur ou égal au type de l'expression $e$. De par l'affectation, le type dynamique de la référence $e$ devient le type dynamique de la référence $a$.
8. Dans une application de méthode virtuelle de type $o.m$ ($o$ : expression d'objet, $m$ : méthode) la méthode $m$ doit être une méthode (appropriée, redéfinie ou héritée) du type statique de $o$. Autrement dit, si $t$ est le type statique de $o$, $m$ doit être une méthode définie dans une classe supérieure ou égale à $t$.
9. Schématiquement on peut dire qu'une méthode est syntaxiquement du type de l'expression d'objet sur laquelle elle s'applique et sémantiquement du type dynamique de la référence correspondante.

## 16.3 Exemple 3

L'exemple suivant est un classique de la programmation objet. Il montre en particulier l'intérêt des méthodes virtuelles et de la liaison dynamique. Il définit des classes qui permettent de construire des expressions arithmétiques entières utilisant les opérateurs *plus* (+) et *mult* (*), et de les évaluer.

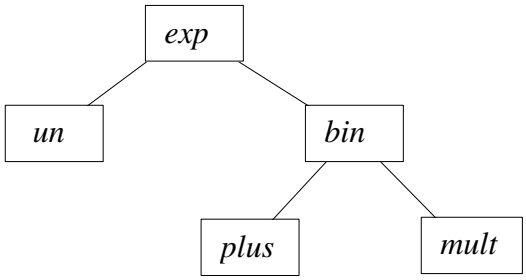
```
01 : procedure e is
02 :    type exp is class
03 :    interface
04 :       constructor create() ;
05 :       function eval() return integer ;
06 :    implementation
07 :       constructor create is
08 :       begin
09 :          error(1)
10 :       end ;
11 :       function eval is
12 :       begin
13 :          error(2) ;return 0
14 :       end ;
15 :    end ;
16 :    type un is class(exp)//expressions unaires
17 :       v : integer ;
18 :    interface
19 :       constructor create(i : integer) ;
20 :       function eval() return integer ;
21 :    implementation
22 :       constructor create is
23 :       begin
```

```
24 :          v : =i
25 :       end ;
26 :       function eval is
27 :       begin
28 :          return v
29 :       end ;
30 :    end ;
31 :    type bin is class(exp)//expressions binaires
32 :       g,d : exp ;
33 :    interface
34 :       constructor create() ;
35 :       function eval() return integer ;
36 :       function op(a,b : exp) return integer ;
37 :    implementation
38 :       constructor create is
39 :       begin
40 :          error(3)
41 :       end ;
42 :       function eval is
43 :       begin
44 :          return this.op(g,d)
45 :       end ;
46 :       function op is
47 :       begin
48 :          error(4) ;return 0
49 :       end ;
50 :    end ;
51 :    type plus is class(bin)//expressions g+d
52 :    interface
53 :       constructor create(l,r : exp) ;
54 :       function op(a,b : exp) return integer ;
55 :    implementation
56 :       constructor create is
57 :       begin
58 :          g :=l ;
59 :          d :=r
60 :       end ;
61 :       function op is
62 :       begin
63 :          return a.eval()+b.eval()
64 :       end ;
65 :    end ;
66 :    type mult is class(bin) //expressions g*d
```

```
67 :     interface
68 :        constructor create(l,r : exp) ;
69 :        function op(a,b : exp) return integer ;
70 :     implementation
71 :        constructor create is
72 :        begin
73 :           g :=l ;
74 :           d :=r
75 :        end ;
76 :        function op is
77 :        begin
78 :           return a.eval()*b.eval()
79 :        end ;
80 :     end ;
81 :     a,b,e : un ;
82 :     c : mult ;
83 :     d : plus ;
84 :     f : exp ;
85 :     val : integer ;
86 : begin
87 :     get(val) ;
88 :     a :=un.create(val) ;
89 :     b :=un.create(2) ;
90 :     c :=mult.create(a,b) ;
91 :     e :=un.create(1) ;
92 :     d :=plus.create(e,c) ;
93 :     f :=d ;
94 :     put(f.eval())
95 : end.
```

**Commentaires**

1. La hiérarchie des classes se représente par l'arbre suivant :



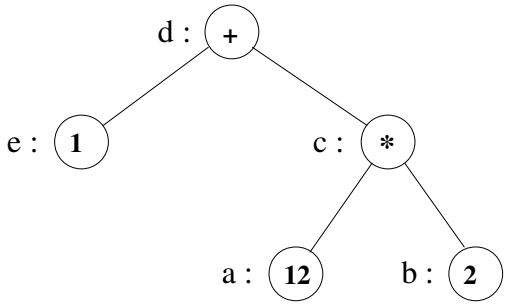
La définition de la portée d'un identificateur de classe (cf. p. 69) entraîne qu'il n'est pas possible d'avoir de circuit dans le graphe matérialisant la hiérarchie, il s'agit toujours d'un arbre fini.

2. La classe *exp* (lignes 02 à 15) définit les expressions arithmétiques en général. Elle n'a pas d'attribut mais elle est dotée d'un constructeur (obligatoire), et d'une méthode-fonction *eval* dont le rôle est de permettre d'évaluer la valeur de l'expression représentée par l'objet. On note que le corps des méthodes (*create* comme *eval*) contient un appel à la procédure prédéfinie *error* dont l'exécution a pour but de déclencher une erreur et de procéder à un arrêt intempestif du programme en affichant la valeur de l'expression passée en paramètre. Aucune référence n'est destinée à avoir le type *dynamique exp* (le cas échéant l'exécution d'une de ses méthodes provoquerait un abandon par l'appel de la procédure prédéfinie *error*). La classe *exp* n'est donc qu'un artifice pour permettre de définir une racine commune (avec une méthode virtuelle commune : *eval*) à la description. Par contre, nous verrons un exemple à partir de la ligne 84, où une variable objet peut avoir le type *exp*. La classe *exp* telle qu'elle est définie ici représente ce qui est parfois appelé *classe abstraite* (Delphi, Java) [13] ou *classe différée* (Eiffel).
3. La classe *un* (lignes 16 à 30) est une sous-classe de la classe *exp*. Elle permet de définir des expressions réduites à un entier. Elle est constituée d'un attribut propre, $v$, du constructeur *create*, et d'une redéfinition de la méthode-fonction *eval* qui spécifie que l'évaluation d'une expression de type *un* se réduit à délivrer la valeur du champ $v$. On note que l'on utilise les attributs de la classe sans préciser de quel objet il s'agit : implicitement il s'agit de l'objet pour qui on exécute l'opération.
4. La classe *bin* (lignes 31 à 50) définit les expressions binaires. Elle possède deux attributs $g$, et $d$ (sous-arbres gauche et droit) qui sont du type *exp*. La classe *bin* est, à l'instar de la classe *exp*, une classe « abstraite » : ses méthodes *create* et *op* ne sont pas véritablement implantée à ce niveau mais à un niveau inférieur. Par contre la méthode *eval* est bien implantée à ce niveau : elle se définit par l'appel de la méthode (virtuelle) *op* sur les deux attributs. Ce procédé va permettre de ne plus redéfinir la méthode *eval* aux niveaux inférieurs, seule la méthode *op*, spécifique de chaque sous-classe, sera implantée dans chacune des sous-classes. Le principe de l'appel de méthode virtuelle permet de « choisir » la bonne instance de *op* appelée dans la méthode *eval*.
5. À la ligne 44 on note l'utilisation du pronom **this** qui désigne l'objet pour lequel on exécute l'opération. nilNovi Objet oblige à appliquer une méthode virtuelle sur un *objet*. Cet objet est issu d'un identificateur d'objet ou est créé « sur place ». Dans les deux cas il est désigné par une référence. Dans certains cas on souhaite appliquer une méthode sur l'objet courant, qui ne possède pas de nom. On utilise alors le pronom **this**. On reviendra sur cette notion dans l'exemple suivant.
6. La classe *plus* (lignes 51 à 65) définit les expressions binaires additives. Elle hérite des attributs de la classe *bin* (les deux « opérandes » $g$ et $d$), elle définit le

13. En Delphi ce sont les méthodes (et non les classes) qui peuvent être abstraites.

constructeur *create* et la méthode-fonction *op*. Cette dernière délivre la somme de l'évaluation des deux sous-arbres passés en paramètres.

7. La classe *mult* (lignes 66 à 80) est l'homologue de la classe *bin* pour la multiplication. On note en particulier que l'implémentation de la classe *op* délivre cette fois le produit de l'évaluation des deux sous-arbres passés en paramètres.
8. Les lignes 81 à 85 déclarent les variables objets. On note que, bien que la classe *exp* soit « abstraite » on s'autorise à déclarer la variable objet $f$ comme étant de ce type.
9. La ligne 87 montre comment on peut lire une variable entière (depuis le clavier).
10. Les lignes 88 à 93 permettent de construire l'arbre suivant (en supposant que 12 est la valeur lue) :




Il aurait été possible (et probablement plus élégant) de définir l'objet $d$ sans passer par des variables auxiliaires $a, b, c$. On aurait alors remplacé les lignes 88 à 92 par

```
d :=plus.create(un.create(1),mult.create(un.create(val),un.create(2)))
```

Rien n'interdirait dans ce cas que $d$ soit déclaré du type *exp*. La création (par *plus.create*(...)) aurait alors pour effet d'attribuer directement à la référence $d$ le type dynamique *plus*.

11. La ligne 93 réalise l'affectation de la variable $f$ (de type *exp*) par l'objet résultant de l'évaluation de l'expression $d$ (désigné par une référence de type statique et dynamique *plus*). On note que la règle d'affectation de variables objets est bien respectée : le type de $d$ est bien inférieur ou égal au type de $f$. L'effet de la ligne 94 sera d'évaluer (appel de la méthode *eval*) l'objet désigné par la référence $f$. La méthode *eval* exécutée est la méthode héritée de la classe *bin*. Elle fait elle-même appel à la méthode *op*. La méthode *op* qui sera retenue est celle du type dynamique de la référence $f$ : la méthode *op* de la classe *plus*.
12. On note que, pour « satisfaire » le compilateur, les méthodes-fonctions *eval* (ligne 13) et *op* (ligne 48) contiennent une instruction *return*. Celle-ci est factice dans la mesure où elle ne peut s'exécuter, étant précédée d'une instruction *error*.

**Ce qu'il faut retenir de cet exemple**

1. nilNovi Objet ne permet pas de déclarer explicitement des classes abstraites : s'il existe une déclaration de méthode, la définition correspondante existe obligatoirement et est située dans la partie *implementation* de la classe considérée.
2. La modification du type dynamique d'une référence peut se faire soit par affectation soit par passage de paramètre.
3. L'opération [14] *error* a comme paramètre une expression entière.

## 16.4 Exemple 4

Cet exemple, incomplet et sans signification particulière, va nous permettre de présenter quelques points spécifiques qui n'ont pas été rencontrés dans les exemples précédents (les variables locales) et de préciser des points qui n'ont pas été approfondis (le pronom **this**, les modes de passage de paramètres).

```
01 : procedure pp is
02 :    type a is class
03 :       r : integer ;
04 :    interface
05 :       constructor create(i : in integer) ;
06 :       procedure p(j : in out a ; k : in a) ;
07 :       procedure q() ;
08 :    implementation
09 :       constructor create is
10 :       begin
11 :          r:=i
12 :       end ;
13 :       procedure p is
14 :       begin
15 :          this.q() ;
16 :          j :=this ;
17 :          j :=k ;
18 :          k.q()
19 :       end ;
20 :       procedure q is
21 :       begin
22 :          r :=r+1
23 :       end ;
24 :    end ;
```

14. Le terme *opération* est utilisé comme terme générique recouvrant les termes de procédure et de fonction.

```
25 :     type b is class(a)
26 :        s : boolean ;
27 :     interface
28 :        constructor create(i : integer ; b : boolean) ;
29 :        procedure t(s : in integer) ;
30 :     implementation
31 :        constructor create is
32 :           ...
33 :        end ;
34 :        procedure t is
35 :           x :a ;
36 :        begin
37 :           ...
38 :        end ;
39 :     end ;
40 :     e :a ;
41 :     f :b ;
42 : begin
43 :     e :=a.create(6) ;
44 :     f :=b.create(4,true) ;
45 :     e.p(f,a.create(8));
46 :     if e=f then
47 :        put(0)
48 :     end
49 : end.
```

**Commentaires**

1. À la ligne 06 la méthode-procédure $p$ est dotée de deux paramètres dont le type est $a$ : le premier $j$, est passé en mode *in out* tandis que le second est passé en mode *in* (le mode implicite). Le mode *in* permet de consulter la valeur d'un paramètre, pas de la modifier. Un paramètre effectif de mode *in* est toujours une expression. Le mode *in out* permet aussi bien de modifier que de consulter le paramètre. Le paramètre effectif *in out* est *in fine* une entité objet (directement ou par passage de paramètre) .
2. À la ligne 15 se trouve l'instruction

   **this**.q()

   qui invoque la méthode-procédure $q$ sur l'objet pour qui s'exécute la méthode $p$ considérée. Le désignateur **this** peut être utilisé à chaque fois que l'on veut mentionner l'objet courant, il désigne toujours un objet effectif. Rappelons cependant que dans le cas d'un attribut on ne peut employer le pronom **this**. Les règles d'utilisation du pronom **this** sont développées à la section 17.8.

3. À la ligne 16 nous trouvons un autre exemple de l'utilisation du désignateur **this**. L'instruction affecte au paramètre effectif correspondant à $j$ la valeur actuellement possédée par l'objet pour qui on exécute la méthode $p$ considérée. $j$ étant de mode *in out* il est possible de modifier sa valeur (par affectation ou par passage de paramètre).
4. La ligne 18 constitue un appel de la méthode $q$ sur l'objet désigné par le paramètre effectif $k$. $k$ est un paramètre formel de type $a$ et de mode $in$. Tout type inférieur ou égal à $a$ possède une méthode $q$. Le paramètre $k$ étant de mode $in$, il est impossible de le modifier (par exemple par une affectation), par contre rien n'interdit de modifier l'objet qu'il désigne. C'est le cas ici où, par l'appel de la procédure $q$, le champ $r$ de l'objet désigné sera incrémenté.
5. Les lignes 25 à 39 définissent la classe $b$, fille de la classe $a$. La classe $b$ possède les attributs $r$ (hérité) et $s$ (propre). Elle possède les méthodes *create* (constructeur, statique), $p$ et $q$ (héritées) et $t$ (méthode-procédure propre).
6. À la ligne 35 on note la déclaration de $x$, variable locale à la procédure $t$.
7. À la ligne 45 on invoque la méthode-procédure $p$ sur l'objet désigné par $e$ avec comme paramètres effectifs d'une part $f$ et d'autre part un objet de type $a$ créé « sur place » par appel du constructeur *create*. Le type statique du paramètre de $p$ est $a$. On peut donc, à l'instar de l'affectation, accepter comme paramètre effectif toute expression de type inférieur ou égal à $a$.
8. À la ligne 46 se trouve une expression booléenne construite à partir de deux variables objet. Nous avons vu, lors de l'étude de nilNovi Algorithmique, que les opérateurs relationnels $=$ et $/=$ permettent de construire une expression booléenne à partir de deux opérandes booléens ou entiers. De la même façon ces deux opérateurs permettent de construire une expression booléenne à partir de deux opérandes qui sont des expressions d'objet. Si $a$ et $b$ sont deux expressions d'objet, l'expression booléenne $a = b$ délivre la valeur $vrai$ si et seulement si $a$ et $b$ désignent le même objet. De même $a/= b$ délivre $vrai$ si et seulement si $a = b$ délivre $faux$. La compilation de $a = b$ provoque une erreur si $a$ et $b$ ne sont pas comparables. Les autres opérateurs relationnels ($<$, $<=$, etc.) ne peuvent être utilisés pour construire une expression booléenne à partir de deux expressions d'objet.

**Ce qu'il faut retenir de cet exemple**

1. L'appel d'une méthode est *toujours* préfixé (par une classe pour les constructeurs, par une expression d'objet pour les autres méthodes).
2. Le pronom **this** ne peut être utilisé que dans le corps d'une opération. Il se comporte comme un paramètre d'entrée anonyme.
3. Il existe deux modes de passage de paramètres, le mode *in* et le mode *in out*. Le premier permet de transmettre une valeur lors de l'appel. Le second (*in out*) transmet une variable qui peut lors de l'appel contenir une valeur qui peut être modifiée lors

de l'exécution. Le mode par défaut est *in*. Les paramètres des méthodes-fonctions et des constructeurs sont toujours de mode *in*.

# 17 Rattachement d'un identificateur à une entité : portée et visibilité des identificateurs

## 17.1 Introduction

Dans un programme nilNovi Objet il est possible d'utiliser un même identificateur pour désigner différentes entités. Cependant un usage particulier d'un identificateur ne doit pas être ambigu. Lever les ambiguïtés possibles se fait soit en exploitant le contexte (voir l'exemple ci-dessous) soit par des conventions sur le lien entre l'identificateur et l'une des entités possibles (voir le second exemple). Dans ce dernier cas on utilise les notions de portée et de visibilité.

**Exemples :**

```
01 : procedure pp is
02 :     type a is class
03 :         p : integer ;
04 :     interface
05 :         procedure p() ;
06 :             ...
07 :     implementation
08 :         procedure p() is
09 :             x :a ;
10 :         begin
11 :             a.p()
12 :         end ;
13 :         ...
14 :     end ;
15 : begin
16 :     ...
17 : end.
```

La ligne 03 définit l'identificateur $p$ comme un attribut entier de la classe $a$. La ligne 05 définit l'identificateur $p$ comme une méthode-procédure de la classe $a$. À la ligne 11 l'identificateur $p$ est utilisé. Fait-il référence à la variable ou à la méthode procédure ? La réponse est à rechercher dans le contexte : l'occurrence de $p$ est non ambiguë, il s'agit d'un appel de la méthode.

```
01 : procedure pp is
```

```
02 :     type a is class
03 :        t : integer ;
04 :           ...
05 :     implementation
06 :        procedure p() is
07 :           t : integer ;
08 :        begin
09 :           t :=45
10 :        end ;
11 :     end ;
12 :     ...
13 : begin
14 :     ...
15 : end.
```

Dans cet exemple la ligne 03 définit un attribut $t$ pour la classe $a$. À la ligne 07 on définit une variable locale $t$ pour la méthode-procédure $p$. La ligne 09 utilise l'identificateur $t$ dans une affectation. Cet usage est valable tant comme attribut que comme variable locale. Le contexte ne peut permettre de décider. Dans ce cas ce sont les notions de portée et de visibilité (voir ci-dessous) qui permettent de dire que l'affectation porte sur la variable locale.

Définir les notions de portée et de visibilité va permettre de répondre aux questions suivantes : étant donné une déclaration d'entité (classe, variable globale, attribut de classe, méthode, paramètre ou variable locale), dans quel fragment du texte du programme a-t-on le droit d'utiliser cet identificateur pour désigner l'entité en question (notion de portée) ? Cette liaison entité-identificateur peut-elle être momentanément invalidée par une autre déclaration du même identificateur pour désigner une entité différente (notion de visibilité) ?

Une question surgit cependant dans le cas des méthodes virtuelles à propos du lien entre l'appel et la méthode invoquée. Considérons l'exemple suivant :

```
01 : procedure pp is
02 :     type a is class
03 :        ...
04 :     interface
05 :        procedure p() is
06 :           ... ;
07 :     end ;
08 :     type b is class(a)
09 :        ...
10 :     interface
11 :        procedure p() is
12 :        ... ;
```

```
13 :     end ;
14 :     ...
15 :     x : a ;
16 :     y : b ;
17 : begin
18 :     ...
19 :     x.p() ;
20 :     ...
21 : end.
```

À quelle occurrence de la procédure virtuelle $p$ est lié l'appel de la ligne 19 ? À celle définie à la ligne 05 ? à celle de la ligne 11 ? Il est impossible de le savoir puisque le type dynamique de la référence $x$ est inconnu à la compilation. Cependant, d'un point de vue syntaxique, cela n'a pas d'importance. Nous savons d'une part que la méthode invoquée doit être une méthode du type statique de la référence (et donc du type de l'entité dont elle est issue), et d'autre part que la redéfinition conserve le profil. En conséquence, comme notre souci dans cette section concerne tout d'abord les vérifications extra-syntaxiques, il suffit d'associer un appel de méthode virtuelle à la déclaration présente dans la classe constituant le type.

## 17.2 Cas de l'identificateur de la procédure principale

**Portée.** La portée de l'identificateur de la procédure principale est vide.

**Visibilité.** Sans objet.

**Conflit.** Sans objet.

**Exemple :**

```
01 : procedure p is
02 :     type p is class
03 :         ...
04 :     end;
05 :     a: p;
06 : begin
07 :         ...
08 : end.
```

La déclaration de la ligne 02 est correcte, celle de la ligne 05 également, il n'y a pas de conflit avec l'identificateur de la procédure principale.

## 17.3 Cas d'un identificateur de classe et d'un identificateur de variable globale

**Portée.** La portée d'un identificateur de classe est constituée du texte du programme à partir de l'occurrence de la « ) » suivant l'identificateur de la classe mère ou, le cas échéant, à partir de l'occurrence du mot clé *class*. La portée d'un identificateur de variable globale est constituée du texte du programme à partir de l'occurrence de la déclaration.

**Visibilité.** Un identificateur de classe peut être masqué par un identificateur d'attribut, de variable locale ou de paramètre.

**Conflit.** Un conflit est possible (déclaration multiple) avec un autre identificateur de classe ou de variable globale.

**Exemple :**

```
01 : procedure pp is
02 :     type a is class( a )
03 :         ...
04 :         constructor create() ;
05 :         ...
06 :     end ;
07 :     type b is class
08 :         a :a ;
09 :     implementation
10 :         procedure p() is
11 :         begin
12 :             a :=a.create()
13 :         end ;
14 :     end ;
15 :     ...
16 : begin
17 :     ...
18 : end.
```

La déclaration de la ligne 02 est incorrecte : la portée de l'identificateur $a$ ne débute qu'après la « ) » , $a$ ne peut donc être utilisé comme classe mère de la classe $a$. La déclaration de la ligne 08 est correcte : le contexte détermine qu'après le « : » se trouve un identificateur de classe (ou de type scalaire). Par contre l'affectation de la ligne 12 est incorrecte, l'attribut à gauche du symbole d'affectation est bien identifié comme l'attribut de la classe $b$ en revanche le préfixe est lui aussi identifié comme l'attribut, cependant un constructeur (*create*) ne peut être préfixé par un attribut : une erreur sera détectée lors du traitement de l'appel du constructeur, ligne 12.

**Remarque.** Pour ce qui concerne la portée et la visibilité des identificateurs des types prédéfinis *integer* et *boolean*, tout se passe comme si le prologue suivant était compilé avant chaque programme nilNovi Objet :

```
01 : type integer is class
02 : end ;
03 : type boolean is class
04 : end ;
```

Cependant aucune classe ne peut hériter de ces deux classes.

## 17.4 Cas d'un identificateur d'attribut

**Portée.** La portée d'un identificateur d'attribut est constituée des déclarations des classes inférieures ou égales à la classe où apparaît la déclaration.

**Visibilité.** Un identificateur d'attribut peut être masqué par un identificateur de variable locale ou de paramètre dans la hiérarchie.

**Conflit.** Un conflit est possible (déclaration multiple) avec un autre identificateur d'attribut de la hiérarchie.

**Exemples :**

```
01 : procedure pp is
02 :     type a is class
03 :         x : integer ;
04 :             ...
05 :     end ;
06 :     type b is class(a)
07 :         x : boolean ;
08 :         ...
09 :     end ;
10 :     ...
11 :     type c is class
12 :         x : integer ;
13 :         ...
14 :     end ;
15 :     ...
16 : begin
17 :     ...
18 : end.
```

La déclaration de la ligne 07 est incorrecte : l'identificateur $x$ désigne déjà un attribut dans cette hiérarchie. Par contre la déclaration de la ligne 12 est correcte : les hiérarchies de $a$ et de $c$ sont disjointes.

```
01 : procedure pp is
02 :     type a is class
03 :         x : ...;
04 :         ...
05 :     implementation
06 :         procedure p() is
07 :             x : integer;
08 :         begin
09 :             ...
10 :             x :=...
11 :             ...
12 :         end ;
13 :     end ;
14 :     ...
15 : begin
16 :     ...
17 : end.
```

La déclaration de la ligne 07 dans la procédure auxiliaire $p$ est correcte. Elle masque celle de la ligne 03. En conséquence, à la ligne 10 l'identificateur $x$ désigne la variable locale déclarée à la ligne 07.

```
01 : procedure pp is
02 :     ...
03 :     type a is class
04 :         x : a;
05 :     interface
06 :         constructor x() ;
07 :         ...
08 :     implementation
09 :         procedure p() is
10 :         begin
11 :             ...
12 :             x :=a.x();
13 :             ...
14 :         end ;
15 :     end ;
16 :     ...
17 : begin
18 :     ...
19 : end.
```

La déclaration du constructeur $x$ à la ligne 06 est correcte bien qu'il existe un attribut de même nom. À la ligne 12 la première occurrence de $x$ désigne l'attribut tandis que la seconde désigne le constructeur.

## 17.5 Cas d'un identificateur de constructeur

**Portée.** La portée d'un identificateur de constructeur est constituée du texte du programme à partir de l'en-tête de la *définition* du constructeur (partie *interface*).

**Visibilité.** Pas de restriction.

**Conflit.** Un conflit est possible (déclaration multiple) avec un autre identificateur d'opération dans la même classe.

**Exemples :**

```
01 : procedure pp is
02 :     type a is class
03 :     interface
04 :         constructor creer() ;
05 :         procedure creer() ;
06 :         ...
07 :     end ;
08 :     type b is class(a)
09 :     interface
10 :         constructor creer(i : integer) ;
11 :         ...
12 :     end ;
13 : begin
14 :     ...
15 : end.
```

La déclaration de la ligne 05 est incorrecte : l'identificateur *creer* désigne déjà le constructeur de la classe *a*. La déclaration de la ligne 10 est correcte : il s'agit du constructeur de la classe *b*, qui est indépendant de celui de la classe *a*.

Compte tenu de la définition de la portée, un constructeur ne peut être utilisé avant d'être défini. Le cas échéant un message avertit le programmeur.

## 17.6 Cas d'un identificateur de méthode virtuelle

**Portée.** La portée d'un identificateur de méthode virtuelle est constituée du texte du programme à partir de la déclaration de la méthode.

**Redéfinition.** Un identificateur de méthode virtuelle peut être redéfini dans les classes inférieures à la classe où elle est définie. Dans ce cas le profil des deux

déclarations doit être identique (nom, ordre, type et mode de passage des paramètres, nature de la méthode).

**Visibilité.** Pas de restriction.

**Conflit.** Un conflit est possible (déclaration multiple) avec un autre identificateur d'opération dans la même classe.

**Exemples :**

```
01 : procedure pp is
02 :     type a is class
03 :     interface
04 :         procedure p() ;
05 :         function f() return boolean ;
06 :         ...
07 :     end ;
08 :     type b is class(a)
09 :     interface
10 :         procedure p() ;
11 :         procedure f(i : integer) ;
12 :         ...
13 :     end ;
14 : begin
15 :     ...
16 : end.
```

La redéclaration de la ligne 10 est correcte. Celle de la ligne 11 ne l'est pas : les opérations ne sont pas de même nature (fonction d'un côté, procédure de l'autre), en outre l'une possède un paramètre l'autre pas.

```
01 : procedure pp is
02 :     type a is class
03 :     interface
04 :         procedure p() ;
05 :             ...
06 :     end ;
07 :     type b is class
08 :     interface
09 :         procedure p() ;
10 :     implementation
11 :         procedure aux is
12 :             s :a ;
```

```
13 :           t :b ;
14 :       begin
15 :           ...
16 :           s.p() ;
17 :           t.p()
18 :       end ;
19 :       ...
20 :   end ;
21 : begin
22 :   ...
23 : end.
```

La déclaration de la méthode-procédure $p$ à la ligne 09 est correcte bien que la méthode procédure $p$ de la classe $a$ soit visible. À la ligne 16 c'est la méthode de la classe $a$ [15] qui est désignée par cet appel tandis qu'à la ligne 17 c'est la méthode de la classe $b$.

## 17.7 Cas d'un identificateur de paramètre ou de variable locale

**Portée.** La portée d'un identificateur de paramètre ou de variable locale est constituée du texte de l'opération où apparaît la déclaration (partie **implementation**), à partir de la déclaration.

**Visibilité.** Pas de restriction.

**Conflit.** Un conflit est possible (déclaration multiple) avec un autre identificateur de paramètre ou de variable locale.

**Exemples :**

```
01 : procedure pp is
02 :   type a is class
03 :     ...
04 :   interface
05 :     procedure p(x : integer) ;
06 :     procedure q(q : integer) ;
07 :     ...
08 :   implementation
09 :     procedure p is
10 :       x : boolean ;
11 :     begin
12 :       ...
```

15. On rappelle que compte tenu du polymorphisme, il est impossible de dire à la compilation quelle est précisément la méthode qui s'appliquera. Elle est cependant compatible avec celle du type de l'entité.

```
13 :         end ;
14 :         ...
15 :     end ;
16 : begin
17 :     ...
18 : end.
```

La déclaration (ligne 05) et la définition (lignes 09 à 13) de la méthode procédure $p$ sont incorrectes : il y a conflit entre le paramètre et la variable locale $x$. Par contre la déclaration de la méthode-procédure $q$ est correcte : le paramètre $q$ n'est pas en conflit avec le nom de la procédure.

## 17.8 Cas du pronom *this*

On rappelle que le mot clé **this** est destiné à représenter l'objet sur lequel on exécute une méthode. Il peut être assimilé à un attribut qui référencerait toujours l'objet courant. En conséquence il est obligatoire que ce mot-clé apparaisse uniquement dans la partie implémentation (et donc dans le corps des méthodes). Considérons l'appel

```
o1.p(12)
```

de la méthode-procédure $p$ sur l'objet $o1$. L'objet $o1$ peut être référencé implicitement dans la définition de la méthode-procédure $p$ en utilisant le pronom **this**. Cette référence est implicitement de mode *in* (il est impossible de modifier l'objet référencé bien qu'il soit possible de modifier ses champs). Ainsi dans l'exemple de la méthode $q$ ci-dessous

```
01 : procedure q is
02 :     ...
03 : begin
04 :     ...
05 :     if x=this then
06 :         this :=y
07 :     end
08 :     ...
09 : end ;
```

la ligne 05 est correcte : **this** permet la consultation de l'objet considéré. Par contre la ligne 06 est incorrecte puisque l'on tente de modifier la référence.

Dans le cas d'un constructeur, la sémantique de **this** est légèrement différente puisque le préfixe n'est pas un objet mais une classe. Considérons la classe $cl$ et son constructeur $c$, et l'appel :

```
cl.c(14)
```

Il est cependant encore possible d'utiliser **this** dans le corps du constructeur $c$ mais cette fois le pronom désigne non pas la classe $cl$ mais l'objet qui a été créé au début de l'appel du constructeur avant l'exécution de son code.

**Remarque :** du point de vue de la portée et de la visibilité, tout se passe comme si la déclaration

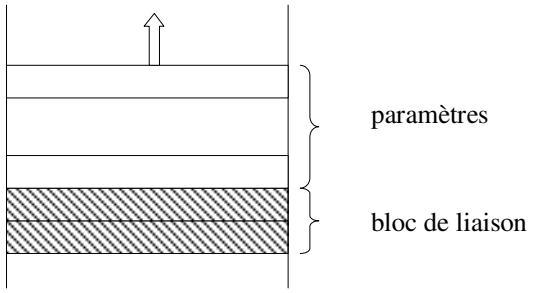
**this : in <c>**

était réalisée à l'entrée de chaque opération. ($< c >$ étant le nom de la classe en cours de définition).

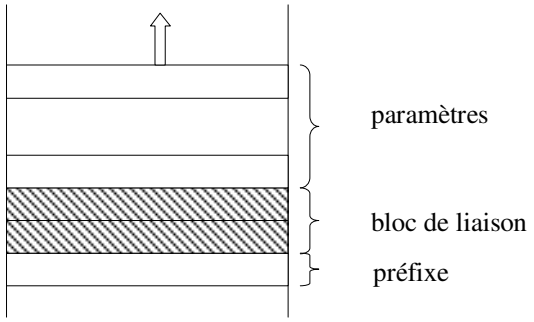
# 18 Attribution d'adresses statiques

## 18.1 Introduction

En préambule à cette section il faut savoir (cela sera détaillé dans les sections ultérieures) que dans un langage tel que NILNOVI OBJET, et si l'on excepte les objets qui sont gérés différemment, les entités (variables, paramètres, etc.) sont gérées dans une pile (la pile d'exécution). Notons également que lors de l'exécution d'une opération, un bloc de données (le bloc de liaison) est créé, il permet d'effectuer correctement le retour. Les paramètres effectifs sont placés au dessus du bloc de liaison.




Par contre, dans le cas d'une méthode, la classe ou l'objet sur lequel s'applique la méthode considérée (le *préfixe* de la méthode) est désigné depuis un emplacement situé *sous* le bloc de liaison.



Différentes raisons obligent, ou incitent, dans un langage tel que nilNovi Objet, à gérer dynamiquement la mémoire. Citons :

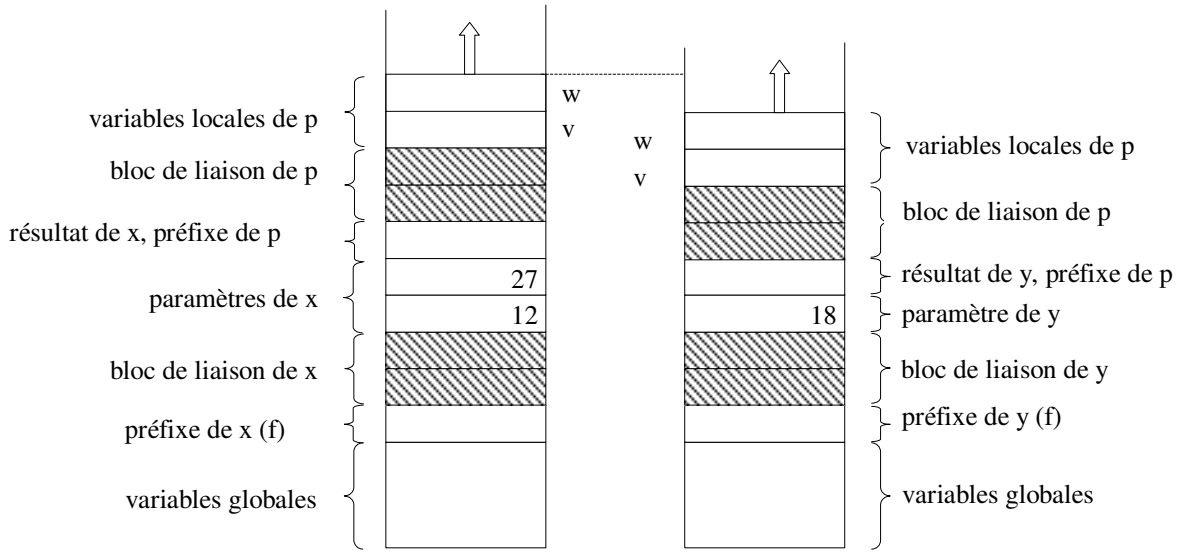
— une meilleure gestion de la place libre (seules les entités utiles existent au moment souhaité),

— la récursivité qui, du point de vue des variables locales, va conduire à associer au cours du temps des emplacements différents à un même identificateur.

Une difficulté surgit alors : à l'exception des variables globales, le compilateur n'est pas capable de calculer l'adresse définitive d'une variable. Pour lever cette difficulté on utilise une adresse intermédiaire entre l'adresse symbolique et l'adresse définitive (l'adresse « dynamique ») : c'est ce que nous appelons l'adresse *statique*. Cette adresse statique se présente comme un déplacement par rapport à une base.

**Exemple.** Dans l'exemple qui suit nous allons montrer que des variables locales (celles de la procédure $p$) peuvent se voir attribuer des adresses dynamiques différentes au cours de l'exécution. On suppose par ailleurs que les fonctions $x$ et $y$ n'ont pas de variables locales.

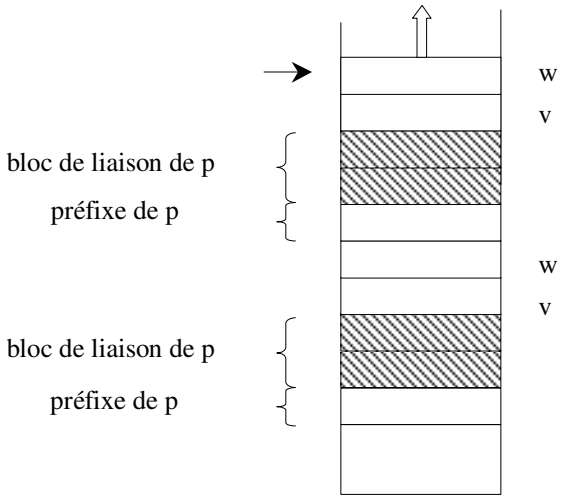
```
01 : procedure p is
02 :    type a is class
03 :       r : integer ;
04 :    interface
05 :       constructor create(...) ;
06 :       function x(x1,x2 : integer) return a ;
07 :       function y(y1 : integer) return a ;
08 :       procedure p();
09 :    implementation
10 :       ...
11 :       procedure p is
12 :          v,w : integer ;
13 :       begin
14 :          ...
15 :       end ;
16 :       ...
17 :    end ;
18 :    f :a ;
19 :    ...
20 : begin
21 :    ...
22 :    f.x(12,27).p() ;
23 :    f.y(18).p()
24 : end.
```




Ci-dessus la partie gauche de la figure est un instantané de la pile d'exécution au moment de l'appel de la procédure $p$ de la ligne 22. La partie droite est l'homologue pour la ligne 23.

On note effectivement que d'un appel à l'autre les adresses d'implantation des variables locales $v$ et $w$ sont différentes.

Le cas de figure suivant est encore plus caractéristique. Il fait l'hypothèse que la procédure $p$ est récursive [16]. Plusieurs instances des variables locales $v$ et $w$ existent, seule celle qui est en sommet de pile est accessible.




16. La récursivité est autorisée pour toutes les opérations (à l'exception du programme principal).

## 18.2 Expansion d'une classe

L'attribution des adresses statiques pour les entités situées à l'intérieur d'une déclaration de classe passe par l'utilisation de l'*expansion* de la classe (et non de la classe elle-même).

**Définition 6 (Expansion d'une classe)** *Soit $C$ le texte d'une classe. L'expansion $C'$ de $C$ est le texte obtenu en utilisant le procédé inductif suivant :*

**Base.** *Si $C$ est une classe racine, alors $C' = C$.*

**Induction.** *Sinon, soit $M$ la classe mère de la classe $C$, et soit $M'$ son expansion. L'expansion $C'$ est définie de la manière suivante :*

- *l'en-tête de $C'$ est*

  **type** `C` **is class**

  *(sans la qualification « (M) »),*
- *la partie « attributs » de $C'$ est la concaténation de la partie « attributs » de $M'$ et de la partie « attributs » de $C$,*
- *la rubrique « interface » de $C'$ est constituée de :*
  - *la spécification du constructeur de $C$,*
  - *la concaténation de la rubrique « interface » de $M'$ (sans le constructeur) et de la liste des méthodes propres de la rubrique « interface » de $C$.*
- *la rubrique « implementation » de $C'$ est la rubrique « implementation » de $C$.*

**Exemple.** L'exemple qui suit décrit l'expansion d'une classe fille. Considérons le programme (incomplet) suivant :

```
01 : procedure p is
02 :    type navire is class
03 :       longueur : integer ;
04 :       age : integer ;
05 :       tirantDEau : integer ;
06 :       aVoile : boolean ;
07 :    interface
08 :       constructor createNavire(...) ;
09 :       procedure vieillir();
10 :       function longueur() return integer ;
11 :       function tirant() return integer ;
12 :       function aMoteur() return boolean ;
13 :    implementation
14 :       ...
15 :    end ;
16 :    type paquebot is class(navire)
17 :       capacitePassager : integer ;
18 :       niveauLocation : integer ;
```

```
19 :     interface
20 :        constructor createPaquebot(...) ;
21 :        procedure reserver(n: integer);
22 :        function longueur() return integer ;
23 :     implementation
24 :        ...
25 :     end ;
26 :     ...
27 : begin
28 :     ...
29 : end.
```

L'expansion de la classe *navire* est la classe *navire* elle-même, tandis que l'expansion de la classe *paquebot* est :

```
16 :     type paquebot is class
03 :        longueur : integer ;
04 :        age : integer ;
05 :        tirantDEau : integer ;
06 :        aVoile : boolean ;
17 :        capacitePassager : integer ;
18 :        niveauLocation : integer ;
19 :     interface
20 :        constructor createPaquebot(...) ;
09 :        procedure vieillir();
22 :        function longueur() return integer ;
11 :        function tirant() return integer ;
12 :        function aMoteur() return boolean ;
21 :        procedure reserver(n: integer);
23 :     implementation
24 :        ...
25 :     end ;
```

## 18.3 Adresses statiques : attribution proprement dite

Dans la suite de la section nous montrons comment s'effectue l'attribution des adresses statiques des entités d'un programme nilNovi Objet.

**Adresses statiques des classes et variables globales.** Les adresses statiques des classes et variables globales sont attribuées séquentiellement, à partir de 0. Ces adresses correspondent à des adresses absolues dans la base de la pile d'exécution, à raison d'un emplacement par entité.

**Adresses statiques des attributs d'une classe.** L'attribution se fait à partir de l'expansion (cf. p. 78) de la classe. Les adresses sont attribuées séquentiellement à partir de 0.

**Adresses statiques de méthodes virtuelles.** L'attribution se fait également à partir de la rubrique « interface » de l'expansion de la classe. Les adresses sont attribuées séquentiellement à partir de 0.

**Adresses statiques des paramètres des opérations et des variables locales.** Elles sont considérées globalement et attribuées séquentiellement à partir de 0.

**Adresses statiques des préfixes de méthodes.** Elles valent systématiquement -3.

**Adresses statiques des constructeurs.** L'adresse statique d'un constructeur est l'adresse de la première instruction du code objet du constructeur.

**Exemple**

```
01 : procedure p is
02 :    type a is class
03 :       r : integer ;
04 :    interface
05 :       constructor create(r1 : integer) ;
06 :       procedure x(x1,x2 : integer) ;
07 :       procedure y(y1 : integer) ;
08 :    implementation
09 :       ...
10 :       procedure x is
11 :          v,w : integer ;
12 :       begin
13 :          ...
14 :          v :=x1 ;
15 :          ...this...
16 :          ...
17 :       end ;
18 :       ...
19 :    end ;
20 :    type b is class(a)
21 :       s : integer ;
22 :    interface
23 :       constructor creation(r1 : integer ;s1 : integer) ;
24 :       procedure y(y1 : integer) ;
25 :       procedure z() ;
26 :    implementation
27 :       ...
```

```
28 :     end ;
29 :     f,g : a ;
30 :     h : integer ;
31 : begin
32 :     f :=b.create(12,45) ;
33 : end.
```

La classe $a$ a comme expansion elle-même, la classe $b$ a comme expansion :

```
type b is class
   r : integer ;
   s : integer ;
interface
   constructor creation(r1 : integer ;s1 : integer) ;
   procedure x(x1,x2 : integer) ;
   procedure y(y1 : integer) ;
   procedure z() ;
implementation
   ...
end ;
```

**Commentaires**

1. Aux lignes 02, 20, 29 et 30 les adresses statiques sont attribuées séquentiellement aux identificateurs de classes et de variables $a(0), b(1), f(2), g(3), h(4)$.
2. À la ligne 03, l'attribut $r$ se voit attribuer l'adresse statique 0.
3. À la ligne 05 on attribue au paramètre formel $r1$ l'adresse 0.
4. À la ligne 06 est déclarée la méthode $x$. Elle se voit attribuer l'adresse statique 0, étant la première méthode virtuelle de la déclaration de la classe $a$. Ses paramètres se voient attribuer les adresses statiques 0 et 1.
5. La ligne 07 est similaire à la ligne 06. La méthode $y$ se voit attribuer l'adresse statique 1.
6. La méthode $x$ est définie aux lignes 10 à 17. Venant après le paramètre formel $x2$ doté de l'adresse 1, les variables locales $v$ et $w$ sont dotées des adresses statiques 2 et 3. La ligne 15 utilise le pronom **this** qui désigne l'objet en cours de traitement. Son adresse statique est -3.
7. Pour ce qui concerne la classe $b$ (ligne 20 et expansion), les attributs $r$ et $s$ se voient attribuer respectivement les adresses statiques 0 et 1. Les méthodes virtuelles $x, y$ et $z$ se voient attribuer les adresses statiques 0, 1 et 2.

# 19 La machine nilNovi Objet

Nous allons dans cette section étudier le « matériel » et la représentation des entités (classes et objets) de la machine nilNovi Objet. Le langage machine sera décrit dans une section ultérieure en utilisant une sémantique pré/post.

## 19.1 Le matériel de la machine nilNovi Objet

La machine nilNovi Objet est constituée de deux parties : la partie « données » et la partie « programme ». Nous allons présenter successivement ces deux parties.

La partie « données » se compose d'une zone de mémoire et de trois registres. La zone de mémoire est elle-même composée de trois parties : la pile d'exécution, la table des classes et le tas. Ces trois parties partagent le même espace d'adressage. La pile d'exécution est destinée à mémoriser les variables, paramètres et références ainsi que les informations nécessaires à l'exécution d'une opération. Elle permet également d'évaluer les expressions. Comme son nom l'indique, cette zone est gérée en pile. Elle est gérée par deux registres : *ip* et *base*. *ip* désigne le sommet de pile tandis que *base* désigne le premier élément du bloc de liaison [17] de l'opération en cours d'exécution.

La table des classes permet quant à elle d'accéder aux méthodes virtuelles de chacune des classes. Chaque entrée de la table des classes est elle-même une table. Chacune de ses propres entrées contient une information qui permet de retrouver le code des différentes méthodes virtuelles de la classe considérée. Lors de son élaboration, au début de l'exécution d'un programme nilNovi Objet, la table des classes est gérée par le registre $ipTas$.

Le tas est une structure de données qui contient la représentation des objets. Chaque objet est en principe désigné par (au moins) une référence (localisée dans la pile d'exécution ou dans le tas lui-même). Contrairement aux variables locales [18] et aux variables globales, les objets sont créés *explicitement*. En conséquence, par le jeu des affectations des variables objet, il est possible qu'un objet donné ne soit plus accessible (ni directement ni indirectement) depuis la pile d'exécution. Selon un principe de bonne gestion il serait raisonnable de récupérer la place occupée par de tels objets. Ceci pourrait se faire en mettant à disposition du programmeur des « destructeurs », homologue des constructeurs pour la récupération de place, ou (solution préférable en général) en intégrant à l'interpréteur un « ramasse-miettes » qui automatiserait le processus de récupération de place. Pour des raisons de temps ce point sera écarté. Compte tenu de la stratégie choisie pour la gestion du tas, celui-ci est géré en pile [19] grâce au registre $ipTas$.

La partie « programme » est destinée à recevoir le programme objet à des fins d'exécution. Elle est composée d'un tableau *po* constituant la mémoire de programme

17. Comme nous l'avons vu dans le § 18, un bloc de liaison est constitué de l'ensemble des informations nécessaires au retour d'une opération. En nilNovi Objet il comprend deux informations élémentaires : l'adresse de retour et l'adresse du bloc de liaison appelant.

18. Locales aux opérations.

19. Cependant on empile sans jamais dépiler.

et d'un registre *co* : le compteur ordinal.

## 19.2 Représentation des classes

Lors de l'exécution d'un programme nilNovi Objet, une classe possède une représentation. Celle-ci permet d'attribuer un type dynamique à chaque référence et de retrouver le code des méthodes virtuelles invoquées. Une classe est constituée :

- d'une partie statique (la référence), située dans la pile d'exécution, qui désigne le début de la partie dynamique.
- d'une partie dynamique elle-même constituée :
  - d'un emplacement contenant la taille de chaque objet de cette classe,
  - pour chaque méthode virtuelle de cette classe, d'un emplacement désignant, dans le programme objet, le début du code objet de la méthode virtuelle considérée.

La partie statique possède comme adresse, dans la pile d'exécution, l'adresse statique [20] de la classe.

Un exemple est présenté dans la section suivante.

## 19.3 Représentation des objets

Lors de l'exécution d'un programme nilNovi Objet, un objet possède une représentation située dans le tas et constituée :

- d'un pointeur sur la représentation de la classe,
- de $n$ emplacements correspondants aux $n$ champs de l'objet.

Le pointeur sur la représentation de la classe assure un typage de l'objet et permet de retrouver les méthodes virtuelles quand nécessaire.

**Exemple.** Nous reprenons dans son principe l'exemple 1 de la section 16 page 59 en nous focalisant sur l'aspect représentation.

```
01 : procedure p is
02 :    type liste is class
03 :       val : integer ;
04 :       svt :liste ;
05 :    interface
06 :       constructor create(v : integer ;s :liste) ;
07 :       function valeur() return integer ;
08 :       function suivant() return liste ;
09 :    implementation
```

20. Comme pour toutes les entités globales l'adresse dynamique et l'adresse statique sont identiques.

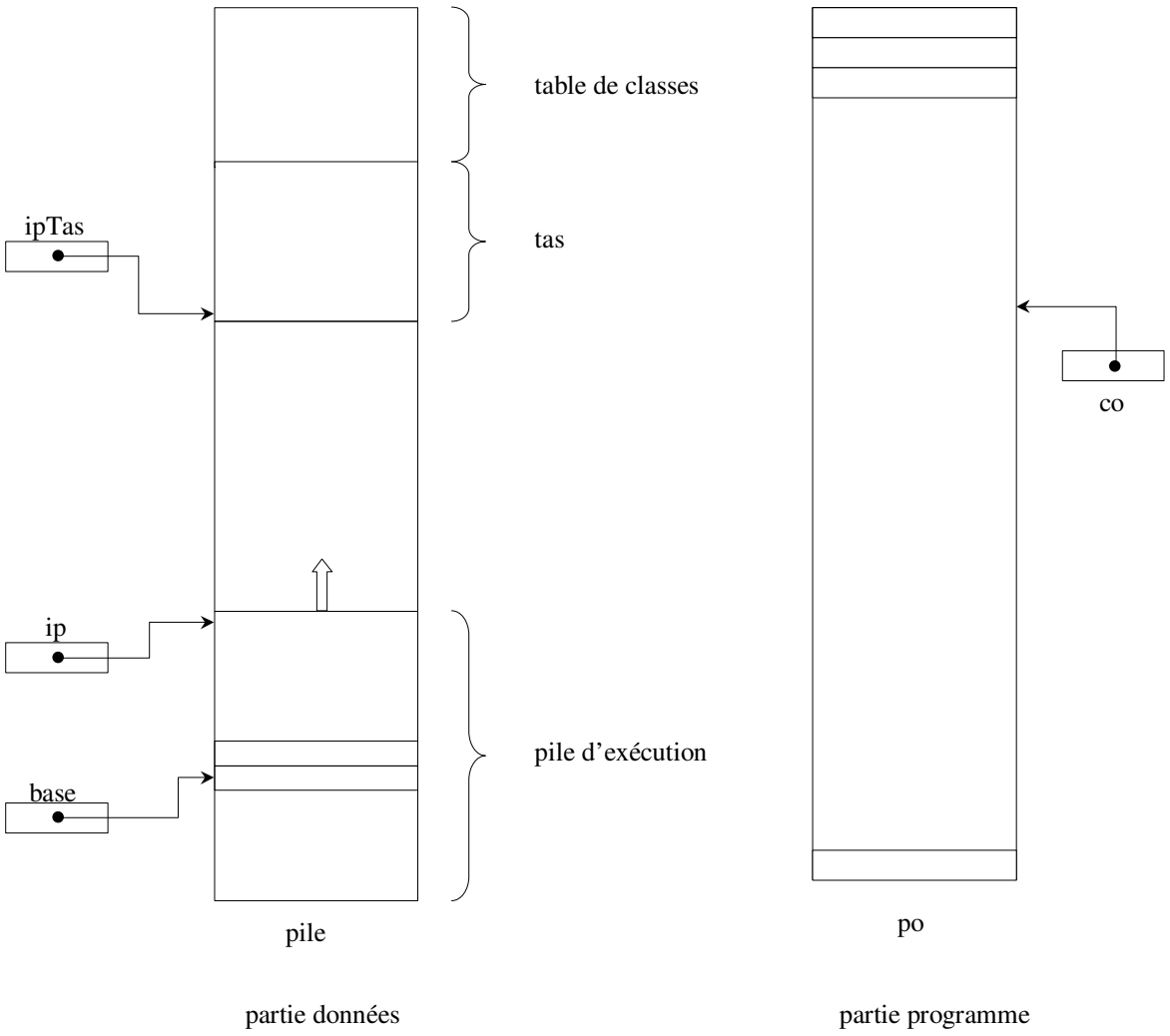



FIGURE 4 – Structure de la machine NILNOVI OBJET

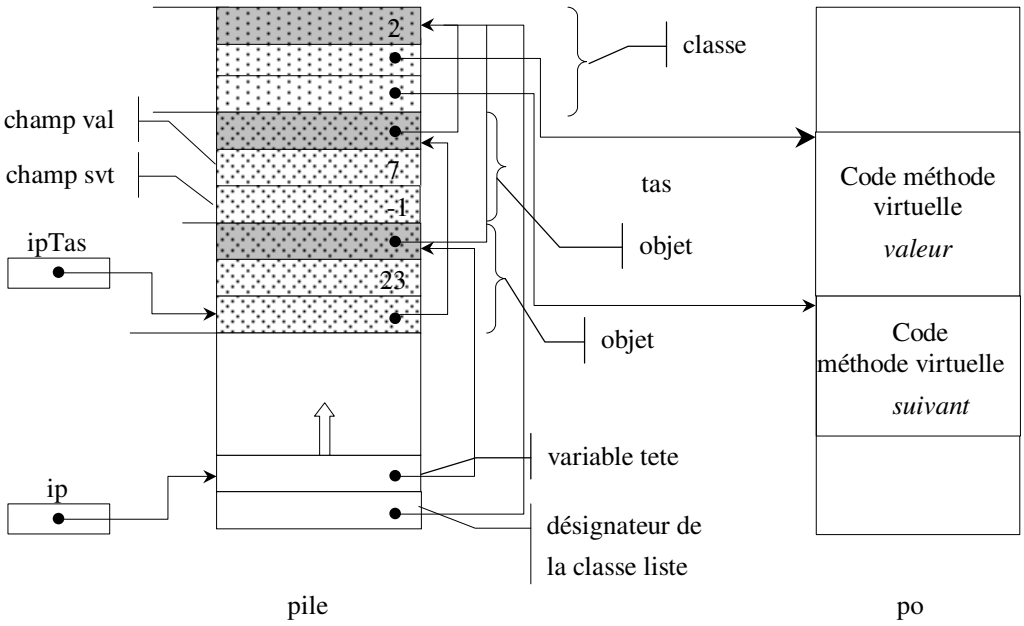



FIGURE 5 – Configuration de la machine à l'issue de l'exécution de l'exemple *liste*

```
10 :        ...
11 :     end ;
12 :     tete :liste ;
13 : begin
14 :     tete :=liste.create(23,liste.create(7,nil))
15 : end.
```

Au moment de l'exécution de la ligne 11, la classe *liste* a été créée dans la table des classes. Elle est désignée par le premier emplacement de la pile d'exécution. Après exécution de la ligne 12, l'emplacement pour la variable *tete* est réservé dans la pile d'exécution. Après exécution de l'affectation de la ligne 14, deux objets du type *liste* ont été créés. Le premier créé (celui contenant la valeur 7 dans le champ *val*) est désigné par le second, celui-ci étant lui-même désigné par la variable *tete*. L'ensemble est présenté à la figure 5.

# 20 Jeu d'instructions de la machine NILNOVI OBJET

Afin de ne pas alourdir la description, aucun contrôle (débordement de pile, division par 0, etc.) n'est spécifié. L'introduction de telles vérifications constitue un bon exercice et devra de toutes façons être réalisée dans le cadre d'un éventuel projet.

Le nombre noté entre parenthèses (par exemple (1) après *debutProg*) désigne le code machine de l'instruction. Dans la suite l'évolution du compteur ordinal n'est pas prise en compte excepté dans les instructions de contrôle. La suite de cette section spécifie les instructions de la machine nilNovi Objet sous la forme pré/post. Les instructions sont classées par famille (entrée/sortie, variables et affectation, etc.) bien que certaines d'entre elles puissent appartenir à plusieurs familles (l'instruction *tra* par exemple est utilisée pour compiler les structures de contrôle mais aussi pour compiler les classes).

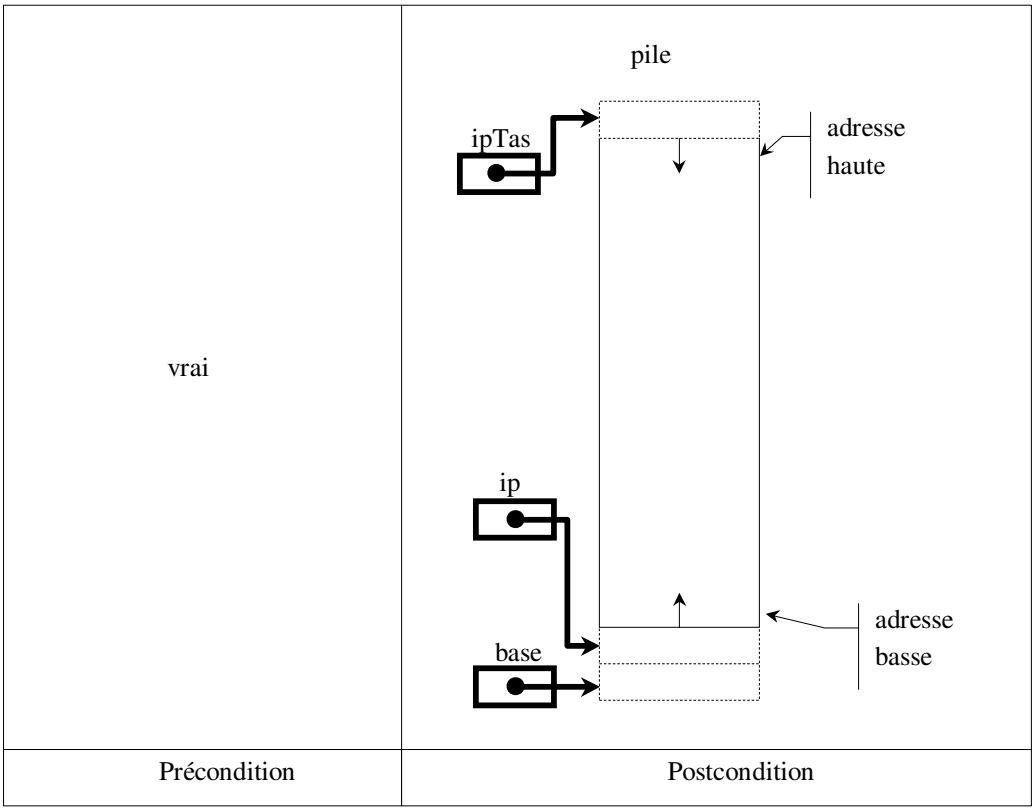
L'initialisation de la machine place l'adresse de la première instruction machine dans le compteur ordinal.

## 20.1 Unité de compilation

### 20.1.1 débutProg (1)

**procedure** debutProg() ;

Cette instruction est générée au début d'un programme nilNovi Objet. Elle permet d'initialiser la structure de données (piles, registres).




### 20.1.2 finProg (2)

**procedure** finProg() ;

Cette instruction arrête la machine nilNovi Objet.

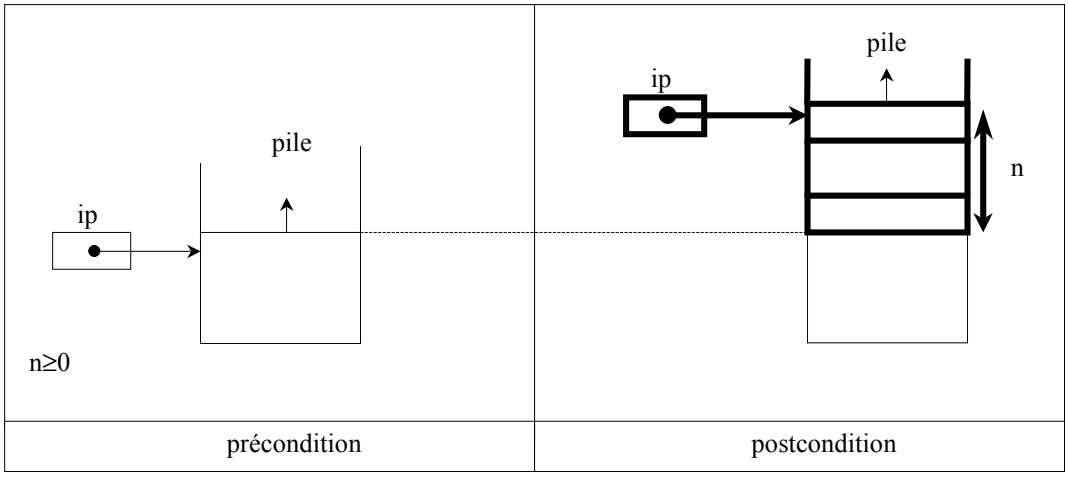
## 20.2 Variables et affectation

Nous avons vu ci-dessus que le compilateur ne peut en général attribuer une adresse absolue (adresse « dynamique ») à une variable locale et qu'il est nécessaire de passer par une adresse intermédiaire : l'adresse statique. Celle-ci représente à l'exécution un déplacement par rapport à une information contenue dans le registre *base*. Ces deux informations (valeur du registre *base* et adresse statique) permettent de calculer l'adresse dynamique à chaque accès à la variable. Concernant les entités globales (variables et classe) leur adresse dynamique est connue dès la compilation. Ce qui permet de se passer de l'étape intermédiaire dans le mécanisme de traduction.

### 20.2.1 reserver (3)

**procedure** reserver(n : integer) ;

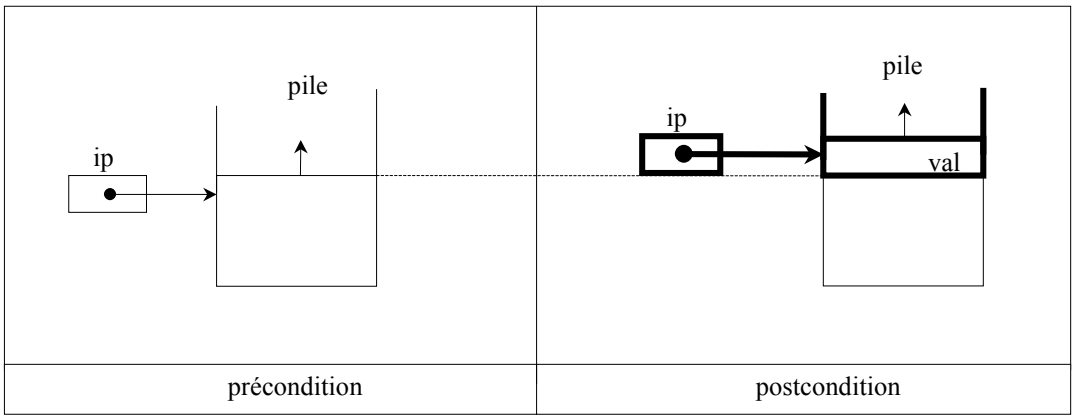
Cette instruction a pour but de réserver $n$ emplacements dans la pile d'exécution.




### 20.2.2 empiler (4)

**procedure** empiler(val : integer) ;

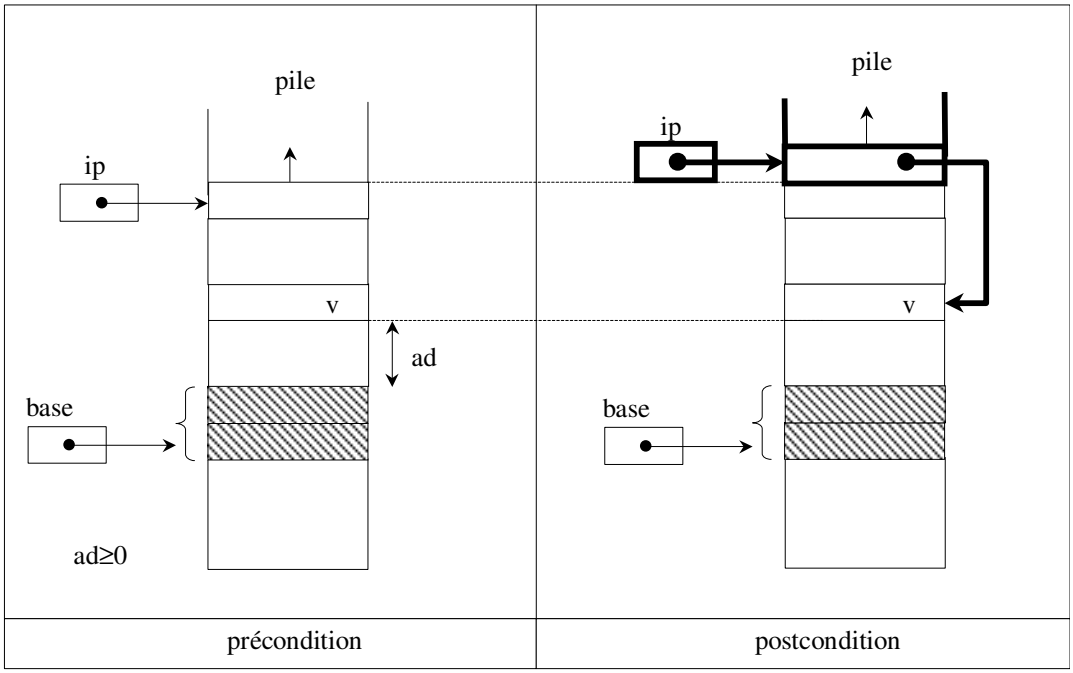
Cette instruction est utilisée pour empiler les adresses des entités globales (variables et classes) ainsi que les valeurs scalaires présentes dans une expression.




### 20.2.3 empilerAd (5)

**procedure** empilerAd(ad : integer) ;

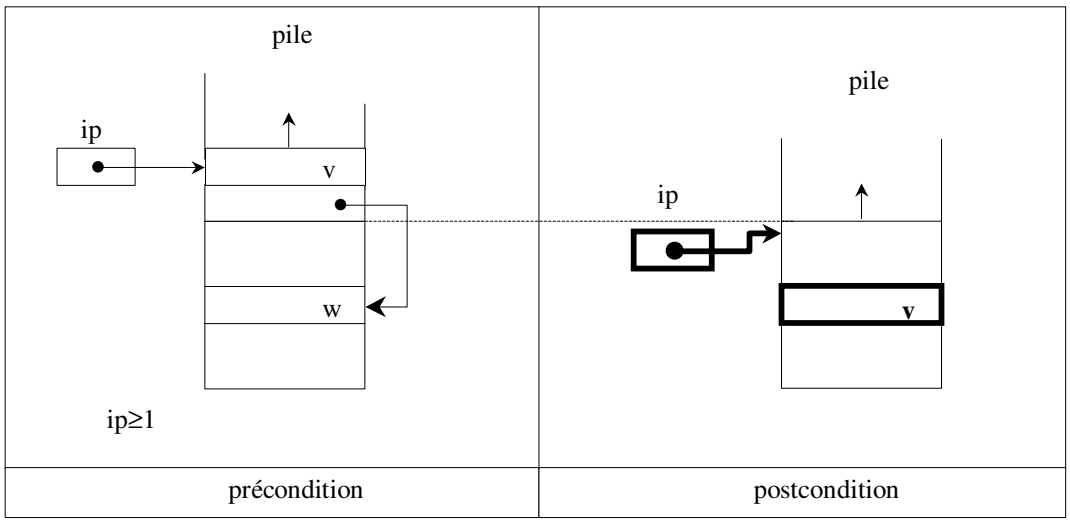
Cette instruction est utilisée dans le cas de variables locales pour transformer l'adresse statique en adresse dynamique. La valeur *ad* désigne le $ad^e$ emplacement au-dessus du bloc de liaison courant.




### 20.2.4 affectation (6)

**procedure** affectation() ;

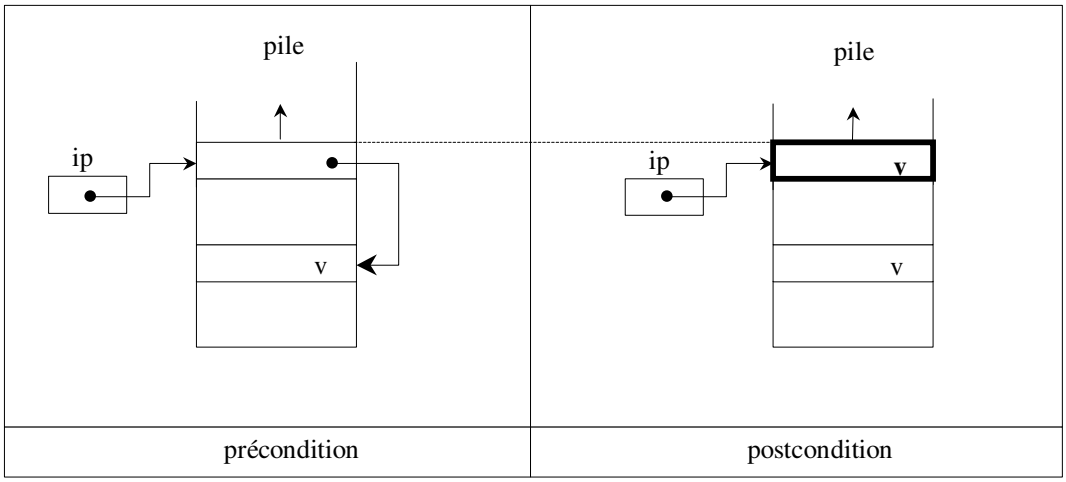
Cette instruction place la valeur située en sommet de pile à l'adresse désignée par l'emplacement sous le sommet.




### 20.2.5 valeurPile (7)

**procedure** valeurPile() ;

Cette instruction remplace le sommet de pile par le contenu de l'emplacement désigné par ce sommet.



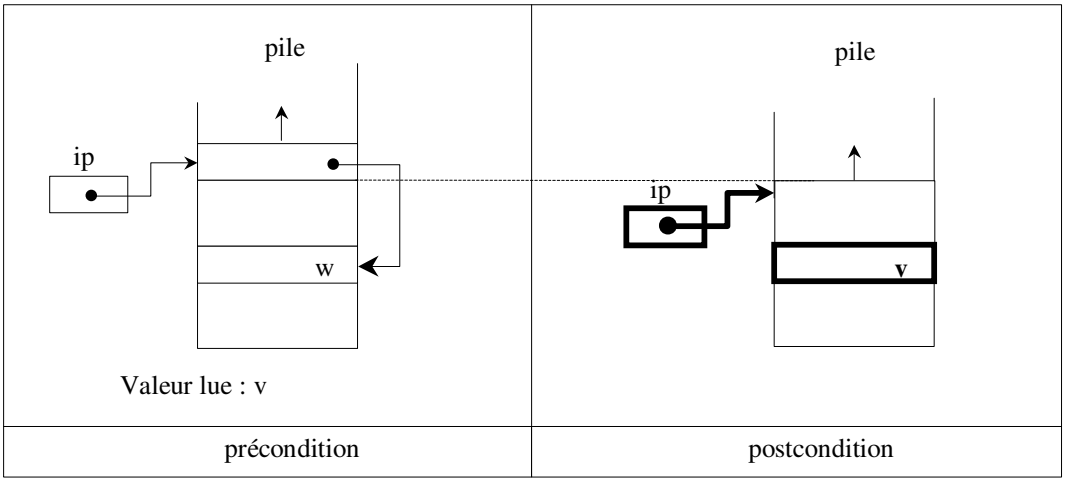

## 20.3 Entrées-Sorties

Il existe deux instructions dans cette famille : une pour les entrées (clavier) et une pour les affichages écran.

### 20.3.1 get (8)

**procedure** get() ;

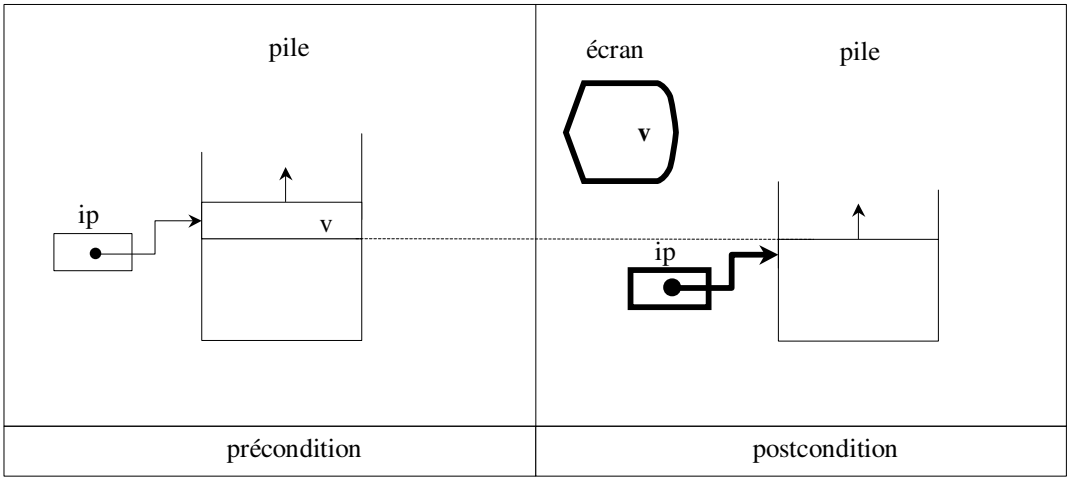
Cette instruction permet de placer la valeur lue sur le clavier dans la variable, le paramètre d'entrée-sortie effectif ou l'attribut qui est désignée par le sommet de pile.




### 20.3.2 put (9)

**procedure** put() ;

Cette instruction permet d'afficher la valeur présente au sommet de la pile.



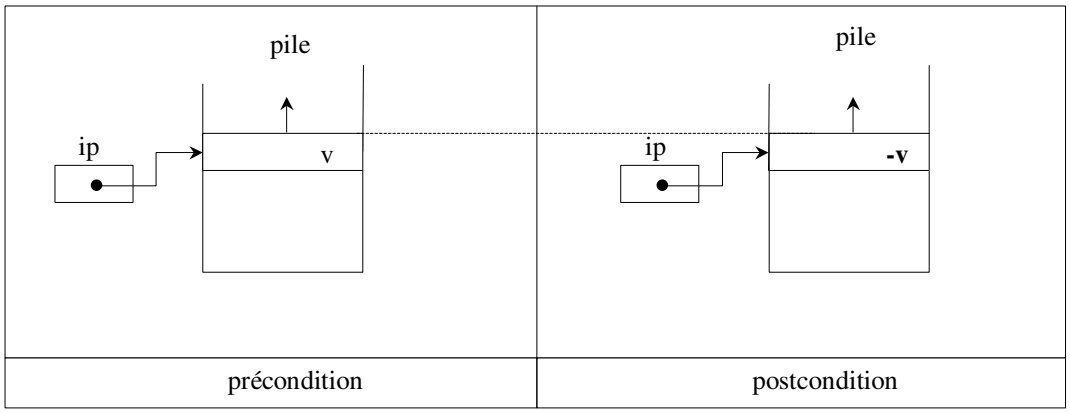

## 20.4 Expressions arithmétiques

Il existe une instruction unaire, *moins,* et quatre instructions binaires : *mult, add, sous* et *div.* Les instructions binaires opèrent sur les deux éléments en sommet de pile.

### 20.4.1 moins (10)

**procedure** moins() ;

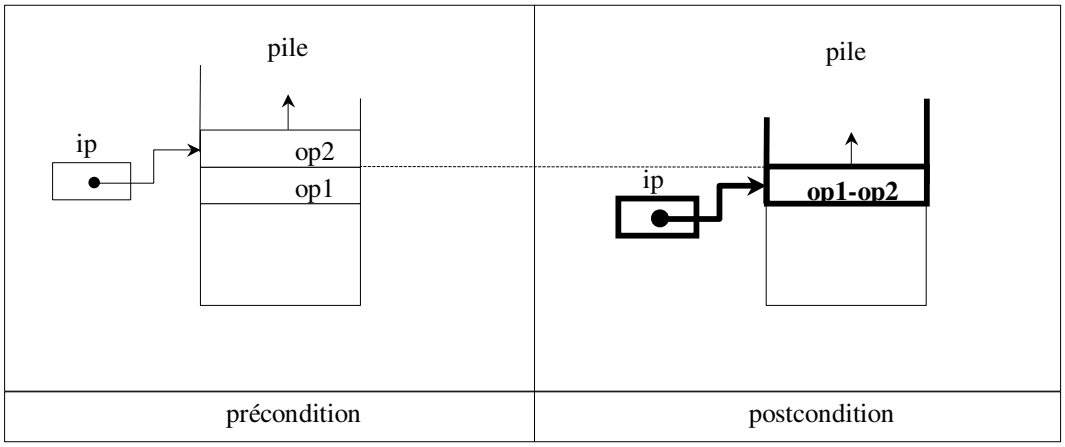
Cette instruction permet de calculer l'opposé de la valeur entière en sommet de pile.




### 20.4.2 sous (11)

**procedure** sous() ;

Cette instruction permet de calculer la différence entre les deux valeurs au sommet de pile.




### 20.4.3 add (12), mult (13), div(14)

**procedure** add() ;
**procedure** mult() ;
**procedure** div() ;

Ces instructions sont similaires à l'instruction *sous* pour les opérations d'addition, de multiplication et de division. Leur sémantique pré/post n'est pas présentée ici.

## 20.5 Expressions relationnelles et booléennes

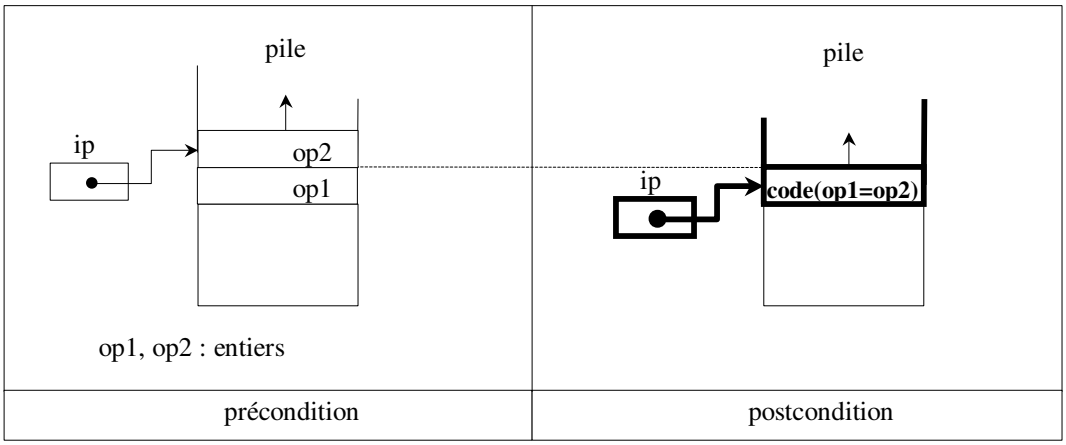
Il existe six instructions relationnelles : *egal, diff, inf, infeg, sup* et *supeg.* Elles correspondent respectivement aux opérateurs $=, / =, <, <=, >, >=$. Leurs opérandes sont entiers. Il existe deux instructions booléennes binaires : *et* et *ou.* Elles correspondent aux opérations *and* et *or.* On rappelle ici que ces opérations n'ont pas une sémantique « court-circuit » : elles évaluent *toujours* leurs deux opérandes. Il existe une instruction booléenne unaire : *non,* qui correspond à l'opérateur *not.*

Les constantes booléennes *vrai* et *faux* sont respectivement codées par les entiers 1 et 0.

### 20.5.1 egal (15)

Cette instruction compare deux entiers et empile le code de l'expression booléenne $op1 = op2$.

**procedure** egal() ;




### 20.5.2 diff (16), inf (17), infeg(18), sup (19), supeg (20)

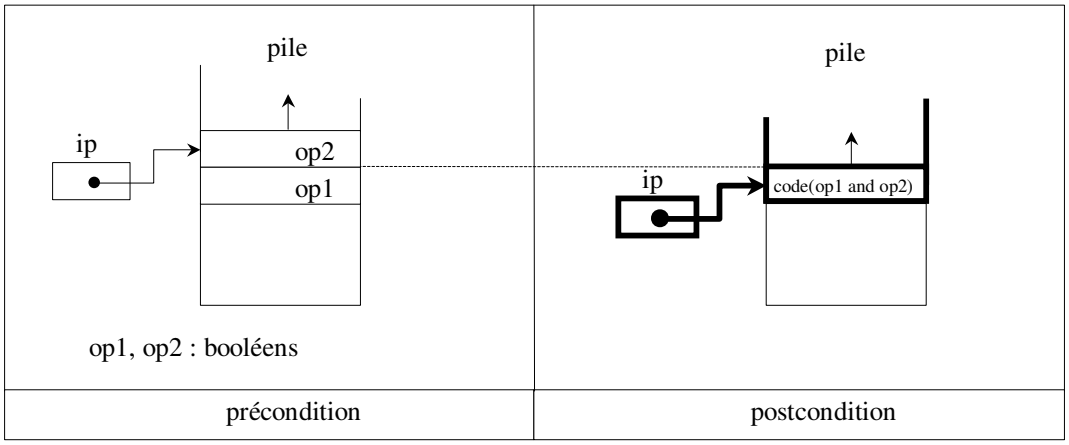
Ces instructions sont similaires à l'instruction *egal* pour les opérations $/ =, <, <=, >, >=$ . Leur sémantique pré/post n'est pas présentée ici.

**procedure** diff() ;
**procedure** inf() ;
**procedure** infeg() ;
**procedure** sup() ;
**procedure** supeg() ;

### 20.5.3 et (21)

Cette instruction prend en compte deux booléens *op1* et *op2* et empile le code de l'expression booléenne *(op1 and op2)*.

**procedure** et() ;




### 20.5.4 ou (22)

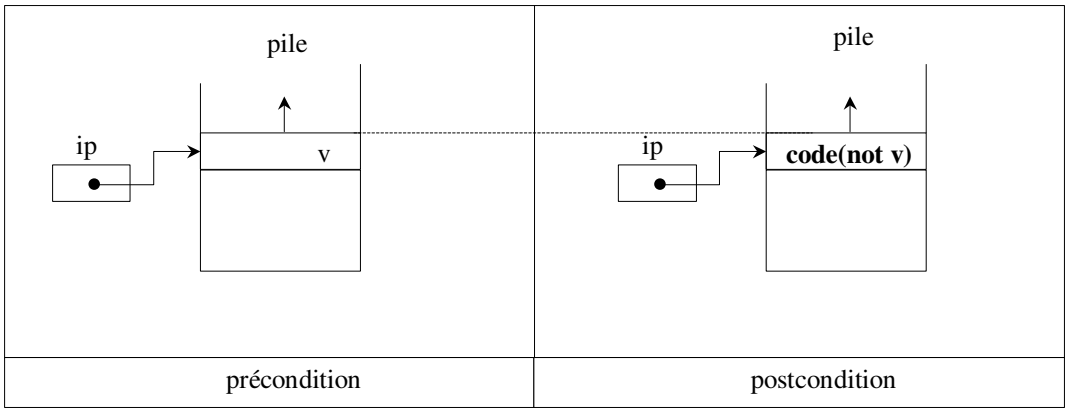
Cette instruction prend en compte deux booléens *op*1 et *op*2 et empile le code de l'expression booléenne *(op1 ou op2)*. Cette instruction est similaire à l'instruction *et*. Sa sémantique pré/post n'est pas présentée ici.

**procedure** ou() ;

### 20.5.5 non (23)

Cette instruction permet de calculer la négation du booléen en sommet de pile.

**procedure** non() ;



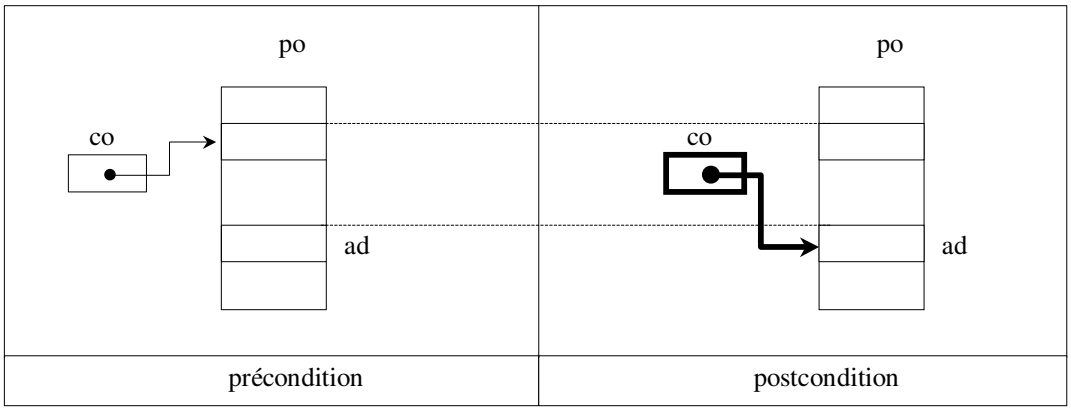
## 20.6 Contrôle

Nous étudions ici trois instructions de contrôle : *tra, tze, erreur*. Ces opérations agissent sur la progression du compteur ordinal. Deux autres instructions de contrôle, spécifiques des méthodes (*traVirt* et *traConstr*) seront étudiées plus tard.

### 20.6.1 tra (24)

**procedure** tra(ad : integer) ;

Cette instruction donne le contrôle à l'instruction située à l'adresse *ad*. Il s'agit d'un branchement inconditionnel.

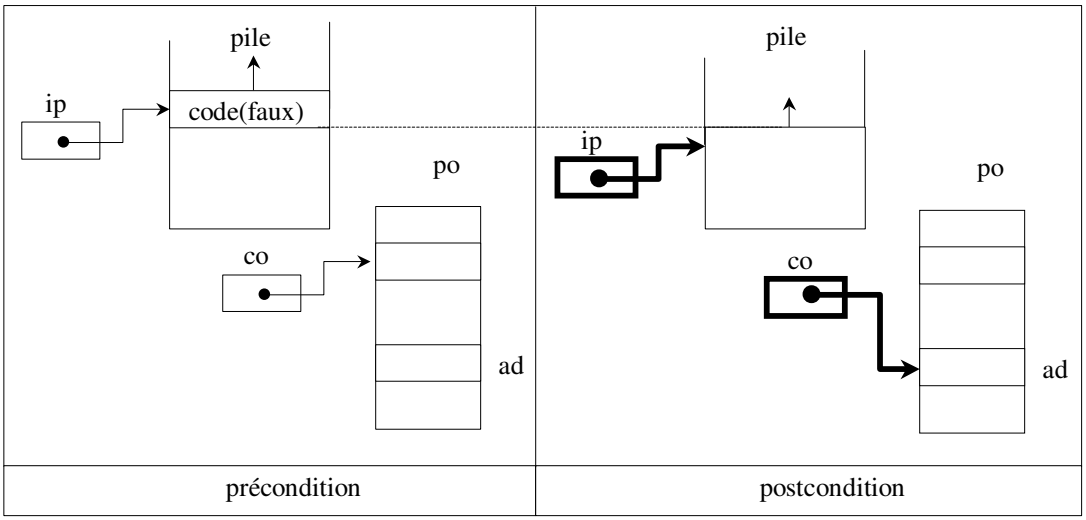



### 20.6.2 tze (25)

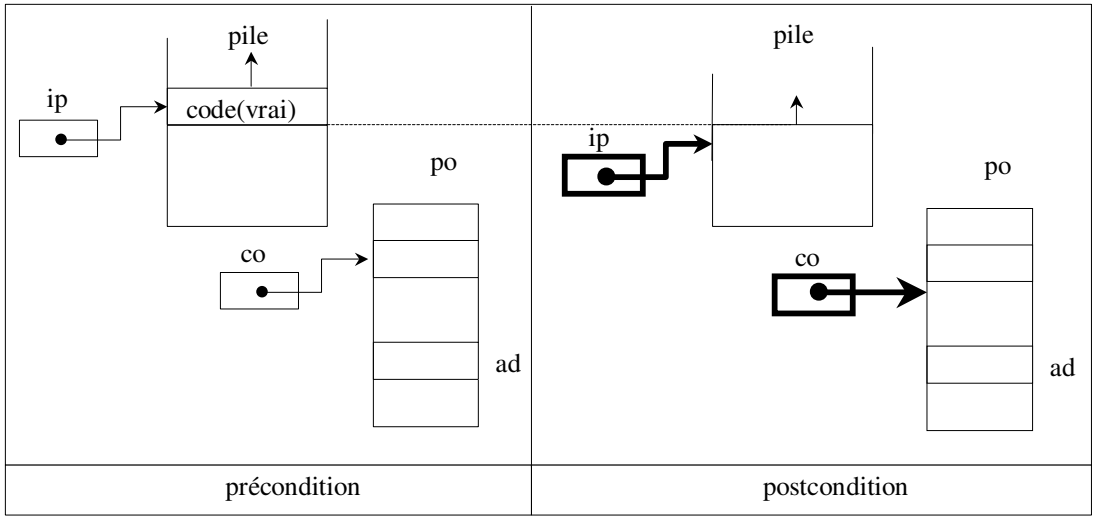
**procedure** tze(ad : integer) ;

Cette instruction donne le contrôle à l'instruction située à l'adresse *ad* si le sommet de pile contient *faux,* continue en séquence sinon.

**Premier cas :** 0 (*faux*) en sommet de pile.




**Second cas :** 1 (*vrai*) en sommet de pile.




### 20.6.3 erreur (26)

**procedure** erreur() ;

Cette instruction machine est compilée à la rencontre de l'instruction nilNovi Objet *error(exp)*. Elle affiche la valeur de l'expression *exp* passée en paramètre et arrête la machine nilNovi Objet.

## 20.7 Classes et objets

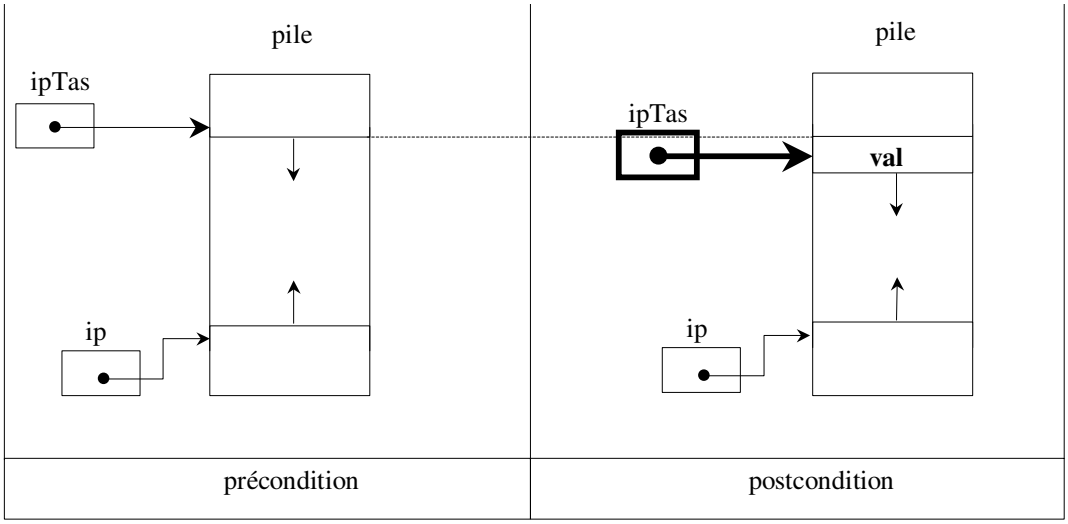
Cette famille est constituée de trois instructions *empilerTas, empilerIpTas* et *empilerAdAt*. Elles permettent la compilation et l'exécution du code source des classes (hors

opérations) et l'utilisation des objets.

### 20.7.1 empilerTas (27)

**procedure** empilerTas(val : integer) ;

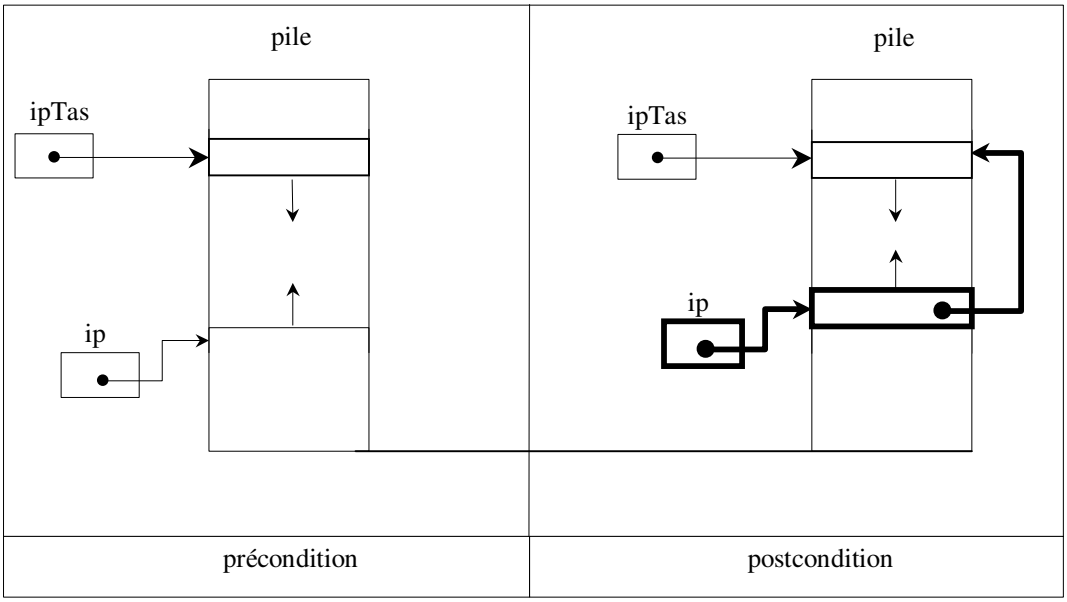
Cette instruction est l'homologue pour le tas de l'instruction *empiler* pour la pile. Elle permet d'empiler une valeur entière sur le tas.




### 20.7.2 empilerIpTas (28)

**procedure** empilerIpTas() ;

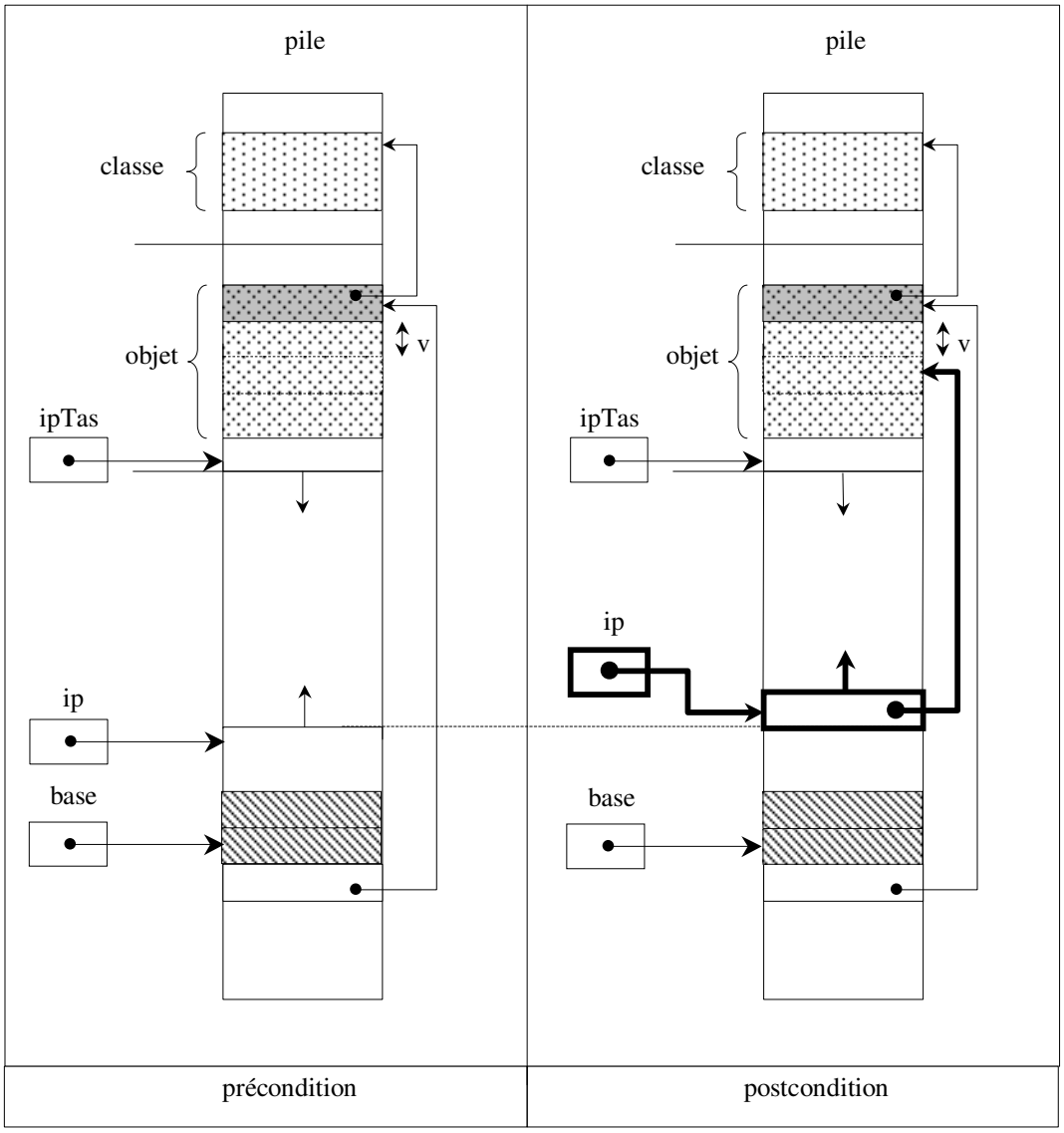
Cette instruction permet de créer, dans un emplacement de la pile d'exécution, un pointeur sur la représentation de la classe en cours d'élaboration. Ce pointeur désigne le début (adresse haute) de la représentation de la classe dans la zone de la mémoire consacrée à la table des classes.




### 20.7.3 empilerAdAt (29)

**procedure** empilerAdAt(v : integer) ;

Cette instruction permet, à partir de l'adresse statique d'un champ de l'objet, d'empiler l'adresse dynamique de l'attribut correspondant. Cette instruction est générée lors de la compilation du champ d'un objet.



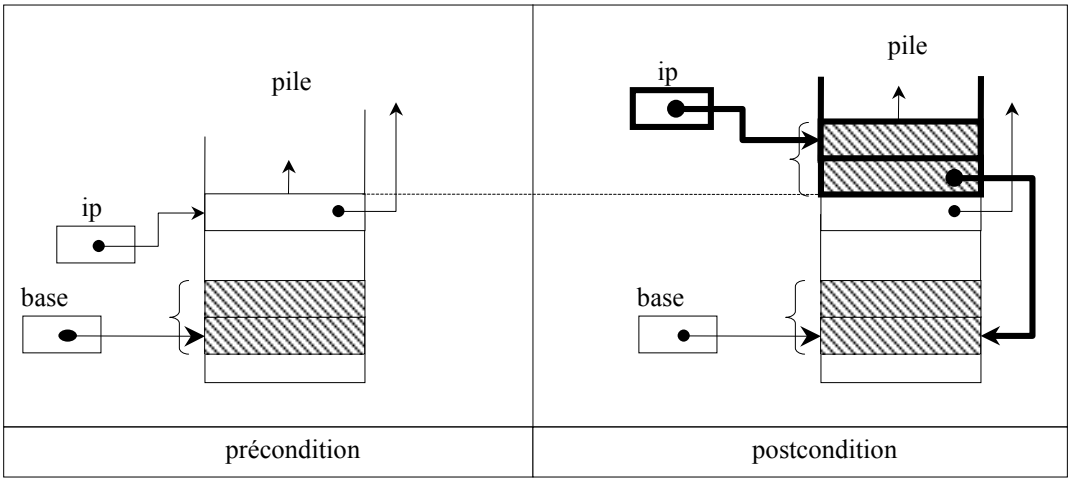

## 20.8 Opérations

Cette famille regroupe les instructions spécifiques à la compilation des méthodes.

### 20.8.1 reserverBloc (30)

**procedure** reserverBloc() ;

Cette instruction permet, lors de l'appel d'une opération, de réserver les emplacements du futur bloc de liaison et d'initialiser la cellule « pointeur sur le bloc de liaison de l'appelant ».



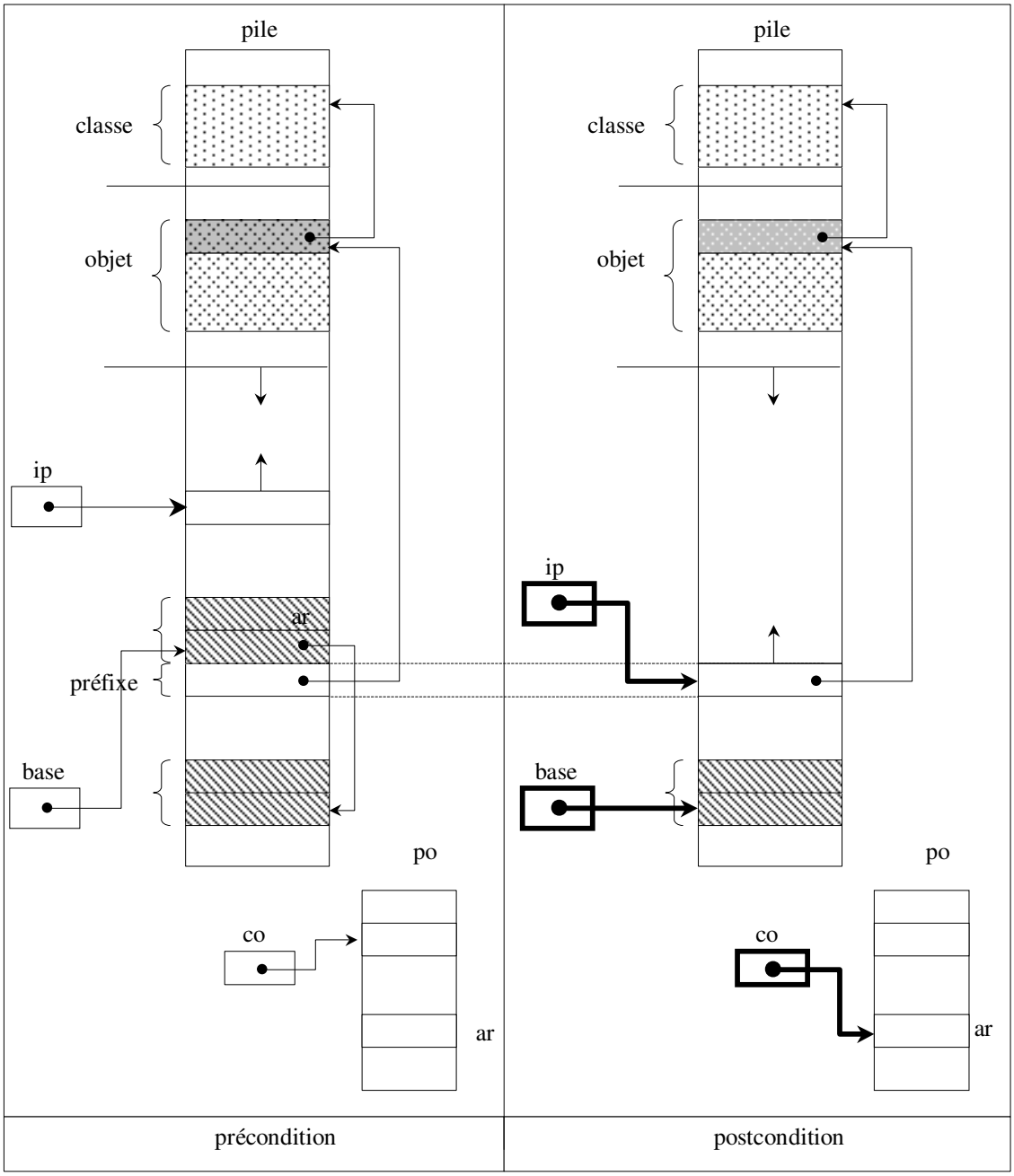

En précondition on trouve au sommet de pile le préfixe, qui désigne l'objet sur lequel va s'appliquer l'opération (ou qui désigne une classe dans le cas d'un constructeur). On note que l'opération ne provoque pas de changement de bloc de liaison [21].

### 20.8.2 retourConstr (31)

**procedure** retourConstr() ;

Cette instruction est produite à la fin de la compilation d'un constructeur. Comme un constructeur est une fonction (avec effet de bord), cette instruction assure qu'une référence à l'objet créé sera bien délivré en fin d'exécution du constructeur.

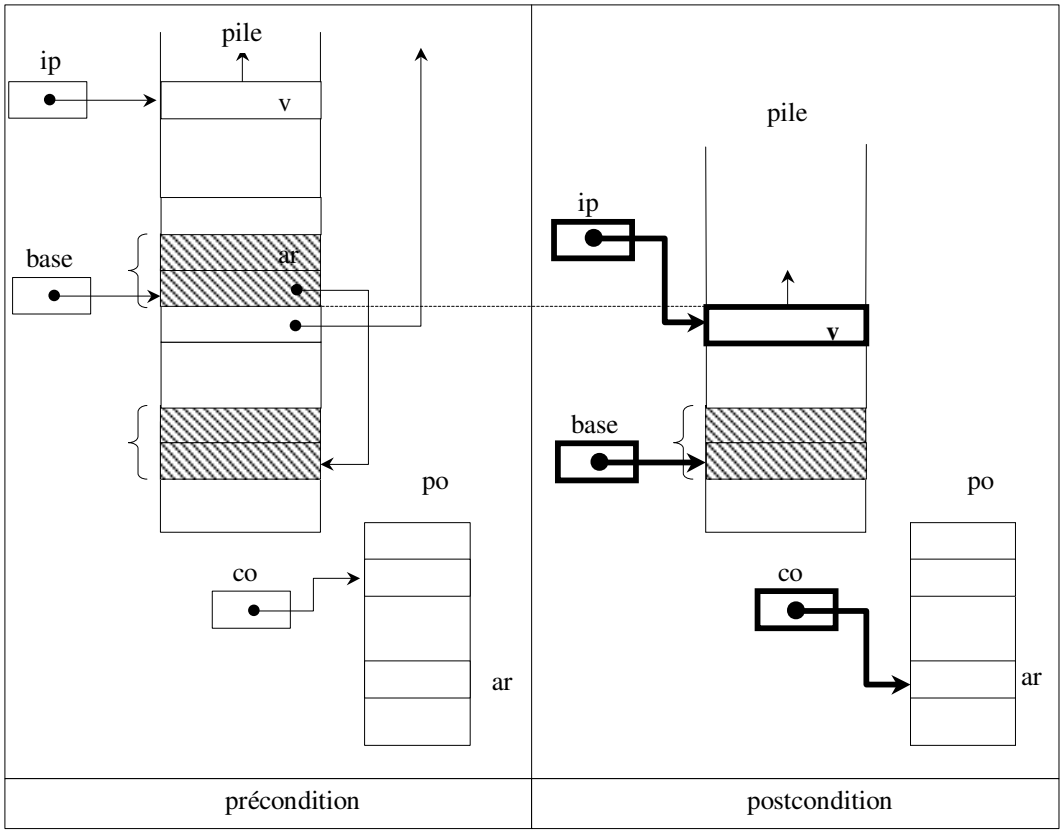
21. En effet il faut pouvoir utiliser le bloc courant pour la création des paramètres effectifs ainsi que pour l'imbrication d'appels de fonctions.




### 20.8.3 retourFonct (32)

**procedure** retourFonct() ;

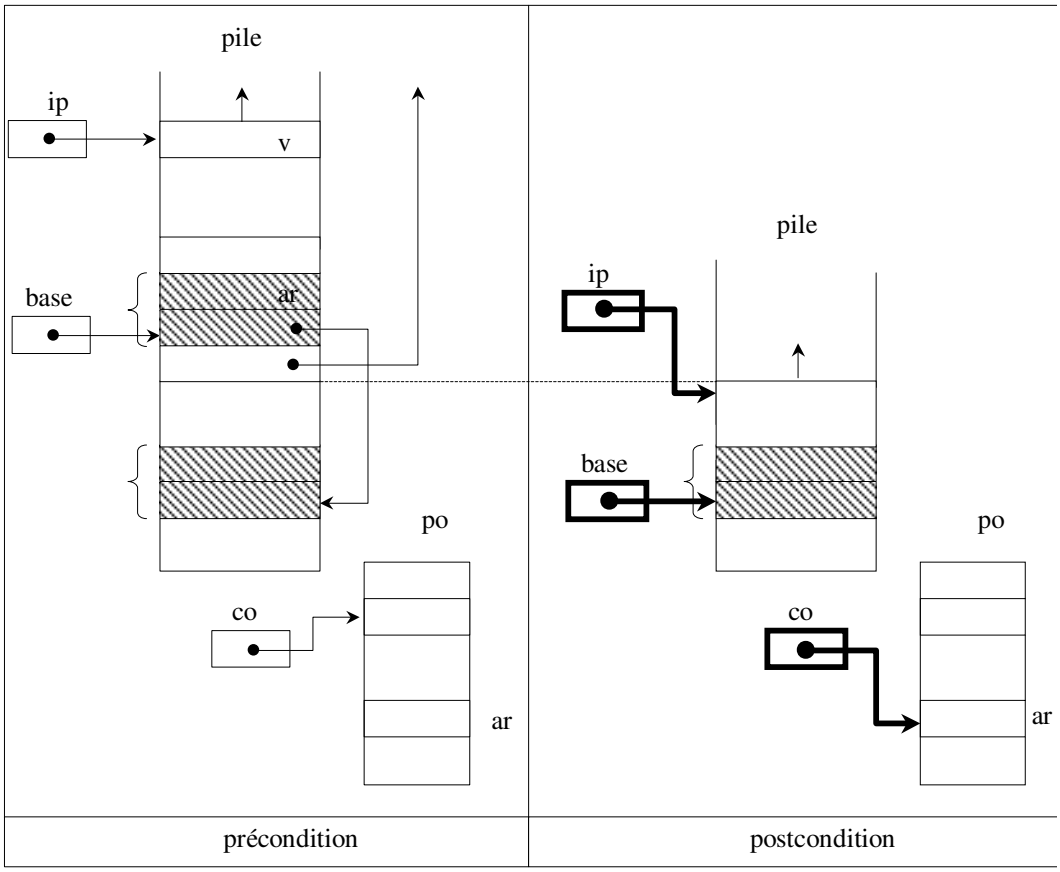
Cette instruction est produite à la fin de la compilation d'une instruction *return* dans une méthode-fonction. Outre son rôle dans le retour, elle assure que la valeur en sommet de pile sera le résultat de l'appel.




### 20.8.4 retourProc (33)

**procedure** retourProc() ;

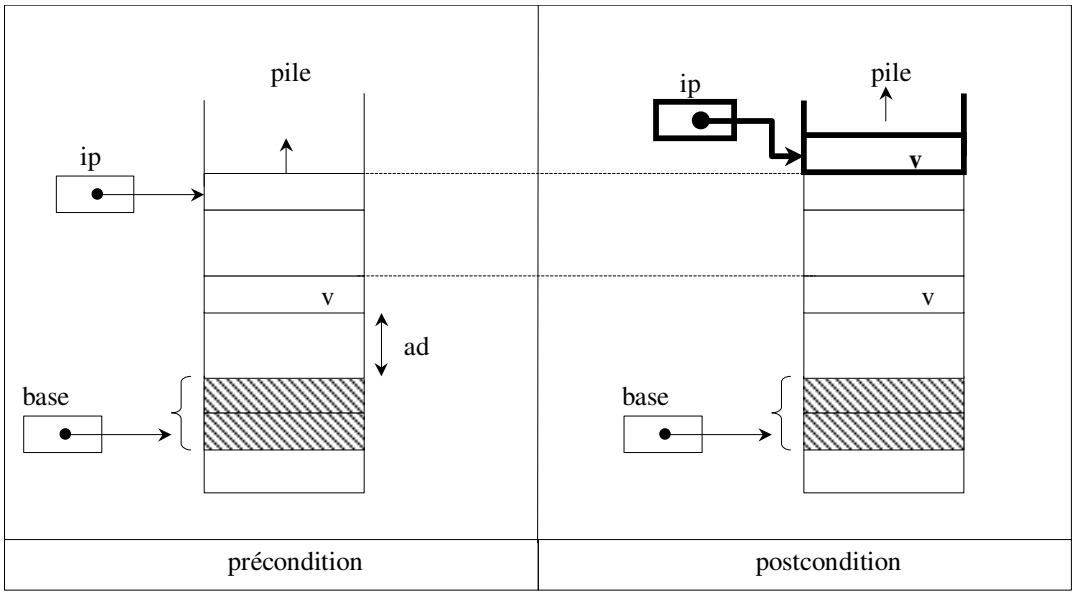
Cette instruction est produite à la fin de la compilation d'une méthode-procédure. Elle assure le retour à l'appelant.




### 20.8.5 empilerParam (34)

**procedure** empilerParam(ad : integer) ;

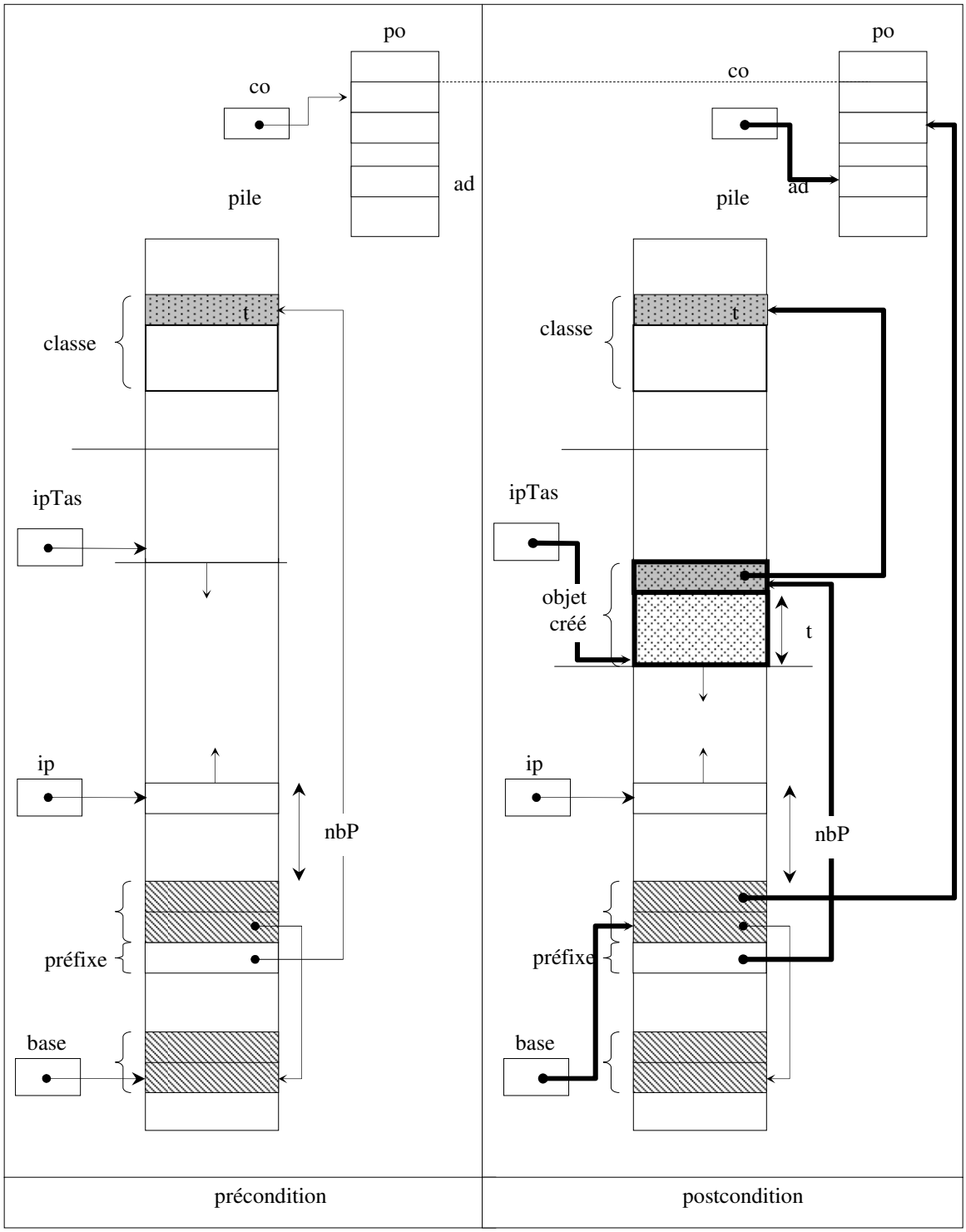
Cette instruction permet de gérer les paramètres effectifs.




### 20.8.6 traConstr (35)

**procedure** traConstr(ad : integer, nbP : integer) ;

*ad* est l'adresse effective où débute le code objet du constructeur, *nbP* est son nombre de paramètres. Cette dernière valeur permet de retrouver le bloc de liaison correspondant à l'appel.



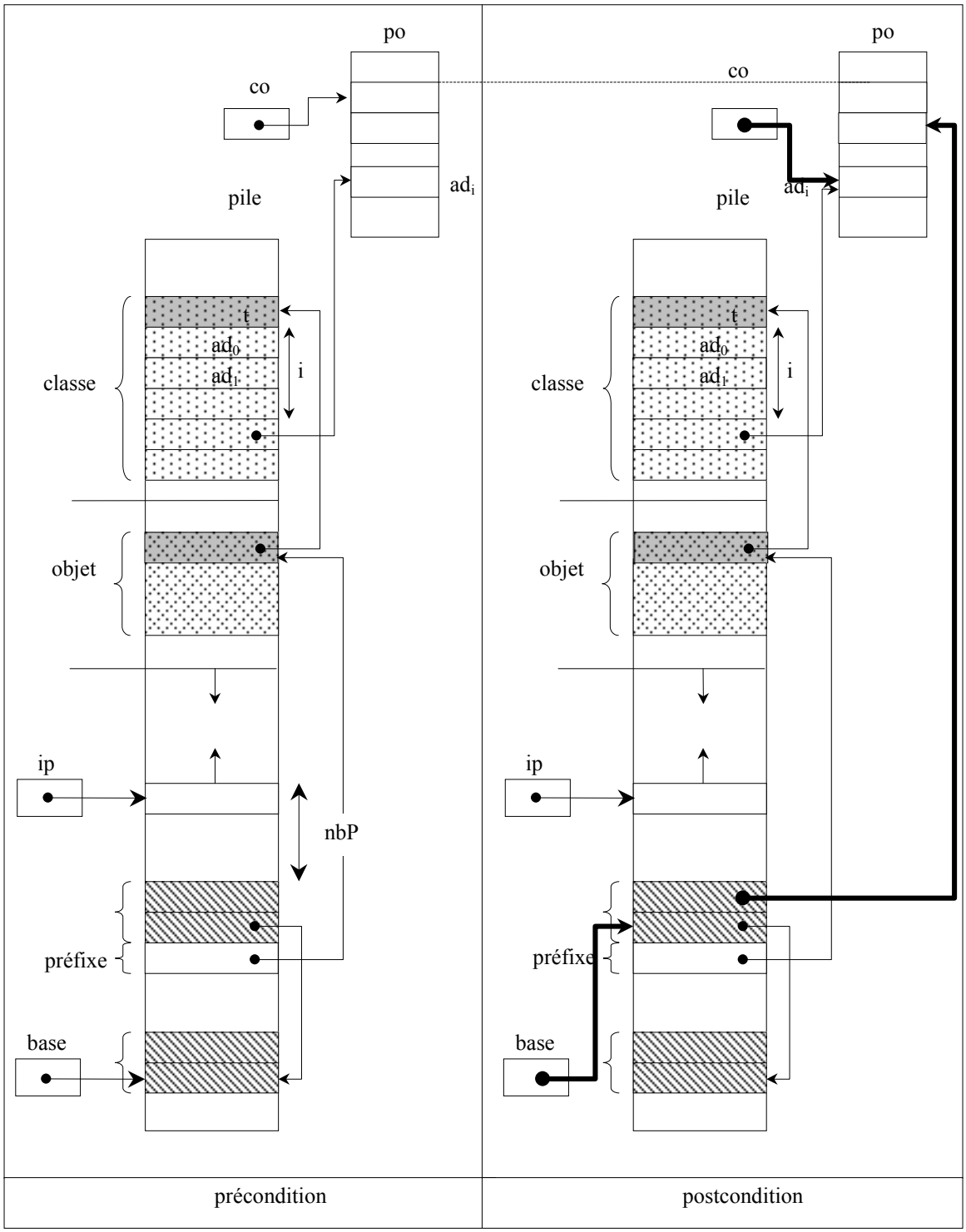

Cette instruction :
— complète la cellule « adresse de retour » en y plaçant l'adresse de l'instruction qui suit *traConstr*,
— érige le bloc de liaison ainsi complété en bloc de liaison courant,

— crée dans le tas un objet du type de la classe désigné par la cellule sous le bloc de liaison.
— fait désigner l'objet créé par la cellule sous le bloc de liaison,
— affecte le compteur ordinal avec la valeur *ad* de sorte que la prochaine instruction exécutée sera la première du constructeur.

### 20.8.7 traVirt (36)

**procedure** traVirt(i : integer, nbP : integer) ;

Cette instruction effectue l'appel effectif à une méthode virtuelle. $i$ est l'adresse statique de la méthode, et $nbP$ est son nombre de paramètres. Cette dernière valeur permet de retrouver le bloc de liaison correspondant à l'appel, et donc indirectement la table des méthodes virtuelles.


po
co
pile
$ad_i$
t
$ad_0$
$ad_i$
i
classe
objet
ip
nbP
préfixe
base
précondition
po
co
pile
$ad_i$
t
$ad_0$
$ad_i$
i
classe
objet
ip
préfixe
base
postcondition

# 21 Schéma de compilation et d'exécution

## 21.1 Introduction

Nous présentons ici la compilation des différentes constructions d'un programme NIL-Novi Objet. La compilation y est vue comme un processus inductif : on distingue donc les cas de base (déclaration de variables, expressions réduites à une constante, une variable, un paramètre, un attribut, ordre de lecture) et les cas inductifs, pour lesquels la compilation d'une structure $A$ constituée des sous-structures $X$ et $Y$ est composée de la compilation de $X$ et de $Y$, ce que l'on formalisera de la manière suivante, en notant $\{P\}$ la compilation d'un programme source $P$ :

$$\{A\} = \Phi(\{X\}, \{Y\})$$

Dans la suite $as(< V >)$ signifie « adresse statique de l'identificateur $< V >$ ».

## 21.2 Unité principale

Soit $< C >$ la déclaration des classes et $< V >$ la déclaration des variables.

$$\left\{\begin{array}{l}\textbf{procedure}\ \text{p}\ \ \textbf{is}\\ \quad < \texttt{C} >\\ \quad < \texttt{V} >\\ \textbf{begin}\\ \quad < \texttt{B} >\\ \textbf{end}.\end{array}\right\} = \begin{array}{l}\texttt{debutProg}();\\ \{< \texttt{C} >\};\\ \{< \texttt{V} >\};\\ \{< \texttt{B} >\};\\ \texttt{finProg}()\end{array}$$

## 21.3 Ordre d'écriture

$$\{\text{put}(< \texttt{E} >)\} = \begin{array}{l}\{< \texttt{E} >\};\\ \texttt{put}()\end{array}$$

On rappelle que le paramètre $< \texttt{E} >$ est une expression entière.

## 21.4 Variables

Nous considérons ici que l'on sait compiler toute expression. Cette lacune sera comblée dans la section suivante.

### 21.4.1 Déclaration

On considère la déclaration successive de $n$ variables.

$$\{\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}\ :< \texttt{type} >\} = \texttt{reserver(n)}$$

### 21.4.2 Affectation

Le schéma de compilation suivant de l'affectation concerne les variables locales (scalaire ou objet). $< E >$ est une expression.

$$\{< \texttt{V} >:=< \texttt{E} >\} = \begin{array}{l} \texttt{empilerAd}(\texttt{as}(< \texttt{V} >)); \\ \{< \texttt{E} >\}; \\ \texttt{affectation}() \end{array}$$

Le schéma suivant s'applique au cas des variables globales :

$$\{< \texttt{V} >:=< \texttt{E} >\} = \begin{array}{l} \texttt{empiler}(\texttt{as}(< \texttt{V} >)); \\ \{< \texttt{E} >\}; \\ \texttt{affectation}() \end{array}$$

## 21.5 Expressions

### 21.5.1 Expressions arithmétiques

Quatre cas de base sont développés. Par induction structurelle, et en prenant en considération la priorité des opérateurs et le parenthèsage, il sera possible d'appliquer la démarche à toute expression arithmétique.

#### Expression réduite à une constante entière

$$\{< \texttt{C} >\} = \texttt{empiler}(< \texttt{C} >)$$

#### Expression avec un opérateur binaire

$$\{< \texttt{E1} > \textbf{op} < \texttt{E2} >\} = \begin{array}{l} \{< \texttt{E1} >\}; \\ \{< \texttt{E2} >\}; \\ \texttt{op}() \end{array}$$

où *op* représente l'un des opérateurs arithmétiques $+$, $-$, $*$, $/$ et *op()* est l'instruction correspondante (*add(), sous(), mult(), div()*) de la machine nilNovi Objet.

#### Expression avec l'opérateur moins unaire

$$\{- < \texttt{E} >\} = \begin{array}{l} \{< \texttt{E} >\}; \\ \texttt{moins}() \end{array}$$

**Expressions réduites à une variable**

Les expressions réduites à une variable locale se compilent de la manière suivante :

$$\{< \texttt{V} >\} = \begin{array}{l}\texttt{empilerAd}(\texttt{as}(< \texttt{V} >));\\ \texttt{valeurPile}()\end{array}$$

Les expressions réduites aux variables globales se compilent selon le schéma :

$$\{< \texttt{V} >\} = \begin{array}{l}\texttt{empiler}(\texttt{as}(< \texttt{V} >));\\ \texttt{valeurPile}()\end{array}$$

Pour être complet il nous restera à étudier le cas des expressions réduites au paramètre d'une opération ou à l'attribut d'un objet et le cas des expressions constituées par l'appel d'une fonction.

### 21.5.2 Expressions booléennes

Trois cas sont considérés : les constantes, les opérateurs binaires et l'opérateur *not.* Ensuite, à l'instar des expressions arithmétiques, par induction structurelle, et en prenant en considération la priorité des opérateurs et le parenthèsage, il sera possible d'appliquer la démarche à toute expression booléenne.

**Expression réduite à une constante booléenne**

On rappelle que *vrai* se code 1 et *faux* 0.

$$\{\text{true}\} = \texttt{empiler}(\texttt{1})$$

$$\{\text{false}\} = \texttt{empiler}(\texttt{0})$$

**Expression avec un opérateur booléen binaire**

La démarche s'applique aussi bien aux expressions booléennes pures qu'aux expressions relationnelles.

$$\{< \texttt{E1} > \textbf{ op } < \texttt{E2} >\} = \begin{array}{l}\{< \texttt{E1} >\};\\ \{< \texttt{E2} >\};\\ \texttt{op}()\end{array}$$

où *op* représente l'un des opérateurs booléens **and, or** ou relationnel =, / =, <, <=, >, >= et *op()* est l'instruction correspondante (*et(), ou(), egal(), diff(), inf(), infeg(), sup(), supeg()*) de la machine nilNovi Objet.

**Expression avec un opérateur** *not*

$$\boxed{\{\textbf{not} < \texttt{E} >\} = \begin{array}{l}\{< \texttt{E} >\};\\ \texttt{non}()\end{array}}$$

### 21.5.3 Expressions d'objets

Quatre cas sont à considérer : la constante *nil* (qui dénote l'absence d'objet), le pronom *this* (qui ne peut être utilisé que dans le contexte d'une classe et qui dénote l'objet courant), l'invocation d'un constructeur, et celle d'une fonction délivrant un objet. Ces deux derniers cas seront étudiés plus loin.

**Expression réduite à la constante *nil***

Par convention *nil* est représenté par la valeur -1.

$$\boxed{\{\textbf{nil}\} = \texttt{empiler}(-1)}$$

**Expression réduite au pronom *this***

Le pronom *this* ne peut être utilisé que dans le corps d'une opération, il fait référence à l'objet en cours de traitement. Cet objet est toujours désigné par l'emplacement situé sous le bloc de liaison courant. D'où :

$$\boxed{\{\textbf{this}\} = \begin{array}{l}\texttt{empilerAd}(-3);\\ \texttt{valeurPile}()\end{array}}$$

Pour être complet avec les expressions il nous restera à étudier le cas des expressions constituées d'un paramètre (d'une opération), le cas des expressions constituées par un appel à une fonction et le cas des expressions constituées par le champ d'un objet. Cette lacune sera comblée lors de l'étude de la compilation des classes.

## 21.6 Séquentialité

La composition séquentielle se traduit trivialement de la manière suivante :

$$\boxed{\{< \text{S1} >; < \text{S2} >\} = \{< \text{S1} >\}; \{< \text{S2} >\}}$$

## 21.7 Ordre de lecture

L'ordre de lecture *get* permet de lire une variable, un paramètre effectif d'entrée-sortie ou un attribut depuis le clavier. *get* ne permet de lire que des valeurs entières. Dans le cas de variables locales nous avons :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empilerAd}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

Dans le cas de variables globales nous avons :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empiler}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

Dans le cas de paramètres formels d'entrée-sortie (cf. § 21.11) nous avons :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empilerParam}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

Et dans le cas d'attributs (cf. § 21.11) :

$$\{\text{get}(< \texttt{V} >)\} = \begin{array}{l}\texttt{empilerAdAt}(\texttt{as}(< \texttt{V} >));\\ \texttt{get}()\end{array}$$

## 21.8 Instruction d'erreur

L'instruction d'erreur *error* permet d'afficher l'expression entière passée en paramètre puis d'arrêter le programme en cours d'exécution.

$$\{\text{error}(< \texttt{E} >)\} = \begin{array}{l}\{< \texttt{E} >\};\\ \texttt{erreur}()\end{array}$$

## 21.9 Alternatives

Deux cas d'alternatives sont à considérer : les alternatives simples et les alternatives doubles.

### Alternatives simples

$$\left\{\begin{array}{l}\textbf{if} < \texttt{C} > \textbf{ then}\\ \quad < \texttt{A} >\\ \textbf{end}\end{array}\right\} = \begin{array}{l}\{< \texttt{C} >\};\\ \texttt{tze}(\texttt{ad});\\ \{< \texttt{A} >\};\\ \texttt{ad} : \ldots\end{array}$$

**Alternatives doubles**

$$\left\{\begin{array}{l}\textbf{if} < \texttt{C} > \textbf{ then}\\ \quad < \texttt{A} >\\ \textbf{else}\\ \quad < \texttt{B} >\\ \textbf{end}\end{array}\right\} = \begin{array}{l}\{< \texttt{C} >\};\\ \texttt{tze(ad1)};\\ \{< \texttt{A} >\};\\ \texttt{tra(ad2)};\\ \texttt{ad1} : \{< \texttt{B} >\};\\ \texttt{ad2} : \ldots\end{array}$$

## 21.10 Boucle

$$\left\{\begin{array}{l}\textbf{while} < \texttt{C} > \textbf{ loop}\\ \quad < \texttt{A} >\\ \textbf{end}\end{array}\right\} = \begin{array}{l}\texttt{ad1} : \{< \texttt{C} >\};\\ \texttt{tze(ad2)};\\ \{< \texttt{A} >\};\\ \texttt{tra(ad1)};\\ \texttt{ad2} : \ldots\end{array}$$

## 21.11 Classe

La compilation d'une classe a pour but de produire du code pour, au moment de l'exécution :

— créer, dans la pile d'exécution, la référence de classe,

— créer la table de la classe (un champ « taille des objets », et un champ « adresse » par méthode virtuelle).

Par ailleurs la compilation produit le code objet pour chacune des opérations de la classe (constructeur, opérations auxiliaires et méthodes virtuelles, propres ou redéfinies).

Dans le schéma de compilation des classes on prend en compte *l'expansion* de chaque classe. Ci-dessous les adresses $ad_0, .., ad_m$ représentent les adresses où débute le code respectif des méthodes virtuelles d'adresses statiques $0, ..., m$. Le schéma de traduction se définit par :

$$\left\{\begin{array}{l}\textbf{type } t \textbf{ is class}\\ \quad \mathrm{a_0} : \mathrm{t_0};\\ \quad \vdots\\ \quad \mathrm{a_n} : \mathrm{t_n};\\ \textbf{interface}\\ \quad \text{spécif constructeur};\\ \quad \text{spécif méthode}_0;\\ \quad \vdots\\ \quad \text{spécif méthode}_\mathrm{m};\\ \textbf{implementation}\\ \quad < \texttt{I} >\\ \textbf{end};\end{array}\right\} = \begin{array}{l}\\ \\ \\ \\ \texttt{empilerTas(n + 1)};\\ \texttt{empilerIpTas()};\\ \texttt{tra(ad)};\\ \{< \texttt{I} >\};\\ \texttt{ad :}\\ \texttt{empilerTas(ad}_0\texttt{)};\\ \quad \vdots\\ \texttt{empilerTas(ad}_\mathrm{m}\texttt{)};\end{array}$$

Il faut à présent étudier la compilation des méthodes et de leur appel.

Dans une première étape nous étudions la compilation d'une méthode-procédure sans paramètre et de son appel. Ensuite nous considérons les cas des méthodes-procédures avec paramètres avant d'envisager le cas des autres opérations (constructeurs et méthodes-fonctions). L'étude de la compilation des méthodes-fonctions et des constructeurs vient compléter l'étude de la compilation d'expressions.

### 21.11.1 Méthode-procédure sans paramètre

Nous rappelons qu'à tout appel de méthode-procédure est associé l'objet (le préfixe) sur lequel l'opération s'applique : $< O > .p$ ($< O >$ : l'objet, $p$ : la méthode-procédure). Le préfixe se comporte comme un paramètre d'entrée dont l'adresse dynamique est empilée *sous* le bloc de liaison au moment de l'appel.

#### Compilation de la définition

$$\left\{\begin{array}{l}\textbf{procedure p is}\\ \quad < \texttt{A} >\\ \textbf{begin}\\ \quad < \texttt{B} >\\ \textbf{end};\end{array}\right\} = \begin{array}{l}\{< \texttt{A} >\};\\ \{< \texttt{B} >\};\\ \texttt{retourProc}()\end{array}$$

Comment accède-t-on au préfixe dans le corps de la méthode-procédure ? Nous étudierons ce problème ci-dessous. Pour le moment il suffit de se souvenir que le préfixe est traité comme un paramètre d'entrée.

**Affectation d'un attribut.** Un attribut de classe (un champ d'objet) peut être affecté dans le corps de toute opération (et en particulier des méthodes-procédures). Soit $< A >$ l'identificateur de l'attribut, et $< E >$ l'expression qui lui est affectée.

$$\{< \texttt{A} >:=< \texttt{E} >\} = \begin{array}{l}\texttt{empilerAdAt}(\texttt{as}(< \texttt{A} >));\\ \{< \texttt{E} >\};\\ \texttt{affectation}()\end{array}$$

**Attribut en tant qu'expression.**

$$\{< \texttt{A} >\} = \begin{array}{l}\texttt{empilerAdAt}(\texttt{as}(< \texttt{A} >));\\ \texttt{valeurPile}()\end{array}$$

#### Compilation de l'appel

Soit l'expression d'objet $< O >$ et la méthode-procédure $p$.

$$\{< \texttt{0} >.\texttt{p}\} = \begin{array}{l} \{< \texttt{0} >\}; \\ \texttt{reserverBloc}(); \\ \texttt{traVirt}(\texttt{as}(\texttt{p}), \texttt{0}); \end{array}$$

### 21.11.2 Méthode-procédure avec paramètres

**Remarque (Abus de notation).** Dans la suite de la section 21 les définitions des méthodes sont présentées *avec* leurs paramètres.

On rappelle que nilNovi Objet distingue deux types de paramètres : les paramètres d'entrée (mode *in*) et les paramètres d'entrée-sortie (mode *in out*). Dans le premier cas le paramètre effectif transmis est une *expression* qui est évaluée, qui est considérée dans le corps de l'opération comme une constante, que l'on peut consulter mais pas modifier. Dans le second cas (paramètre d'entrée-sortie) le paramètre effectif est une *variable,* un *attribut,* ou un *paramètre d'entrée-sortie* qui peut, lors de l'appel contenir une valeur, qui peut être modifiée et consultée dans le corps de l'opération. Le principe du traitement des paramètres découle de ces éléments et s'articule autour des conventions suivantes :

- dans le cas d'un paramètre formel de mode *in,* à l'exécution le paramètre effectif est empilé lors de l'appel (au-dessus du bloc de liaison),
- dans le cas d'un paramètre formel de mode *in out,* à l'exécution l'adresse dynamique du paramètre effectif est empilée à l'appel (au-dessus du bloc de liaison),
- le cas du préfixe se traite comme un paramètre de mode *in* excepté que le paramètre effectif est empilé juste *sous* le bloc de liaison.

La déclaration de l'en-tête de la procédure ne produit pas de code objet :

$$\{\textbf{procedure}\ \texttt{p}(\texttt{p}_1, \ldots, \texttt{p}_\texttt{n}\ : \ldots)\ \textbf{is}\} = \texttt{rien}$$

si $n$ est le nombre total de paramètres de la procédure.

**Paramètres de mode *in*.**

Considérons la méthode-procédure $p$ suivante :

```
procedure p(...,<X> : in <type>;...) is
    ...
begin
    ...
    ... <X>...
    ...
end;
```

et l'appel :

```
p(...,<E>,...)
```

où $< E >$ est le paramètre effectif correspondant à $< X >$.

**Paramètre formel :** le $< X >$ de l'en-tête ne produit aucun code objet. Le $< X >$ du bloc impératif de la procédure joue le rôle d'une expression et se compile comme une variable locale de la manière suivante :

| | |
|---|---|
| {< X >}= | empilerAd(as < X >));<br>valeurPile() |

**Paramètre effectif :** dans l'appel, $< E >$ est une expression. Elle se compile comme toute expression.

**Paramètres de mode *in out*.**

Considérons la méthode-procédure $p$ suivante :

```
procedure p(...,<X> : in out <type>;...) is
    ...
begin
    ...
    ... <X>:=<E>;
    ... <X> ...
    ...
end;
```

où la première occurrence de $< X >$ dans le corps de la méthode en fait un usage de type variable et la seconde est une sous-expression. Soit l'appel :

```
<O>.p(...,<V>,...)
```

où $< V >$ est une variable (un paramètre formel de mode *in out* ou un attribut) qui constitue le paramètre effectif correspondant à $< X >$.

**Paramètre formel :** le $< X >$ de l'en-tête ne produit aucun code objet. Utilisé comme une variable, $< X >$ se compile de la manière suivante :

| | | |
|---|---|---|
| {< X >:=< E >} | = | empilerParam(as(< X >));<br>{< E >};<br>affectation() |

Utilisé comme une expression, $< X >$ se compile de la manière suivante :

| | | |
|---|---|---|
| {< X >} | = | empilerParam(as(< X >));<br>valeurPile() |

**Paramètre effectif :** quatre cas sont à considérer : $< V >$ est une variable globale, $< V >$ est une variable locale, $< V >$ est un paramètre formel et $< V >$ est un attribut. Cas d'une variable globale :

$$\boxed{\{< \texttt{V} >\} \quad = \quad \texttt{empiler}(\texttt{as}(< \texttt{V} >))}$$

Cas d'une variable locale :

$$\boxed{\{< \texttt{V} >\} \quad = \quad \texttt{empilerAd}(\texttt{as}(< \texttt{V} >))}$$

Cas d'un paramètre formel :

$$\boxed{\{< \texttt{V} >\} \quad = \quad \texttt{empilerParam}(\texttt{as}(< \texttt{V} >))}$$

Cas d'un attribut :

$$\boxed{\{< \texttt{V} >\} \quad = \quad \texttt{empilerAdAt}(\texttt{as}(< \texttt{V} >))}$$

**Compilation de l'appel**

Soit $< O >$ une expression d'objet, $p$ une méthode-procédure d'adresse statique $i$ et $p_1, \ldots, p_n$ $n$ paramètres effectifs de $p$.

$$\boxed{\begin{array}{lll} & & \{< \texttt{O} >\} \\ & & \texttt{reserverBloc}() \\ \{< \texttt{O} > .\texttt{p}(\texttt{p}_1, \ldots, \texttt{p}_\texttt{n})\} & = & \{\texttt{p}_1\}; \\ & & \ldots \\ & & \{\texttt{p}_\texttt{n}\}; \\ & & \texttt{traVirt}(\texttt{i}, \texttt{n}); \end{array}}$$

**Préfixe :** ainsi que nous l'avons précisé ci-dessus le préfixe joue le rôle (et donc se traite comme) un paramètre de mode *in.* La seule différence est que l'adresse statique d'un préfixe est toujours $-3$ puisque le paramètre effectif est empilé *sous* le bloc de liaison.

### 21.11.3 Méthode-fonction

La compilation et l'exécution d'une méthode-fonction ne sont pas fondamentalement différentes de celles d'une méthode-procédure. Un point l'en distingue : les fonctions délivrent une valeur. Cette particularité est prise en compte par l'instruction de retour *retourFonct().*

**Compilation de la définition**
Soit une méthode-fonction dotée de $n$ paramètres formels.

$$\left\{\begin{array}{l}\textbf{function } \mathrm{f}(\mathrm{p_1},\ldots,\mathrm{p_n}:\ldots) \textbf{ return } <\mathtt{t}> \textbf{ is}\\ \quad <\mathtt{A}>\\ \textbf{begin}\\ \quad <\mathtt{B}>\\ \textbf{end};\end{array}\right\} = \begin{array}{l}\{<\mathtt{A}>\};\\ \{<\mathtt{B}>\};\end{array}$$

Une méthode-fonction contient toujours au moins (et son exécution s'achève toujours par) une instruction *return* <*E*>. Une telle instruction se compile de la manière suivante :

$$\{\textbf{return} <\mathbf{E}>\} \quad = \quad \begin{array}{l}\{<\mathtt{E}>\};\\ \mathtt{retourFonct}()\end{array}$$

Concernant les paramètres effectifs (de mode *in* uniquement), ils sont traités de la même façon que dans les méthodes-procédures.

**Compilation de l'appel**

Soit $< O >$ une expression d'objet, $f$ une méthode-fonction d'adresse statique $i$ et $p_1, \ldots, p_n$ $n$ paramètres effectifs.

$$\{<\mathtt{O}>.\mathtt{f}(\mathrm{p_1},\ldots,\mathrm{p_n})\} \quad = \quad \begin{array}{l}\{<\mathtt{O}>\}\\ \mathtt{reserverBloc}()\\ \{\mathrm{p_1}\};\\ \ldots\\ \{\mathrm{p_n}\};\\ \mathtt{traVirt}(\mathtt{i},\mathtt{n})\end{array}$$

### 21.11.4 Constructeur

Du point de vue de la compilation/exécution un constructeur se comporte comme une méthode-fonction à l'exception des points suivants :
- le préfixe est une classe (et non un objet),
- la valeur délivrée par l'appel est toujours une référence.
- La valeur délivrée par l'appel l'est implicitement (et non par l'utilisation d'une instruction *return*).
- c'est une méthode *statique* : dans un appel à un constructeur, l'adresse du code à exécuter est connue dès la compilation.

**Compilation de la définition**

$$\left\{\begin{array}{l}\textbf{constructor } \mathtt{c}(\mathtt{p_1},...,\mathtt{p_n} : \ldots) \textbf{ is} \\ \quad < \mathtt{A} > \\ \textbf{begin} \\ \quad < \mathtt{B} > \\ \textbf{end};\end{array}\right\} = \begin{array}{l}\{< \mathtt{A} >\}; \\ \{< \mathtt{B} >\}; \\ \mathtt{retourConstr}()\end{array}$$

**Compilation de l'appel**

Soit $< C >$ un identificateur de classe, $c$ le constructeur correspondant, implanté à l'adresse $adc$, et soit $p_1, .., p_n$ les $n$ paramètres effectifs.

$$\{< \mathtt{C} > .\mathtt{c}(\mathtt{p_1},...,\mathtt{p_n})\} = \begin{array}{l}\mathtt{empiler}(\mathtt{as}(< \mathtt{C} >); \\ \mathtt{valeurPile}(); \\ \mathtt{reserverBloc}(); \\ \{\mathtt{p_1}\}; \\ \quad \vdots \\ \{\mathtt{p_2}\}; \\ \mathtt{traConstr}(\mathtt{adc},\mathtt{n});\end{array}$$

# 22 Exemples

Dans cette section nous présentons quelques exemples de compilation de programmes nilNovi Objet.

## 22.1 Exemple 1

Dans cet exemple nous reprenons le programme de l'exemple 1 de la section 16 page 51 du document. Ce programme est rappelé ci-dessous, les adresses statiques sont mentionnées en commentaires.

```
01 : procedure p is
02 :    type liste is class                          //liste : 0
03 :       val : integer ;                           //val : 0
04 :       svt :liste ;                              //svt : 1
05 :    interface
06 :       constructor create(v : integer ;s :liste) ; //v : 0 ; s : 1
07 :       function valeur() return integer ;        //valeur : 0
08 :       function suivant() return liste ;         //suivant : 1
09 :    implementation
10 :       constructor create is
```

```
11 :        begin
12 :           val :=v ;                          //val : 0 ;v : 0
13 :           svt :=s                            //svt : 1  s : 1
14 :        end ;
15 :        function valeur is                    //valeur : 0
16 :        begin
17 :           return val                         //val : 0
18 :        end ;
19 :        function suivant is                   //suivant : 1
20 :        begin
21 :           return svt                         //svt : 1
22 :        end ;
23 :     end ;
24 :     i : integer ;                            //i : 1
25 :     e,tete :liste ;                          //e : 2 ;tete : 3
26 : begin
27 :     tete :=nil ;                             //tete : 3
28 :     i := 1;                                  //i : 1
29 :     while i/=10 loop                         //i : 1
30 :        //la liste des valeurs de 1 à i-1 est créée,
31 :        //elle est désignée par la variable tete
32 :        tete :=liste.create(i,tete) ;         //tete : 3 ; i : 1
33 :        i :=i+1                               //i : 1
34 :     end ;
35 :     e :=tete ;                               //e :2 ; tete : 3
36 :     while e/=nil loop                        //e : 2
37 :        //les valeurs de la liste depuis tete inclus
38 :        //jusqu'à e exclus sont affichées
39 :        put(e.valeur()) ;                     //e : 2 ; valeur : 0
40 :        e :=e.suivant()                       //e : 2 ; suivant : 1
41 :     end
42 : end.
```

Le code objet correspondant à la compilation de ce programme se présente de la manière suivante (la ligne source est rappelée à droite de la ligne objet) :

```
00 :   debutProg()                    01
01 :   empilerTas(2)                  02/04
02 :   empilerIpTas()                 02/04
03 :   tra(19)                        02
04 :   empilerAdAt(0)                 12
05 :   empilerAd(0)                   12
```

```
06 :   valeurPile()                    12
07 :   affectation()                   12
08 :   empilerAdAt(1)                  13
09 :   empilerAd(1)                    13
10 :   valeurPile()                    13
11 :   affectation()                   13
12 :   retourConstr()                  14
13 :   empilerAdAt(0)                  17
14 :   valeurPile()                    17
15 :   retourFonct()                   18
16 :   empilerAdAt(1)                  21
17 :   valeurPile()                    21
18 :   retourFonct()                   22
19 :   empilerTas(13)                  06/15
20 :   empilerTas(16)                  07/19
21 :   reserver(3)                     24/25
22 :   empiler(3)                      27
23 :   empiler(-1)                     27
24 :   affectation()                   27
25 :   empiler(1)                      28
26 :   empiler(1)                      28
27 :   affectation()                   28
28 :   empiler(1)                      29
29 :   valeurPile()                    29
30 :   empiler(10)                     29
31 :   diff()                          29
32 :   tze(50)                         29
33 :   empiler(3)                      32
34 :   empiler(0)                      32
35 :   valeurPile()                    32
36 :   reserverBloc()                  32
37 :   empiler(1)                      32
38 :   valeurPile()                    32
39 :   empiler(3)                      32
40 :   valeurPile()                    32
41 :   traConstr(04,2)                 32
42 :   affectation()                   33
43 :   empiler(1)                      33
44 :   empiler(1)                      33
45 :   valeurPile()                    33
46 :   empiler(1)                      33
47 :   add()                           33
48 :   affectation()                   33
```

```
49 :   tra(29)                          34
50 :   empiler(2)                       35
51 :   empiler(3)                       35
52 :   valeurPile()                     35
53 :   affectation()                    35
54 :   empiler(2)                       36
55 :   valeurPile()                     36
56 :   empiler(-1)                      36
57 :   diff()                           36
58 :   tze(71)                          36
59 :   empiler(2)                       39
60:    valeurPile()                     39
61:    reserverBloc()                   39
62 :   traVirt(0,0)                     39
63 :   put()                            39
64 :   empiler(2)                       40
65 :   empiler(2)                       40
66 :   valeurPile()                     40
67 :   reserverBloc()                   40
68 :   traVirt(1,0)                     40
69 :   affectation()                    40
70 :   tra(54)                          41
71 :   finProg()                        42
```

## 22.2 Exemple 2

Cet exemple reprend le programme de l'exemple 3 de la section 16 page 59 du document.

```
01 : procedure e is
02 :    type exp is class
03 :    interface
04 :       constructor create() ;
05 :       function eval() return integer ;
06 :    implementation
07 :       constructor create is
08 :       begin
09 :          error(1)
10 :       end ;
11 :       function eval is
12 :       begin
13 :          error(2) ;return 0
```

```
14 :        end ;
15 :     end ;
16 :     type un is class(exp)//expressions unaires
17 :        v : integer ;
18 :     interface
19 :        constructor create(i : integer) ;
20 :        function eval() return integer ;
21 :     implementation
22 :        constructor create is
23 :        begin
24 :           v : =i
25 :        end ;
26 :        function eval is
27 :        begin
28 :           return v
29 :        end ;
30 :     end ;
31 :     type bin is class(exp)//expressions binaires
32 :        g,d : exp ;
33 :     interface
34 :        constructor create() ;
35 :        function eval() return integer ;
36 :        function op(a,b : exp) return integer ;
37 :     implementation
38 :        constructor create is
39 :        begin
40 :           error(3)
41 :        end ;
42 :        function eval is
43 :        begin
44 :           return this.op(g,d)
45 :        end ;
46 :        function op is
47 :        begin
48 :           error(4) ;return 0
49 :        end ;
50 :     end ;
51 :     type plus is class(bin)//expressions g+d
52 :     interface
53 :        constructor create(l,r : exp) ;
54 :        function op(a,b : exp) return integer ;
55 :     implementation
56 :        constructor create is
```

```
57 :        begin
58 :           g :=l ;
59 :           d :=r
60 :        end ;
61 :        function op is
62 :        begin
63 :           return a.eval()+b.eval()
64 :        end ;
65 :     end ;
66 :     type mult is class(bin) //expressions g*d
67 :     interface
68 :        constructor create(l,r : exp) ;
69 :        function op(a,b : exp) return integer ;
70 :     implementation
71 :        constructor create is
72 :        begin
73 :           g :=l ;
74 :           d :=r
75 :        end ;
76 :        function op is
77 :        begin
78 :           return a.eval()*b.eval()
79 :        end ;
80 :     end ;
81 :     a,b,e : un ;
82 :     c : mult ;
83 :     d : plus ;
84 :     f : exp ;
85 :     val : integer ;
86 : begin
87 :     get(val) ;
88 :     a :=un.create(val) ;
89 :     b :=un.create(2) ;
90 :     c :=mult.create(a,b) ;
91 :     e :=un.create(1) ;
92 :     d :=plus.create(e,c) ;
93 :     f :=d ;
94 :     put(f.eval())
95 : end.
```

Le code objet correspondant à la compilation de ce programme se présente de la manière suivante (la ligne source est rappelée à droite de la ligne objet) :

```
00:    debutProg()                    01
01:    empilerTas(0)                  03
02:    empilerIpTas()                 03
03:    tra(11)                        03
04:    empiler(1)                     09
05:    erreur()                       09
06:    retourConstr()                 10
07:    empiler(2)                     13
08:    erreur()                       13
09:    empiler(0)                     13
10:    retourFonct()                  13
11:    empilerTas(7)                  15
12:    empilerTas(1)                  18
13:    empilerIpTas()                 18
14:    tra(23)                        18
15:    empilerAdAt(0)                 24
16:    empilerAd(0)                   24
17:    valeurPile()                   24
18:    affectation()                  24
19:    retourConstr()                 25
20:    empilerAdAt(0)                 28
21:    valeurPile()                   28
22:    retourFonct()                  29
23:    empilerTas(20)                 30
24:    empilerTas(2)                  33
25:    empilerIpTas()                 33
26:    tra(43)                        33
27:    empiler(3)                     40
28:    erreur()                       40
29:    retourConstr()                 41
30:    empilerAd(-3)                  44
31:    valeurPile()                   44
32:    reserverBloc()                 44
33:    empilerAdAt(0)                 44
34:    valeurPile()                   44
35:    empilerAdAt(1)                 44
36:    valeurPile()                   44
37:    traVirt(1,2)                   44
38:    retourFonct()                  44
39:    empiler(4)                     48
40:    erreur()                       48
41:    empiler(0)                     48
42:    retourFonct()                  48
```

```
43:    empilerTas(30)                50
44:    empilerTas(39)                50
45:    empilerTas(2)                 51
46:    empilerIpTas()                51
47:    tra(67)                       51
48:    empilerAdAt(0)                58
49:    empilerAd(0)                  58
50:    valeurPile()                  58
51:    affectation()                 58
52:    empilerAdAt(1)                59
53:    empilerAd(1)                  59
54:    valeurPile()                  59
55:    affectation()                 59
56:    retourConstr()                60
57:    empilerAd(0)                  63
58:    valeurPile()                  63
59:    reserverBloc()                63
60:    traVirt(0,0)                  63
61:    empilerAd(1)                  63
62:    valeurPile()                  63
63:    reserverBloc()                63
64:    traVirt(0,0)                  63
65:    add()                         63
66:    retourFonct()                 64
67:    empilerTas(30)                65
68:    empilerTas(57)                65
69:    empilerTas(2)                 66
70:    empilerIpTas()                66
71:    tra(91)                       66
72:    empilerAdAt(0)                73
73:    empilerAd(0)                  73
74:    valeurPile()                  73
75:    affectation()                 73
76:    empilerAdAt(1)                74
77:    empilerAd(1)                  74
78:    valeurPile()                  74
79:    affectation()                 74
80:    retourConstr()                75
81:    empilerAd(0)                  78
82:    valeurPile()                  78
83:    reserverBloc()                78
84:    traVirt(0,0)                  78
85:    empilerAd(1)                  78
```

```
86:   valeurPile()                78
87:   reserverBloc()               78
88:   traVirt(0,0)                 78
89:   mult()                       78
90:   retourFonct()                79
91:   empilerTas(30)               80
92:   empilerTas(81)               80
93:   reserver(7)                  81/85
94:   empiler(11)                  87
95:   get()                        87
96:   empiler(5)                   88
97:   empiler(1)                   88
98:   valeurPile()                88
99:   reserverBloc()               88
100:  empiler(11)                  88
101:  valeurPile()                 88
102:  traConstr(15,1)              88
103:  affectation()                88
104:  empiler(6)                   89
105:  empiler(1)                   89
106:  valeurPile()                 89
107:  reserverBloc()               89
108:  empiler(2)                   89
109:  traConstr(15,1)              89
110:  affectation()                89
111:  empiler(8)                   90
112:  empiler(4)                   90
113:  valeurPile()                 90
114:  reserverBloc()               90
115:  empiler(5)                   90
116:  valeurPile()                 90
117:  empiler(6)                   90
118:  valeurPile()                 90
119:  traConstr(72,2)              90
120:  affectation()                90
121:  empiler(7)                   91
122:  empiler(1)                   91
123:  valeurPile()                 91
124:  reserverBloc()               91
125:  empiler(1)                   91
126:  traConstr(15,1)              91
127:  affectation()                91
128:  empiler(9)                   92
```

```
129:   empiler(3)                92
130:   valeurPile()              92
131:   reserverBloc()            92
133:   valeurPile()              92
134:   empiler(8)                92
135:   valeurPile()              92
136:   traConstr(48,2)           92
137:   affectation()             92
138:   empiler(10)               93
139:   empiler(9)                93
140:   valeurPile()              93
141:   affectation()             93
142:   empiler(10)               94
143:   valeurPile()              94
144:   reserverBloc()            94
145:   traVirt(0,0)              94
146:   put()                     94
147:   finProg()                 95
```

# Annexe

## A La grammaire nilNovi Algorithmique

```
<programme> ::=
     <specifProgPrinc> is <corpsProgPrinc>

<corpsProgPrinc> ::=
     <partieDecla> begin <suiteInstr> end . |
     begin <suiteInstr> end .

<specifProgPrinc> ::=
     procedure <ident>

<partieDecla> ::=
     <listeDeclaVar>

<listeDeclaVar> ::=
     <declaVar> <listeDeclaVar> |
     <declaVar>

<declaVar> ::=
     <listeIdent> : <type> ;

<type> ::=
     boolean |
     integer

<listeIdent> ::=
     <ident> , <listeIdent> |
     <ident>

<suiteInstr> ::=
     <instr> ; <suiteInstr> |
     <instr>

<instr> ::=
     <affectation> |
     <boucle> |
     <altern> |
     <es>

<affectation> ::=
```

```
     <ident> := <expression>

<expression> ::=
     <expression> or <exp1> |
     <exp1>

<exp1> ::=
     <exp1> and <exp2> |
     <exp2>

<exp2> ::=
     <exp3> <opRel> <exp3> |
     <exp3>

<opRel> ::=
     = |
     /= |
     < |
     <= |
     > |
     <=

<exp3> ::=
     <exp3> <opAd> <exp4> |
     <exp4>

<opAd> ::=
     + |
     -

<exp4> ::=
     <exp4> <opMult> <prim> |
     <prim>

<opMult> ::=
     * |
     /

<prim> ::=
     <opUnaire> <elemPrim> |
     <elemPrim>

<opUnaire> ::=
     +|
     -|
     not
```

```
<elemPrim> ::=
     <valeur> |
     <ident> |
     ( <expression> )

<valeur> ::=
     <entier> |
     <valBool>

<valBool> ::=
     true |
     false

<es> ::=
     get ( <ident> ) |
     put ( <expression> )

<boucle> ::=
     while <expression> loop <suiteInstr> end

<altern> ::=
     if <expression> then <suiteInstr> end |
     if <expression> then <suiteInstr> else <suiteInstr> end

<ident> ::=
     <lettre> <listeLettreChiffre> |
     <lettre>

<listeLettreChiffre> ::=
     <lettreOuChiffre> <listeLettreChiffre> |
     <lettreOuChiffre>

<lettreOuChiffre> ::=
     <lettre> |
     <chiffre>

<lettre> ::=
     a | ... | z | A | ... | Z

<chiffre> ::=
     0 | 1 | .. | 9

<entier> ::=
     <chiffre> <entier> |
     <chiffre>
```

# B La grammaire nilNovi Procédural

```
<programme> ::=
     <specifProgPrinc> is <corpsProgPrinc>

<corpsProgPrinc> ::=
     <partieDecla> begin <suiteInstr> end . |
     begin <suiteInstr> end .

<specifProgPrinc> ::=
     procedure <ident>

<partieDecla> ::=
     <listeDeclaOp> <listeDeclaVar> |
     <listeDeclaVar> |
     <listeDeclaOp>

<listeDeclaOp> ::=
     <declaOp> ; <listeDeclaOp> |
     <declaOp> ;

<declaOp> ::=
     <fonction> |
     <procedure>

<procedure> ::=
     procedure <ident> <partieFormelle> is <corpsProc>

<fonction> ::=
     function <ident> <partieFormelle> return <type> is <corpsFonct>

<corpsProc> ::=
     <partieDeclaProc> begin <suiteInstr> end |
     begin <suiteInstr> end

<corpsFonct> ::=
     <partieDeclaProc> begin <suiteInstrNonVide> end |
     begin <suiteInstrNonVide> end

<partieFormelle> ::=
```

```
     ( <listeSpecifFormelles> )
     ( )

<listeSpecifFormelles> ::=
     <specif> ; <listeSpecifFormelles> |
     <specif>

<specif> ::=
     <listeIdent> : <mode> <type> |
     <listeIdent> : <type>

<mode> ::=
     in |
     in out

<type> ::=
     integer |
     boolean

<partieDeclaProc> ::=
     <listeDeclaVar>

<listeDeclaVar> ::=
     <declaVar> <listeDeclaVar> |
     <declaVar>

<declaVar> ::=
     <listeIdent> : <type> ;

<listeIdent> ::=
     <ident> , <listeIdent> |
     <ident>

<suiteInstrNonVide> ::=
     <instr> ; <suiteInstrNonVide> |
     <instr>

<suiteInstr> ::=
     <suiteInstrNonVide> |
     ϵ

<instr> ::=
     <affectation> |
     <boucle> |
     <altern> |
     <es> |
```

```
     <retour> |
     <appelProc>

<appelProc> ::=
     <ident> ( <listePe> ) |
     <ident> ( )

<listePe> ::=
     <expression> , <listePe> |
     <expression>

<affectation> ::=
     <ident> := <expression>

<expression> ::=
     <expression> or <exp1> |
     <exp1>

<exp1> ::=
     <exp1> and <exp2> |
     exp2

<exp2> ::=
     <exp3> <opRel> <exp3> |
     <exp3>

<opRel> ::=
     = |
     /= |
     < |
     <= |
     > |
     <=

<exp3> ::=
     <exp3> <opAd> <exp4> |
     <exp4>

<opAd> ::=
     + |
     -

<exp4> ::=
     <exp4> <opMult> <prim> |
     <prim>
```

```
<opMult> ::=
     * |
     /

<prim> ::=
     <opUnaire> <elemPrim> |
     <elemPrim>

<opUnaire> ::=
     +|
     -|
     not

<elemPrim> ::=
     <valeur> |
     ( <expression> ) |
     <ident> |
     <appelFonct>

<appelFonct> ::=
     <ident> ( <listePe> ) |
     <ident> ( )

<valeur> ::=
     <entier> |
     <valBool>

<valBool> ::=
     true |
     false

<es> ::=
     get ( <ident> ) |
     put ( <expression> )

<boucle> ::=
     while <expression> loop <suiteInstr> end

<altern> ::=
     if <expression> then <suiteInstr> end |
     if <expression> then <suiteInstr> else <suiteInstr> end

<retour> ::=
     return <expression>

<ident> ::=
```

```
     <lettre> <listeLettreChiffre> |
     <lettre>

<listeLettreChiffre> ::=
     <lettreOuChiffre> <listeLettreChiffre> |
     <lettreOuChiffre>

<lettreOuChiffre> ::=
     <lettre> |
     <chiffre>

<lettre> ::=
     a | ... | z | A | ... | Z

<chiffre> ::=
     0 | 1 | .. | 9

<entier> ::=
     <chiffre> <entier> |
     <chiffre>
```

# C La grammaire nilNovi Objet

```
<programme> ::=
     <specifProgPrinc> is <corpsProgPrinc>

<corpsProgPrinc> ::=
     <partieDecla> begin <suiteInstr> end . |
     begin <suiteInstr> end .

<specifProgPrinc> ::=
     procedure <identificateur>

<partieDecla> ::=
     <listeDeclaClass> <listeDeclaVar> |
     <listeDeclaVar> |
     <listeDeclaClass>

<listeDeclaClass> ::=
     <declaClass> <listeDeclaClass> |
     <declaClass>
```

<declaClass> ::=
**type** <identificateur> **is class** <heritage> <attributs> **interface**
<interface> **implementation** <implementation> **end ;**

<heritage> ::=
( <identificateur> ) |
$\epsilon$

<attributs> ::=
<listeChamps> |
$\epsilon$

<interface> ::=
<listeSpecifMethodes>

<listeChamps> ::=
<listeDeclaVar>

<listeSpecifMethodes> ::=
<listeSpecifMethodes> <specifMethode> ; |
<specifMethode> ;

<specifMethode> ::=
<specifMethodeProc> |
<specifMethodeFonct> |
<specifMethodeConst>

<specifMethodeProc> ::=
**procedure** <identificateur> <partieFormelle>

<specifMethodeFonct> ::=
**function** <identificateur> <partieFormelle> **return** <identificateur>

<specifMethodeConst> ::=
**constructor** <identificateur> <partieFormelle>

<implementation> ::=
<listeOperations>

<listeOperations> ::=
<operation> ; <listeOperations> |
<operation> ;

<operation> ::=
<constructeur> |

```
     <fonction> |
     <procedure>

<constructeur> ::=
     constructor <identificateur> is <corpsProc>

<procedure> ::=
     procedure <identificateur> is <corpsProc>

<fonction> ::=
     function <identificateur> is <corpsFonct>

<corpsProc> ::=
     <partieDeclaProc> begin <suiteInstr> end |
     begin <suiteInstr> end

<corpsFonct> ::=
     <partieDeclaProc> begin <suiteInstrNonVide> end |
     begin <suiteInstrNonVide> end

<partieFormelle> ::=
     ( <listeSpecifFormelles> ) |
     ( )

<listeSpecifFormelles> ::=
     <specif> ; <listeSpecifFormelles> |
     <specif>

<specif> ::=
     <listeIdent> : <mode> <identificateur> |
     <listeIdent> : <identificateur>

<mode> ::=
     in |
     in out

<partieDeclaProc> ::=
     <listeDeclaVar>

<listeDeclaVar> ::=
     <declaVar> <listeDeclaVar> |
     <declaVar>

<declaVar> ::=
     <listeIdent> : <identificateur> ;
```

```
<listeIdent> ::=
     <identificateur> , <listeIdent> |
     <identificateur>

<suiteInstrNonVide> ::=
     <instr> ; <suiteInstrNonVide> |
     <instr>

<suiteInstr> ::=
     <suiteInstrNonVide> |
     ϵ

<instr> ::=
     <affectation> |
     <boucle> |
     <altern> |
     <es> |
     <retour> |
     <appelOperation> |
     error ( <expression> )

<appelOperation> ::=
     <expressionObjet>.<appelProc>

<appelProc> ::=
     <identificateur> ( <listePe> ) |
     <identificateur> ( )

<listePe> ::=
     <expression> , <listePe> |
     <expression>

<affectation> ::=
     <identificateur> := <expression>

<expression> ::=
     <expression> or <exp1> |
     <exp1>

<exp1> ::=
     <exp1> and <exp2> |
     <exp2>

<exp2> ::=
     <exp3> <opRel> <exp3> |
     <exp3>
```

```
<opRel> ::=
     = |
     /= |
     < |
     <= |
     > |
     <=

<exp3> ::=
     <exp3> <opAd> <exp4> |
     <exp4>

<opAd> ::=
     + |
     -

<exp4> ::=
     <exp4> <opMult> <prim> |
     <prim>

<opMult> ::=
     * |
     /

<prim> ::=
     <opUnaire> <elemPrim> |
     <elemPrim>

<opUnaire> ::=
     +|
     -|
     not

<elemPrim> ::=
     <valeur> |
     ( <expression> ) |
     <identificateur> |
     <expressionObjet> |
     <appelFonct>

<expressionObjet> ::=
     <identificateur> . <listeAppelFonct>

<listeAppelFonct> ::=
     <appelFonct> |
```

```
     <listeAppelFonct> . <appelFonct>

<appelFonct> ::=
     <identificateur> ( <listePe> ) |
     <identificateur> ( )

<valeur> ::=
     <entier> |
     <valBool>

<valBool> ::=
     true |
     false

<es> ::=
     get ( <identificateur> ) |
     put ( <expression> )

<boucle> ::=
     while <expression> loop <suiteInstr> end

<altern> ::=
     if <expression> then <suiteInstr> end |
     if <expression> then <suiteInstr> else <suiteInstr> end

<retour> ::=
     return <expression>

<identificateur> ::=
     <lettre> <listeLettreChiffre> |
     <lettre>

<listeLettreChiffre> ::=
     <lettreOuChiffre> <listeLettreChiffre> |
     <lettreOuChiffre>

<lettreOuChiffre> ::=
     <lettre> |
     <chiffre>

<lettre> ::=
     a | ... | z | A | ... | Z

<chiffre> ::=
     0 | 1 | .. | 9
```

```
<entier> ::=
     <chiffre> <entier> |
     <chiffre>
```

# D Tableau des instructions nilNovi Algorithmique et procédural

| Nom | Description | Alg. | Proc. |
|---|---|---|---|
| debutProg() | début d'un programme nilNovi Objet | 6 | 30 |
| finProg() | Fin d'un programme nilNovi Objet | 7 | 31 |
| reserver(n : integer) | Réserve $n$ emplacements dans la pile d'exécution | 7 | 31 |
| empiler(val : integer) | Empile *val* dans la pile d'exécution | 7 | 31 |
| empilerAd(ad : integer) | Empile l'adresse d'un emplacement de la pile | | 32 |
| affectation() | Affecte la valeur au sommet de pile à la variable désignée par l'emplacement sous le sommet | 8 | 32 |
| valeurPile() | Remplace le sommet de la pile par la valeur désignée par ce sommet | 8 | 32 |
| get() | Place la valeur lue dans la variable désignée par le sommet | 9 | 32 |
| put() | Affiche la valeur au sommet de la pile | 9 | 33 |
| moins() | Calcule l'opposé de la valeur en sommet de pile | 10 | 33 |
| sous() | Calcule la différence entre les deux valeurs en sommet de pile | 10 | 33 |
| add() | Calcule la somme des deux valeurs en sommet de pile | 11 | 33 |
| mult() | Calcule le produit des deux valeurs en sommet de pile | 11 | 33 |
| div() | Divise la valeur sous le sommet par la valeur en sommet de pile | 11 | 33 |
| egal() | Empile vrai si les deux valeurs en sommet de pile sont égales, faux sinon | 11 | 33 |
| diff() | Empile vrai si les deux valeurs en sommet de pile sont différentes, faux sinon | 12 | 34 |
| inf() | Empile vrai si la valeur sous le sommet de la pile est inférieur à la valeur au sommet, faux sinon | 12 | 34 |
| infeg() | Empile vrai si la valeur sous le sommet de la pile est inférieure ou égale à la valeur au sommet, faux sinon | 12 | 34 |
| sup() | Empile vrai si la valeur sous le sommet de la pile est supérieure à la valeur au sommet, faux sinon | 12 | 34 |
| supeg() | Empile vrai si la valeur sous le sommet de la pile est supérieure ou égale à la valeur au sommet, faux sinon | 12 | 34 |

| Nom | Description | Alg. | Proc. |
|---|---|---|---|
| et() | Empile vrai si les deux valeurs en sommet de pile sont vraies, faux sinon | 12 | 34 |
| ou() | Empile vrai si au moins l'une des deux valeurs en sommet de pile est vraie, faux sinon | 13 | 34 |
| non() | Empile vrai si la valeur en sommet de pile est fausse, faux sinon | 13 | 34 |
| tra(ad : integer) | Donne le contrôle à l'instruction à l'adresse ad | 13 | 34 |
| tze(ad : integer) | Donne le contrôle à l'instruction à l'adresse ad si le sommet de pile contient *faux.* Continue en séquence sinon | 14 | 34 |
| reserverBloc() | Au moment de l'appel d'une opération, réserve et initialise un bloc de liaison | | 35 |
| retourFonct() | Instruction générée à rencontre d'un return dans une fonction, assure le retour | | 35 |
| retourProc() | Instruction générée à la fin d'une procédure, qui assure le retour | | 36 |
| empilerParam (ad : integer) | Instruction pour la gestion des paramètres formels | | 37 |
| traStat(ad :integer, nbP : integer) | Instruction permettant l'appel d'une opération | | 38 |

# E Tableau des instructions nilNovi Objet

| Code | Nom | Description | Page |
|---|---|---|---|
| 1 | debutProg() | début d'un programme nilNovi Objet | 86 |
| 2 | finProg() | Fin d'un programme nilNovi Objet | 86 |
| 3 | reserver(n : integer) | Réserve $n$ emplacements dans la pile d'exécution | 87 |
| 4 | empiler(val : integer) | Empile *val* dans la pile d'exécution | 87 |
| 5 | empilerAd(ad : integer) | Empile l'adresse d'un emplacement de la pile | 87 |
| 6 | affectation() | Affecte la valeur au sommet de pile à la variable désignée par l'emplacement sous le sommet | 88 |
| 7 | valeurPile() | Remplace le sommet de la pile par la valeur désignée par ce sommet | 88 |
| 8 | get() | Place la valeur lue dans la variable désignée par le sommet | 89 |
| 9 | put() | Affiche la valeur au sommet de la pile | 89 |
| 10 | moins() | Calcule l'opposé de la valeur en sommet de pile | 90 |
| 11 | sous() | Calcule la différence entre les deux valeurs en sommet de pile | 90 |
| 12 | add() | Calcule la somme des deux valeurs en sommet de pile | 91 |
| 13 | mult() | Calcule le produit des deux valeurs en sommet de pile | 91 |
| 14 | div() | Divise la valeur sous le sommet par la valeur en sommet de pile | 91 |
| 15 | egal() | Empile vrai si les deux valeurs en sommet de pile sont égales, faux sinon | 91 |
| 16 | diff() | Empile vrai si les deux valeurs en sommet de pile sont différentes, faux sinon | 92 |
| 17 | inf() | Empile vrai si la valeur sous le sommet de la pile est inférieur à la valeur au sommet, faux sinon | 92 |
| 18 | infeg() | Empile vrai si la valeur sous le sommet de la pile est inférieure ou égale à la valeur au sommet, faux sinon | 92 |
| 19 | sup() | Empile vrai si la valeur sous le sommet de la pile est supérieure à la valeur au sommet, faux sinon | 92 |
| 20 | supeg() | Empile vrai si la valeur sous le sommet de la pile est supérieure ou égale à la valeur au sommet, faux sinon | 92 |
| 21 | et() | Empile vrai si les deux valeurs en sommet de pile sont vraies, faux sinon | 92 |

| Code | Nom | Description | Page |
|---|---|---|---|
| 22 | ou() | Empile vrai si au moins l'une des deux valeurs en sommet de pile est vraie, faux sinon | 92 |
| 23 | non() | Empile vrai si la valeur en sommet de pile est fausse, faux sinon | 93 |
| 24 | tra(ad : integer) | Donne le contrôle à l'instruction à l'adresse ad | 93 |
| 25 | tze(ad : integer) | Donne le contrôle à l'instruction à l'adresse ad si le sommet de pile contient *faux*. Continue en séquence sinon | 94 |
| 26 | erreur() | Provoque l'arrêt de la machine | 94 |
| 27 | empilerTas(val : integer) | Empile *val* sur le tas | 95 |
| 28 | empilerIpTas() | Utilisé au début de l'exécution d'une classe, crée la référence de la classe dans la pile | 95 |
| 29 | empilerAdAt(v : integer) | Permet d'empiler l'adresse dynamique d'un champ d'un objet à partir de l'adresse statique $v$ | 96 |
| 30 | reserverBloc() | Au moment de l'appel d'une opération, réserve et initialise un bloc de liaison | 97 |
| 31 | retourConstr() | Instruction générée à la fin d'un constructeur, qui assure le retour | 98 |
| 32 | retourFonct() | Instruction générée à rencontre d'un return dans une fonction, assure le retour | 99 |
| 33 | retourProc() | Instruction générée à la fin d'une procédure, qui assure le retour | 100 |
| 34 | empilerParam(ad : integer) | Instruction pour la gestion des paramètres formels | 101 |
| 35 | traConstr (ad :integer, nbP : integer) | Instruction permettant l'appel d'un constructeur | 102 |
| 36 | traVirt (i : integer, nbP : integer) | Instruction permettant l'appel d'une opération virtuelle | 103 |

# Références

[1] Aho, A.V., Ullman, J. *The theory of parsing, translation and compiling.* Prentice Hall. 1973.

[2] Aho, A.V., Sethi, R., Ullman, J.D. *Compilateurs, principes, techniques et outils.* InterEditions. ISBN 2-7296-0295-X. 1991.

[3] Barré, J., Couvert, A. Cours C30. *Analyse syntaxique, schémas d'exécution.* IFSIC, Université de Rennes 1. 1982.

[4] Borland. *Pascal Objet.* 2002.

[5] Charon, I. *Le langage java, concepts et pratique.* Hermès. ISBN 2-7462-0117-8.

[6] CII. *Simula sous SIRIS7/SIRIS 8. Manuel d'utilisation* tome 1. 4179 R/FR, Novembre 1972.

[7] P.Y. Cunin, M. Griffiths, J. Voiron. *Comprendre la compilation.* Springer Verlag. ISBN 0-387-10327-9. 1980.

[8] Elder, J. *Compiler construction. A recursive descent approach.* Prentice hall. ISBN 0-13-291139-6. 1994.

[9] Gallaire, H. *Techniques de compilation, 2e édition.* Cepadues édition. ISBN 2-85428-1195. 1985.

[10] Levine, J., Mason, T., Brown, D. Lex & yacc. ISBN 1-56592-000-7. O'Reilly & Associates, Inc. Second Edition 1992.

[11] [Men,94] Menu, J. *Compilation avec C++. Du concept à la réalisation avec des langages objets.* Addison-Wesley. ISBN 2-87908-092-4. 1994.

[12] Meyer, B. *Conception et programmation par objets.* InterEditions. ISBN 2-7296-0272-0. 1991.

[13] Meyer B. *Introduction à la théorie des langages de programmation.* InterÉdition. 1992.

[14] Noyelle, Y. *Traitement des langages évolués. Compilation, interprétation, support d'exécution.* Masson. ISBN 2-225-81-368-X. 1988.

[15] Pair, C. *Compilation.* École d'été AFCET - Neuchâtel 1974.

[16] Schere, C. *Simula par l'exemple.* 5223 T/Fr, CII. Décembre 1972.

[17] Siverio, N. *Réaliser un compilateur. Les outils Lex et Yacc.* ISBN 2-212-08834-5. Eyrolles. 1994.

[18] Trilling, L. *Cours de compilation.* Polycopié de Rennes 1. 1970

[19] Wirth, N. *Algorithms + Data Structures = Programs.* Chapitre 5. Prentice Hall 1976.

**Documents et sites Internet :**

[20] Mycroft. Compiling Techniques. Cambridge University Computer Laboratory. Lent 1998.

`http://www.cl.cam.ac.uk/users/am/teaching/cmp`

[21] Jerzy Karczmarczuk. Compilateurs et interprètes. Cours de l'université de Caen. Janvier 2002. Consulté le 15 février 2002.

`http://users.info.unicaen.fr/~karczma/matrs/Maitcomp/compilation_h.pdf`

[22] Charles Rapin. Compilation. École Polytechnique Fédérale de Lausanne. Département d'Informatique. Consulté le 20 février 2002.

`http://diwww.epfl.ch/w3lco/pub/compilation/Compilation.html`

[23] Sacha Krakowiak. Imag. Grenoble. Consulté le 26 mars 2003.

`http://sardes.imag.fr/people/krakowia/Enseignement/ti-deug/Flips/PDF/5-Sys-DEUG.PDF`

[24] Publications de Dominique Colnet (concepteur du compilateur SmallEiffel). Université Henri Poincaré Nancy. Consulté le 24 novembre 2003.

`http://www.loria.fr/~colnet/publis/index-fr.html`

[25] Site de D. Lacroix. Généralités sur la compilation de langages objets. Consulté le 24 novembre 2003.

`http://lab.erasme.org/compilation_objet/index.html`

[26] Site de la compagnie Sun. Spécification de la machne virtuelle Java. Consulté le 28 juin 2006.

`http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

[27] Cours de compilation de C. Paulin-Mohring et M. Pouzet. Université de Paris Sud. Consulté le 18 août 2006.

`http://www.lri.fr/~pouzet/COMPIL/`

[28] Cours de compilation d'O. Ridoux. Université de Rennes 1. Consulté le 16 avril 2007.

`http://www.irisa.fr/lande/ridoux/ENS/COMP/compilation.pdf`

# Table des matières