5
Traduction et production de code intermédiaire.

1 – Généralités.

Une traduction consiste à transformer *un programme source*, écrit dans un langage de programmation, en *un programme cible*, écrit en code machine, qui a la même "signification" que le programme initial.

L'analyse lexicale et l'analyse syntaxique se contentent de déterminer la forme d'un programme, qui est purement locale, c'est–à–dire indifférente au contexte. Au contraire, la signification d'un programme est très globale : par exemple, un identificateur peut apparaître en plusieurs endroits d'un même programme, dans des rôles quelquefois identiques et quelquefois différents.

Cette part de la description d'un langage de programmation, qui ne peut pas être faite par le seul moyen de la grammaire, constitue ce que l'on peut appeler *la sémantique* du langage en question. L'analyse sémantique qui en découle est une phase importante de la traduction.

On oppose traditionnellement *la syntaxe* et *la sémantique* d'un programme mais en réalité, les deux points de vue sont intimement liés :

- La syntaxe est définie par une grammaire qui, en particulier, est insensible au contexte ; on sait qu'une grammaire est capable de définir des mots (des "phrases") qui n'ont pas de sens *a priori*.
- La sémantique doit impliquer un sens "effectif". Pour cela, il faut interpréter les symboles de la grammaire en question :
  - certains d'entre eux sont interprétés comme des ensembles, que l'on a coutume d'appeler *types*,
  - d'autres sont interprétés comme des relations (ou des opérations) entre les éléments de ces ensembles,
  - d'autres enfin, jouent des rôles purement syntaxiques (les séparateurs, ...).

  Toutes ces données doivent satisfaire des contraintes qui n'étaient pas exprimables par le moyen d'une simple grammaire, parce qu'elles mettent le contexte en jeu ! Par exemple, le sens à attribuer à une occurrence d'un "identificateur" peut dépendre de l'emplacement où elle se trouve dans le programme.

Nous considérons ici une forme de code dite *intermédiaire* (dans certains cas, on parle aussi de code *objet*) qui s'exprime dans un langage dont les instructions et la structure sont extrêmement simples, mais qui n'est pas encore du code machine. C'est sur ce code que l'on peut pratiquer diverses opérations regroupées sous le vocable d'*optimisation* : certaines sont faites en fonction

du langage source, d'autres, au contraire, tentent de profiter des particularités du langage cible, c'est–à–dire de la machine sur laquelle on compte exécuter le programme en question.

Un tel code se présente sous la forme d'une liste de lignes, comportant chacune une instruction élémentaire et, éventuellement une étiquette. On peut citer rapidement la forme générale des instructions de trois types de codes intermédiaires (OP désigne un opérateur) :

- Code à trois adresses : $OP\ x\ y\ z$.
  OP opère sur les trois arguments x, y et z.
  Par exemple : $ADD\ x\ y\ z$, que dans la suite on écrira plus explicitement $z := x + y$, est l'affectation à z de la somme des valeurs des variables x et y.
  Ce type de code, très immédiat à concevoir et à manipuler, est utilisé dans toute la suite.
- Code à registres : OP *source destination*.
  OP opère sur le contenu des registres *source* et *destination* et range le résultat dans le registre *destination*.
  Par exemple : $ADD\ R_1\ R_2$ représente l'affectation $R_2 := R_2 + R_1$.
  Ce type de code est très utilisé dans les langages d'assemblage.
- P–code ou "code à pile" : OP.
  OP opère sur les éléments qui sont au sommet d'une pile et empile le résultat à leur place.
  Par exemple : ADD représente la suite d'instructions

```
PILE[SP - 1] := PILE[SP - 1] + PILE[SP] ;
SP := SP - 1 :
```

  où `SP` désigne le sommet de `PILE`.
  Ce type de code n'est plus utilisé en tant que tel, mais sa compréhension est nécessaire pour bien saisir le fonctionnement de la pile d'exécution (cf. section 5 ci–dessous) : des détails relatifs à un P–code sont donnés dans les exercices.

Parmi ces instructions, les branchements nécessitent l'adressage des lignes du code ; ceci se fait par le moyen de l'indice de la ligne ou par l'usage d'étiquettes "symboliques".

La propagation des informations sémantiques doit éviter toute circularité : la conception d'un langage nécessite la satisfaction d'une telle condition (qui relève de la théorie des graphes). Lorsque la propagation des informations sémantiques est trop complexe pour permettre leur calcul en "une seule passe" (c'est le cas lorsque cette propagation ne s'effectue pas simplement dans le même sens que l'analyse syntaxique) on peut être amené à traduire le programme sous la forme d'un **arbre abstrait** ou mieux, d'un **graphe orienté acyclique*** ; une telle transformation est utilisable dans la mesure où l'arbre obtenu est équipé des pointeurs nécessaires à une circulation descendante et ascendante. Une telle représentation peut aussi être considérée comme un code intermédiaire.

L'utilisation de Yacc (ou d'un autre ruminant comme Bison ou Gnu), qui est un programme très puissant pour spécifier et engendrer des programmes d'analyse ascendante (en fait, des analyses du type $LALR(1)$, modifiées de façon à pouvoir traiter des grammaires dont les ambiguïtés sont résolubles par des conditions de précédence) permet d'effectuer facilement le calcul des informations dont la propagation est ascendante.

1.1 – Sémantique : attributs et règles.

Les informations qui viennent d'être évoquées sont plus précisément appelées des *attributs sémantiques* et leur mode de propagation est défini par des *règles sémantiques*.

* On utilise couramment le sigle DAG de l'expression angloricaine "directed acyclic graph".

Attributs sémantiques.

Chaque symbole d'une grammaire $G = (\mathcal{V}, \mathcal{A}, R)$ représente, non pas un objet isolé mais, un ensemble homogène. Par exemple :

- Chaque symbole terminal $x \in \mathcal{A}$ est censé représenter une *unité lexicale*, c'est–à–dire, un langage régulier. Par exemple :
  id est l'unité lexicale des identificateurs,
  nb est l'unité lexicale des nombres,
  oprel $= \{" = ", " < ", " \leq ", \ldots\}$ est l'unité lexicale des opérateurs de relation.
- Chaque symbole non terminal $X \in \mathcal{V}$ est censé représenter une "*unité syntaxique*" ; c'est le langage algébrique que l'on obtient en remplaçant dans $\mathcal{L}(G, X)$, chaque terminal par son unité lexicale :
  si G est la grammaire d'un langage de programmation et si Pr est la variable choisie comme axiome, l'unité syntaxique de Pr est celle des programmes écrits dans ce langage ; de même, on aura les unités syntaxiques des expressions, des instructions, ...

On attribue un *type* $\mathbf{t}\xi$ à chaque symbole $\xi \in \mathcal{A} + \mathcal{V}$: tout élément de l'unité définie par ξ est maintenant défini comme un objet de type $\mathbf{t}\xi$.

- Lorsque $\mathbf{t}\xi$ est un enregistrement, ses champs sont appelés *les attributs (sémantiques) de* ξ.
- Chaque occurrence de ξ dans un arbre d'analyse représente un élément A de l'unité de ξ : si `attr` est un attribut de ξ, on notera ξ`.attr` au lieu de `A.attr`.

Dans la pratique, cette convention d'écriture est très commode mais il est nécessaire de distinguer explicitement les diverses occurrences d'un symbole figurant dans un arbre se dérivation : ceci se fait par une indexation plus ou moins systématique (voir ci–dessous la méthode utilisée par Yacc).

Règles sémantiques.

On associe *une règle sémantique* à toute règle $\varrho : X \to \xi_1 \ldots \xi_m$ de la grammaire G : une telle règle sémantique est une instruction composée qui opère sur les attributs de X et ceux des $\xi_1 \ldots \xi_m$. On peut regrouper les instructions composant une règle sémantique en deux catégories principales :

(1) `b := f(a1, ..., an)` pour une fonction `f`
(2) `P(a1, ..., an)` pour une procédure `P`

où `b` et `a1, ..., an` sont de la forme X`.attr` ou ξ_i`.attr`$_i$.

- Une instruction (1) correspond à une *propagation* d'attributs : les valeurs de `a1, ..., an` permettent de calculer celle de `b`. Lorsque les `a1, ..., an` sont tous de la forme ξ_i`.attr`$_i$ et `b` de la forme X`.attr`, on dit que `b` est *synthétisé* : la propagation ascendante des attributs synthétisés est particulièrement adaptée à une analyse syntaxique ascendante. Dans les autres cas, où l'on parle d'attributs *hérités* : on parle d'attribut *hérité à gauche* lorsque `b` est ξ_i`.attr`$_i$ et les `a1, ..., an` de la forme X`.attr` ou ξ_j`.attr`$_j$ pour $j < i$ (un héritage à gauche est plus facile à recevoir qu'un héritage quelconque puisqu'on lit le programme source de gauche à droite).
- Une instruction (2) peut remplir des tâches très diverses : classement ou recherche d'un identificateur dans une table de symboles, impression d'une partie de code, émission d'un message d'erreur, ...

Méthode utilisée par Yacc.

Les éléments de la pile d'une analyse du type LR ne sont pas les symboles de la grammaire mais des objets du type correspondant.

- Lorsque l'on s'apprête à effectuer la réduction par une règle $\varrho : X \to \xi_1 \ldots \xi_m$, des objets de types respectifs $\mathbf{t}\xi_1, \ldots, \mathbf{t}\xi_m$ se trouvent donc placés au sommet de la pile : Yacc les désigne respectivement par $\$1, \ldots, \$m$.
- La réduction a pour effet de dépiler les objets précédents pour empiler un objet de type $\mathbf{t}X$: Yacc le désigne par $$.

(Dans toute la suite, nous simulons cette méthode en indexant des occurrences de symboles lorsque c'est utile.)

Par cette méthode, Yacc est évidemment capable de traiter de façon naturelle le cas des attributs synthétisés. Des techniques adaptées doivent être mises en œuvre pour traiter les autres cas !

Vous trouverez en annexe une petite liste d'attributs et de primitives (fonctions, procédures, variables, ...) dont il est fait usage dans le cours de ce chapitre.

Dans ce qui suit, nous donnons des exemples simples mais importants de traductions. Il s'agit essentiellement d'illustrer les notions d'attributs et de règles sémantiques sur des cas très courants. Les règles sémantiques proposées sont incomplètes en général, car nous voulons insister sur ce qui est le plus typique de la partie de la grammaire qui est traduite : les versions utilisables doivent comporter des recherches et des vérifications qu'il serait fastidieux de préciser ici.

2 – Emission directe du code.

On souhaite émettre directement le code ligne par ligne, grâce à une procédure d'impression `emettre`.

La variable globale `Indice` (en fait, c'est plus souvent la valeur d'une fonction renvoyant la position d'un pointeur de fichier), qui a comme valeur le numéro de la prochaine ligne de code, est incrémentée par chaque appel d'`emettre` : ceci suppose évidemment que toutes les lignes du code ont la même taille (un nombre entier de mots).

La question du flux de contrôle se posera nécessairement lors de la traduction des instructions "conditionnelles".

L'adresse de destination d'un branchement n'est en général pas encore connue lorsque l'on écrit celui–ci : la méthode de *reprise* consiste à faire la liste des branchements vers une même destination encore inconnue et de procéder à une seconde passe très limitée pour les informer de cette adresse dès qu'elle est déterminée.

2.1 – Expressions arithmétiques.

$E \to E_1 \oplus E_2$

```
{ E.nom := Ntemporaire ;
emettre( E.nom ' :=' E1.nom '+' E2.nom ) ;
}
```

$E \to E_1 - E_2$

```
{ E.nom := Ntemporaire ;
emettre( E.nom ' :=' E1.nom '-' E2.nom ) ;
}
```

$E \to E_1 * E_2$

```
{ E.nom := Ntemporaire ;
emettre( E.nom ' :=' E1.nom '*' E2.nom ) ;
}
```

$E \to (E_1)$

```
{ E.nom := E1.nom ;
}
```

$E \to -E_1$

```
{ E.nom := Ntemporaire ;
emettre( E.nom ' :=' 'moinsU' E1.nom ) ;
}
```

$E \rightarrow$ **id**
```
{ E.nom := id.nom ;
 }
```

$E \rightarrow$ **nb**
```
{ E.nom := Ntemporaire ;
emettre( E.nom ' :=' nb.val ) ;
 }
```

2.2 – Expressions booléennes.

Il est possible d'évaluer les expressions booléennes à la façon des expressions arithmétiques : seules quelques règles sont propres aux expressions booléennes, dont la traduction nécessite des instructions de branchement. Rappelons que la variable globale `Indice` contient l'adresse de la ligne de code où la procédure `emettre( x )` doit écrire l'instruction `x` et est donc incrémentée à chaque appel de cette procédure.

$E \rightarrow$ **id**$_1$ **oprel id**$_2$
```
{ E.nom := Ntemporaire ;
emettre( 'si' id1.nom oprel.op id2.nom
            'aller à' Indice + 3 ) ;
emettre( E.nom ' := 0' ) ;
emettre( 'aller à' Indice + 2 ) ;
emettre( E.nom ' := 1' ) ;
 }
```

$E \rightarrow$ **vrai**
```
{ E.nom := Ntemporaire ;
emettre( E.nom ' := 1' ) ;
 }
```

$E \rightarrow$ **faux**
```
{ E.nom := Ntemporaire ;
emettre( E.nom ' := 0' ) ;
 }
```

où on a pris `0` et `1` comme valeurs booléennes.

2.3 – Méthode de reprise.

Cette méthode permet d'adresser des branchements *a posteriori* :

- lorsqu'une instruction de branchement est émise avant que sa destination soit connue, on adjoint son adresse à la liste de celles qui doivent avoir la même destination. Une telle liste `l` est un attribut, que l'on désigne en fonction des circonstances, de l'une des façons suivantes :
  - `ls` : liste de sorties,
  - `lv` : liste de branchements pour une valeur `vraie`,
  - `lf` : liste de branchements pour une valeur `fausse`.
- lorsque la destination `i` est enfin déterminée, une procédure `reprendre( l, i )` complète par `i` les instructions de branchement dont les adresses sont les éléments de `l`.

Ces listes permettent d'appliquer le principe d'une évaluation partielle des expressions booléennes, connue sous le nom évocateur de "court–circuit". Elles sont constituées sur la base d'un attribut `indice` qui est l'adresse d'une ligne du code.

Marquage.

Enfin, il est nécessaire d'introduire des variables et des règles auxiliaires à la grammaire. Plus précisément, le *marquage* d'un terminal $\mathbf{x} \in \mathcal{A}$ consiste en l'introduction :

- d'une nouvelle variable, qui sera notée $\langle \mathbf{x} \rangle$, et qui est substituable à $\mathbf{x}$ partout où on le désire,
- d'une nouvelle règle de production $\langle \mathbf{x} \rangle \rightarrow \mathbf{x}$.

Par exemple, le marquage de **ou** transforme la règle $E \rightarrow E\,\mathbf{ou}\,E$ en

$$1 : E \rightarrow E\,\langle \mathbf{ou} \rangle\,E \qquad 2 : \langle \mathbf{ou} \rangle \rightarrow \mathbf{ou}$$

Lors d'une analyse ascendante, on ne connaît **ou** qu'au moment de la réduction par la règle initiale alors qu'après le marquage, une réduction par 2 : doit se produire avant une réduction par 1 : et ceci permet d'appliquer une règle sémantique propre à **ou** avant d'appliquer celle de la règle 1 :. Voici les règles sémantiques, pour les espressions booléennes et les instructions, qui découlent de ces principes.

$E \rightarrow E_1\ \langle \mathbf{ou} \rangle\ E_2$

```
        { reprendre( E1.lf, ⟨ou⟩.indice ) ;
E.lv := concatener( E1.lv, E2.lv ) ;
E.lf := E2.lf ;
        }
```

$\langle \mathbf{ou} \rangle \rightarrow \mathbf{ou}$

```
        { ⟨ou⟩.indice := Indice ;
        }
```

$E \rightarrow E_1\ \langle \mathbf{et} \rangle\ E_2$

```
        { reprendre( E1.lv, ⟨et⟩.indice ;
E.lf := concatener( E1.lf, E2.lf ) ;
E.lv := E2.lv ;
        }
```

$\langle \mathbf{et} \rangle \rightarrow \mathbf{et}$

```
        { ⟨et⟩.indice := Indice ;
        }
```

$E \rightarrow \mathbf{non}\ E_1$

```
        { E.lv := E1.lf ;
E.lf := E1.lv ;
        }
```

$E \rightarrow (\ E_1\ )$

```
        { E.lv := E1.lv ;
E.lf := E1.lf ;
        }
```

$E \rightarrow \mathbf{id}_1\ \mathbf{oprel}\ \mathbf{id}_2$

```
        { E.lv := creerliste( Indice ) ;
emettre( 'si' id1.nom oprel.op id2.nom 'aller à' ) ;
E.lf := creerliste( Indice ) ;
emettre( 'aller à' ) ;
        }
```

$E \rightarrow \mathbf{vrai}$

```
        { E.lv := creerliste( Indice ) ;
E.lf := listevide( ) ;
emettre( 'aller à' ) ;
        }
```

$E \rightarrow$ **faux**
```
        { E.lv := listevide( ) ;
E.lf := creerliste( Indice ) ;
emettre( 'aller à' ) ;
        }
```

$I \rightarrow$ **id** $:= E$
```
        { I.ls := listevide( ) ;
emettre(id.nom ' :=' E.nom ) ;
        }
```

$Li \rightarrow I$
```
        { Li.ls := I.ls ;
        }
```

$Li \rightarrow Li_1$ $\langle ; \rangle$ I
```
        { reprendre( Li1.ls, ⟨;⟩.indice ) ;
Li.ls := I.ls ;
        }
```

$\langle ; \rangle \rightarrow ;$
```
        { ⟨;⟩.indice := Indice ;
        }
```

$I \rightarrow$ **debut** Li **fin**
```
        { I.ls := Li.ls
        }
```

$I \rightarrow$ **si** E $\langle$**alors**$\rangle$ I_1
```
        { reprendre( E.lv, ⟨alors⟩.indice ) ;
I.ls := concatener( I1.ls, E.lf ) ;
        }
```

$I \rightarrow$ **si** E $\langle$**alors**$\rangle$ I_1 $\langle$**sinon**$\rangle$ I_2
```
        { reprendre( E.lv, ⟨alors⟩.indice ) ;
reprendre( E.lf, ⟨sinon⟩.indice ) ;
I.ls := concatener( I1.ls, ⟨sinon⟩.ls, I2.ls ) ;
        }
```

$\langle$**alors**$\rangle \rightarrow$ **alors**
```
        { ⟨alors⟩.indice := Indice ;
        }
```

$\langle$**sinon**$\rangle \rightarrow$ **sinon**
```
        { ⟨sinon⟩.ls := creerliste( Indice ) ;
emettre( 'aller à' ) ;
⟨sinon⟩.indice := Indice ;
        }
```

$I \rightarrow \langle$**tantque**$\rangle$ E $\langle$**faire**$\rangle$ I_1
```
        { reprendre( I1.ls, ⟨tantque⟩.indice ) ;
reprendre(E.lv, ⟨faire⟩.indice ) ;
I.ls := E.lf ;
emettre( 'aller à' ⟨tantque⟩.indice ) ;
        }
```

$\langle$**tantque**$\rangle \rightarrow$ **tantque**
```
        { ⟨tantque⟩.indice := Indice ;
        }
```

$\langle$**faire**$\rangle \rightarrow$ **faire**
```
        { ⟨faire⟩.indice := Indice ;
        }
```

Remarques sur $E \rightarrow$ id oprel id.

• Cette règle comporte le terminal **oprel** qui est l'unité lexicale des *opérateurs de relation* {" < "," ≤ "," = ",...} : **oprel**.op désigne l'opérateur lui–même.

• Une forme plus générale de cette règle est $E \rightarrow E$ **oprel** E dont la traduction doit comporter un contôle du type des expressions : dans la pratique, un tel contrôle est presque toujours nécessaire.

Une autre solution, moins intéressante, consisterait à évaluer complètement les expressions booléennes à la manière des expression arithmétiques (on utiliserait alors les opérations bien connues sur les valeurs de vérité) : les instructions de branchement seraient alors du ressort des instructions "conditionnelles" qui en font usage.

Remarques sur les règles $E \rightarrow (E)$.

Une telle règle se trouve à la fois dans la grammaire des expressions arithmétiques et celle des expressions booléennes : il est clair que cette position n'est pas tenable et qu'il faut se contenter d'une seule d'entre elles ! Il n'est plus possible de distinguer les deux genres d'expressions : il faut alors donner tous les attributs à tout le monde et les utiliser en fonction de l'attribut `type`.

Exemple de traduction avec reprise.

L'instruction suivante est analysée par un algorithme LR (l'indexation des occurrences de variables se fait au fur et à mesure) et traduite par les règles énoncées ci–dessus : l'analyse lexicale n'est évoquée que très sommairement. La traduction est présentée dans les deux tableaux qui se suivent, ci–dessous.

```
                                                          /*I6*/
si
    a < b et a < c                                        /*E3 → E1 ⟨et⟩ E2*/
alors
    debut                                                 /*I3*/
        x := x + y ; y := y + z                           /*Li2*/
    fin
sinon
    si                                                    /*I5*/
        b < c                                             /*E10*/
    alors
        x := y + z                                        /*I4*/
```

Analyse LR	**Règles sémantiques**	**Code émis**
$E_1 \rightarrow$ "$a < b$"	`E1.lv = (100) ;`	`100 : si a < b aller à 102`
	`emettre(...) ;`	
	`E1.lf = (101) ;`	`101 : aller à 109`
	`emettre(...) ;`	
$\langle$**et**$\rangle \rightarrow$ **et**	$\langle$**et**$\rangle$`.indice = 102 ;`	`102 : si a < c aller à 104`
$E_2 \rightarrow$ "$a < c$"	`E2.lv = (102) ;`	
	`emettre(...) ;`	
	`E2.lf = (103) ;`	`103 : aller à 109`
	`emettre(...) ;`	
$E_3 \rightarrow E_1$ $\langle$**et**$\rangle$ E_2	`reprendre((100),102) ;`	`104 : t0 := x + y`
	`E3.lv = (102) ;`	
	`E3.lf = (101, 103) ;`	
$\langle$**alors**$\rangle_1 \rightarrow$ **alors**	$\langle$**alors**$\rangle$`1.indice = 104 ;`	
$E_4 \rightarrow$ "x"	`E4.nom = 'x' ;`	
$E_5 \rightarrow$ "y"	`E5.nom = 'y' ;`	
$E_6 \rightarrow E_4 \ \oplus \ E_5$	`E6.nom = 't0' ;`	
	`emettre(...) ;`	
$I_1 \rightarrow$ "x" $:=$ E_6	`I1.ls = ( ) ;`	`105 : x := t0`
	`emettre(...) ;`	

Analyse LR	**Règles sémantiques**	**Code émis**
$Li_1 \rightarrow I_1$	`Li1.ls = () ;`	`106 : t1 := y + z`
$\langle ; \rangle \rightarrow$;	$\langle ; \rangle$`.indice = 106 ;`	
$E_7 \rightarrow$ "y"	`E7.nom = 'y' ;`	
$E_8 \rightarrow$ "z"	`E8.nom = 'z' ;`	
$E_9 \rightarrow E_7 \ \oplus \ E_8$	`E9.nom = 't1' ;`	
	`emettre(...) ;`	
$I_2 \rightarrow$ "y" $:=$ E_9;	`I2.ls = ( ) ;`	`107 : y := t1`
	`emettre(...) ;`	
$Li_2 \rightarrow Li_1 \ \langle ; \rangle \ I_2$	`reprendre(( ),106) ;`	`108 : aller à`
	`Li2.ls = ( ) ;`	
$I_3 \rightarrow$ **debut** Li_2 **fin**	`I3.ls = ( ) ;`	
$\langle$**sinon**$\rangle \rightarrow$ **sinon**	$\langle$**sinon**$\rangle$`.ls = (108) ;`	
	`emettre(...) ;`	
	$\langle$**sinon**$\rangle$`.indice = 109 ;`	`109 : si b < c aller à 111`
$E_{10} \rightarrow$ "$b < c$"	`E10.lv = (109) ;`	
	`emettre(...) ;`	
	`E10.lf = (110) ;`	`110 : aller à`
	`emettre(...) ;`	
$\langle$**alors**$\rangle_2 \rightarrow$ **alors**	$\langle$**alors**$\rangle$`2.indice = 111 ;`	`111 : t2 := y + z`
$E_{11} \rightarrow$ "y"	`E11.nom = 'y' ;`	
$E_{12} \rightarrow$ "z"	`E12.nom = 'z' ;`	
$E_{13} \rightarrow E_{11} \ \oplus \ E_{12}$	`E13.nom = 't2' ;`	
	`emettre(...) ;`	
$I_4 \rightarrow$ "x" $:=$ E_{13}	`I4.ls = ( ) ;`	`112 : x := t2`
	`emettre(...) ;`	
$I_5 \rightarrow$ **si** $E_{10}\langle$**alors**$\rangle_2 \ I_4$	`reprendre((109),111) ;`	`113 :`
	`I5.ls = (110) ;`	
$I_6 \rightarrow$ **si** $E_3 \ \langle$**alors**$\rangle_1 \ I_3$ $\langle$**sinon**$\rangle \ I_5$	`reprendre((102),104) ;`	
	`reprendre((101,103),109) ;`	
	`I6.ls = (108,110) ;`	

Les adresses de branchement qui ne sont pas encore déterminées à la fin de la traduction sont les éléments de la liste $I_6.ls = (108, 110)$.

Suite et fin de l'exemple de traduction avec reprise.

3 – Attribut `code` et flux de contrôle.

Dans cette autre méthode, le code nécessaire à l'évaluation d'une expression arithmétique est produit "par morceaux" : ces morceaux sont concaténés au fur et à mesure des besoins pour produire le code évaluant une expression composée. Les branchements utilisés dans la traduction des expressions booléennes et des instructions utilisent des étiquettes symboliques qui peuvent être remplacées par la mention explicite d'indices de lignes de code. La traduction qui en découle est assez facile à concevoir mais sa réalisation peut poser quelques problèmes : le code n'est pas produit de façon incrémentale, certains attributs sont hérités, les étiquettes prolifèrent et peuvent même s'accumuler , ... (la méthode de "reprise" évite ces inconvénients !).

Le code produit est constitué d'une liste de commandes qui ont la forme

`etiquette : instruction.`

Les procédures `etiquette( x )` et `instruction( x )` adjoignent au code le premier champ libre qui les concerne ; la fonction `Netiquette()` renvoie un nom nouveau d'étiquette (une chaîne de caractères dans le style `l0`, `l1`, `l2`, ...).

Dans les règles qui suivent, la définition du `code` doit se comprendre comme celle d'une liste d'éléments à deux champs :

- le champ "étiquette", qui peut contenir une étiquette, jouant le rôle d'une adresse symbolique,
- le champ "instruction", qui peut contenir une véritable instruction.

3.1 – Expressions arithmétiques.

$E \to E_1 \oplus E_2$

```
        { E.nom := Ntemporaire ;
E.code := E1.code
E2.code
instruction(E.nom ' :=' E1.nom '+' E2.nom) ;
         }
```

$E \to E_1 - E_2$

```
        { E.nom := Ntemporaire ;
E.code := E1.code
E2.code
instruction(E.nom ' :=' E1.nom '-' E2.nom) ;
         }
```

$E \to E_1 * E_2$

```
        { E.nom := Ntemporaire ;
E.code := E1.code
E2.code
instruction(E.nom ' :=' E1.nom '*' E2.nom) ;
         }
```

$E \to (E_1)$

```
        { E.nom := E1.nom ;
E.code := E1.code ;
         }
```

$E \to -E_1$

```
        { E.nom := Ntemporaire ;
E.code := E1.code
instruction(E.nom ' :=' 'moinsU' E1.nom) ;
         }
```

$E \rightarrow$ **id**
```
        { E.nom := id.nom ;
E.code := '' ;
        }
```

$E \rightarrow$ **nb**
```
        { E.nom := Ntemporaire ;
E.code := instruction(E.nom ' :=' nb.val) ;
        }
```

3.2 – Expressions booléennes.

Il est naturel d'attribuer `vrai` et `faux` à des expresions booléennes : ce sont ici des étiquettes qui permettent des branchements, nécessaires aux instructions "conditionnelles". La traduction qui suit est encore basée sur le principe d'une évaluation partielle des expressions booléennes, en "court–circuit".

$E \rightarrow E_1$ **ou** E_2
```
        { E1.vrai := E.vrai ;
E1.faux := Netiquette ;
E2.vrai := E.vrai ;
E2.faux := E.faux ;
E.code := E1.code
etiquette( E1.faux )
E2.code ;
        }
```

$E \rightarrow E_1$ **et** E_2
```
        { E1.vrai := Netiquette ;
E1.faux := E.faux ;
E2.vrai := E.vrai ;
E2.faux := E.faux ;
E.code := E1.code
etiquette( E1.vrai )
E2.code ;
        }
```

$E \rightarrow$ **non** E_1
```
        { E1.vrai := E.faux ;
E1.faux := E.vrai ;
E.code := E1.code ;
        }
```

$E \rightarrow (E_1)$
```
        { E1.vrai := E.vrai ;
E1.faux := E.faux ;
E.code := E1.code ;
        }
```

$E \rightarrow$ **id**$_1$ **oprel id**$_2$
```
        { E.code :=
instruction( 'si' id1.nom oprel.op id2.nom
             'aller à' E.vrai )
instruction( 'aller à' E.faux ) ;
        }
```

$E \rightarrow$ **vrai**
```
        { E.code := instruction( 'aller à' E.vrai ) ;
        }
```

$E \rightarrow$ **faux** { E.code := instruction('aller à' E.faux) ; }

3.3 – Instructions.

L'attribut `debut` est une étiquette que l'on peut coller au début du code traduisant une instruction, `suivant` est une étiquette que l'on colle à la ligne de code qui suit immédiatement le code de l'instruction à laquelle elle est attribuée : la prolifération des étiquettes est assez remarquable.
(On ne considère que des instructions non vides : tous les ";" doivent être utiles !)

$I \rightarrow$ **id** $:= E$
```
{ I.suivant := Netiquette ;
I.code := E.code
instruction(id.nom ' :=' E.nom) ;
}
```

$Li \rightarrow I$
```
{ Li.suivant := I.suivant ;
Li.code := I.code ;
}
```

$Li \rightarrow Li_1$; I
```
{ Li.suivant := I.suivant ;
Li.code := Li1.code
etiquette( Li1.suivant ) I.code ;
}
```

$I \rightarrow$ **debut** Li **fin**
```
{ I.suivant := Li.suivant ;
I.code := Li.code
etiquette( Li.suivant ) ;
}
```

$I \rightarrow$ **si** E **alors** I_1
```
{ E.vrai := Netiquette ;
E.faux := I.suivant ;
I1.suivant := I.suivant ;
I.code := E.code
etiquette( E.vrai ) I1.code ;
}
```

$I \rightarrow$ **si** E **alors** I_1 **sinon** I_2
```
{ E.vrai := Netiquette ;
E.faux := Netiquette( ) ;
I1.suivant := I.suivant ;
I2.suivant := I.suivant ;
I.code := E.code
etiquette( E.vrai ) I1.code
instruction( 'aller à' I.suivant )
etiquette( E.faux ) I2.code ;
}
```

$I \rightarrow$ **tantque** E **faire** I_1
```
{ I.debut := Netiquette ;
E.vrai := Netiquette( ) ;
E.faux := I.suivant ;
I1.suivant := I.debut ;
I.code := etiquette( I.debut ) E.code
etiquette( E.vrai ) I1.code
instruction( 'aller à' I.debut ) ;
}
```

4 – Une construction de la table des symboles.

La table se présente sous la forme d'un arbre qui est *l'arbre statique* du programme analysé : chaque procédure* possède sa propre table, comportant la liste de ses références locales, constituant un nœud de cet arbre. Les classements et recherches que l'on aura à effectuer nécessitent des visites descendantes et ascendantes de cet arbre : il faut donc prévoir un double système de pointeurs.

Ces mêmes classements et recherches constituent l'une des activités principales d'un compilateur, pour cette raison, toute conception réaliste d'une table doit concilier la structure précédente avec l'efficacité de ces opérations, par exemple en utilisant une méthode d'adressage dispersé (*hashing code*).

On crée préalablement une pile de pointeurs que l'on initialise par le pointeur sur la racine de l'arbre :

```
t := creertable( nil ) ;
empiler( t ) ;
```

puis on analyse le programme défini par la grammaire simplifiée suivante (dont on n'a écrit que ce qui nous intéressent ici) :

$P \rightarrow$ **programme id** ; $Dv\ Dp\ I$ **fin** . {depiler( ) ;}

$Dv \rightarrow \varepsilon$ {
}

$Dv \rightarrow$ **var** Dv' {
}

$Dv' \rightarrow$ **id** : T { entrervar(sommet( ), **id**.nom, T.type, ...) ;
}

$Dv' \rightarrow Dv'$ **id** : T { entrervar(sommet( ), **id**.nom, T.type, ...) ;
}

$Dp \rightarrow \varepsilon$ {
}

$Dp \rightarrow Dp\ R$ {
}

$R \rightarrow U\ A$; $Dv\ Dp\ I$ **fin** ; { depiler( ) ;
}

$U \rightarrow$ **procedure id** { t := creertable(sommet( )) ;
entrerprocedure(sommet( ), **id**.nom, t, ...) ;
empiler(t) ;
}

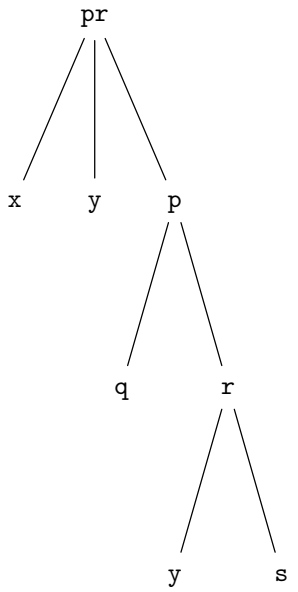
* *Le mot "procédure" est utilisé génériquement pour désigner une procédure, le programme lui–même ou une fonction.*

$A \rightarrow \varepsilon$	{ }
$A \rightarrow (\ A' \ )$	{ }
$A' \rightarrow$ **id** : T	{ entrerpara(sommet( ), **id**.nom, T.type, ...) ; }
$A' \rightarrow A'$; **id** : T	{ entrerpara(sommet( ), **id**.nom, T.type, ...) ; }
$T \rightarrow$ Types	{ ... }
$I \rightarrow$ Instructions	{ ... }

La table des symboles.

La table des symboles, construite par l'application des règles sémantiques est une représentation détaillée des diverses déclarations qui sont faites dans un programme, sous la forme d'un arbre. Une représentation simplifiée sous la forme d'un "arbre abstrait" facilitera la compréhension des calculs qui seront faits à la fin de ce chapitre.

```
Programme pr ;
    var x : Tx y : Ty1
    procedure p ;
        procedure q ;
            ... y ... r
        fin ;
        procedure r ;
            var y : Ty2
            procedure s ;
                ... x ... y ... q
            fin ;
            ... y ... s
        fin ;
        r
    fin ;
    p
fin.
```



Un exemple de programme et son arbre abstrait.

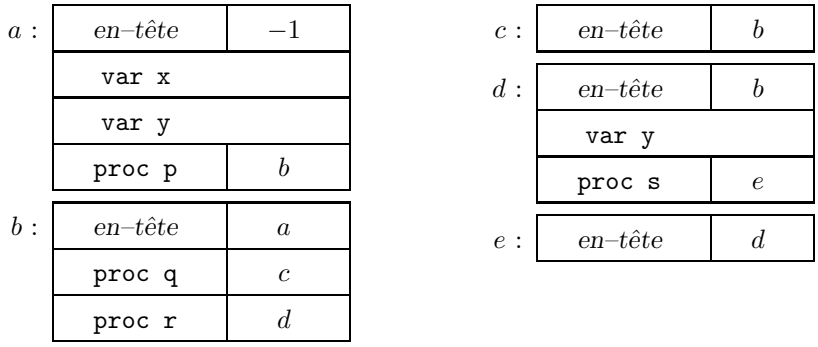
Revenons à la table des symboles elle–même. Chacun de ses nœuds est la "table propre" à une procédure ; les noms qui sont enregistrés dans la table propre d'une procédure P sont génériquement

appelés les *références locales* à `P` : ce sont les noms de ses paramètres, de ses variables locales et de ses procédures locales.

Une table propre à une procédure est caractérisée par son adresse (dans la figure ci–dessous, les adresses sont représentées comme des étiquettes). Elle contient :

- pour chacun de ses paramètres (et chacune de ses variables locales) : un champ contenant les attributs (`nom`, `type`, ...) de celui–ci, utile à la traduction,
- pour chaque procédure qui lui est locale : le nom et l'adresse de la table propre de celle–ci,
- une en–tête contenant les attributs de la procédure qui sont utiles à la traduction (par exemple, le type de retour dans le cas d'une fonction, le nombre de paramètres, mais aussi, le degré d'imbrication, l'indice de la première ligne du code code de cette procédure, ...) et l'adresse de la table de la procédure dans laquelle elle vient d'être déclarée (antécédent "statique" immédiat)

En fait, beaucoup de données statiques, c'est–à–dire calculables au moment de la traduction, prendront place dans la table des symboles, dont nous n'avons donné ici qu'une description sommaire.



La table des symboles du programme `pr`

5 – Enregistrements d'activation et pile d'exécution.

Comme dans la section précédente, le terme *procédure* sera utilisé génériquement pour désigner une procédure, le programme principal ou une fonction.

La traduction des procédures dépend très directement de la façon dont on exécutera le code qu'elle produit (ce que l'on appelle souvent *l'environnement d'exécution*) : nous décrivons ici les grandes lignes de *l'allocation en pile*, qui est la plus courante.

Chaque appel d'une procédure détermine l'empilement d'un *enregistrement d'activation* (EA en abrégé) sur une pile dite *pile d'exécution*. La structure d'un EA, qui peut être très complexe, comporte toujours deux genres de *blocs*[*]

• Les blocs propres, qui ne dépendent que de la procédure en question, c'est–à–dire les éléments nécessaires à l'empilement des données locales (paramètres, variables locales et éventuellement, valeur de retour) et, éventuelement, les éléments nécessaires à l'empilement des données temporaires.

Remarques :

- Lorsqu'un EA est activé, il est naturellement placé au sommet de la pile d'exécution ; l'espace qui suit est donc libre et peut servir pour l'exécution de calculs intermédiaires ou préparatoires à de futurs appels.

[*] Un bloc est un empilement contigu d'éléments.

– La détermination du partage des tâches entre la procédure appelante et la procédure appelée est un sujet important. Si, par exemple, il revient à l'appelante d'évaluer les paramètres de l'appelée, ces valeurs sont déjà empilées au moment de l'appel et une partie de l'EA de l'appelée est donc en place.
– Lorsqu'un EA est désactivé, il est dépilé de la pile d'exécution. Dans le cas d'une fonction, la valeur de retour peut alors être empilée : ceci est cohérent avec les possibilités évoquées dans les deux remarques qui précèdent.

Un "registre" SP (*Stack Pointer*) contient l'adresse du sommet de la pile.

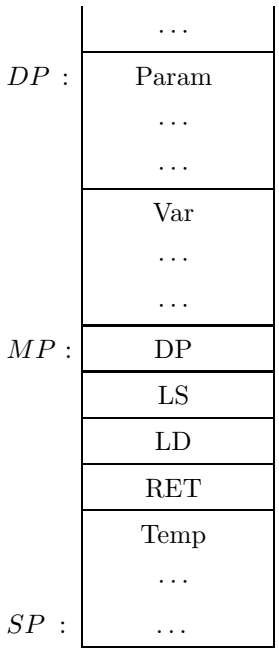
• Le bloc de sauvegarde, qui dépend du mode d'appel et de retour, contient les données nécessaires pour l'exécution ou lors de la désactivation de cet EA (le retour à la procédure appelante). Un de ces éléments est **choisi comme base de l'EA** ; son adresse dans la pile d'exécution est contenue dans un "registre" que nous noterons MP (*Mark Pointer*) : les autres éléments du bloc de sauvegarde occupent une position déterminée par rapport à l'élément de base.

Parmi les éléments à sauvegarder, citons l'adresse du début de l'EA (nous la noterons DP pour *Data Pointer*, car nous l'utiliserons pour marquer le début des données locales) ainsi que LS, LD et RET qui sont décrits à coté de la figure donnant un exemple d'EA.

LD : *Le lien dynamique*, c'est–à–dire la valeur qu'avait MP pour l'EA de la procédure appelante. Ce lien servira lors du retour à ladite procédure appelante. Lorsque MP ne marque pas le début d'un EA, la valeur de DP joue aussi un rôle dans cette opération.

LS : *Le lien statique*, c'est–à–dire la valeur qu'avait MP pour l'EA le plus récent du prédécesseur statique de la procédure en question, dont la définition et le rôle sont les sujets principaux de cette section. Ce lien servira à accéder aux variables qui ne sont pas locales à la procédure en question.

RET : *L'adresse de retour*, c'est–à–dire celle de la première ligne de code à exécuter après l'achèvement du présent appel.



Sauvegardes et exemple d'EA.

Remarque générale.

Les éléments nécessaires à la construction d'un EA (relatifs par exemple aux variables et aux variables temporaires) sont à rechercher dans la table des symboles par le truchement de leur nom, mais cette table ne survit pas à la compilation d'un programme : les noms n'ont donc plus aucune signification au moment de l'exécution ! Tout ce qui suit tend à remplacer chaque nom par une information d'adressage numérique qui pourra jouer le même rôle mais sera effectivement utilisable pendant l'exécution.

5.1 – Noms locaux à une procédure, déplacement.

Soit `P` le nom d'une procédure alors, *loc*(`P`) désigne la liste des références locales à `P` (les noms qui figurent dans sa table propre, c'est–à–dire les noms de ses descendants immédiats dans l'arbre abstrait).

Pour accéder au début d'une donnée `X` locale à `P` dans un EA de cette procédure, il suffit de connaître *le déplacement* qu'il est nécessaire d'effectuer, depuis le début de ces données jusqu'au début de `X`; ce déplacement est un attribut qu'il est facile de calculer, en fonction de l'attribut taille :

Soit $x_0, \ldots, x_m$ la liste des données locales à `P`, c'est–à–dire de ses paramètres et de ses variables locales. Chaque x_i a une taille qui est déterminée par son type : c'est la longueur de l'espace mémoire nécessaire à l'enregistrement d'une valeur de x_i. On peut alors définir :

$$depl(x_0) = 0$$
$$depl(x_{i+1}) = depl(x_i) + taille(x_i).$$

Les attributs **id.**`taille` et **id.**`depl` sont enregistrés dans la table des symboles par la procédure `entrervar` et par son analogue `entrerpara`.

Lorsqu'une procédure `P` utilise un nom `X` qui ne lui est pas local il faut trouver, dans la pile d'exécution, un EA convenable pour une procédure où `X` est locale : c'est l'objet de ce qui suit.

5.2 – La portée statique et le lien statique.

Une procédure doit pouvoir faire appel à des noms qui ne lui sont pas locaux : *la portée d'un nom* est donc une notion essentielle. Nous n'en considérerons que la version statique qui est déterminée par la structure d'arbre que nous avons donnée à la table des symboles : celle–ci est calculable au moment de la compilation, contrairement à la notion de portée dynamique qui est déterminée, de la même façon mais, par l'arbre d'exécution.

Nous utiliserons très souvent l'expression "nom" pour désigner une occurrence particulière de ce nom dans la table des symboles d'un programme donné.

Références possibles pour une procédure.

Une procédure peut se référer aux noms qui lui sont locaux et à ceux qui lui sont "visibles".

Lorsque chaque nom ne figure qu'une seule fois dans la table, la visibilité est définie de la façon suivante :

- *Un élément de loc*(`X`) *est visible par les descendants (immédiats ou non) de* `X`. Mais en général, plusieurs noms de la table des symboles peuvent être identiques et alors, un nom peut en cacher un autre et le rendre invisible :

Un nom cache tout nom identique par lequel il est visible.

Donnons tout d'abord la définition d'un attribut fondamental :

Le degré d'imbrication d'un nom `P` est la longueur d(`P`) du chemin qui le relie à la racine.

Il est clair que :
- d(`P`) = 0 ssi `P` est le nom du programme principal.
- si `X` $\in$ *loc*(`P`) alors d(`X`) = d(`P`) + 1.

On peut maintenant définir les références éventuelles (visibles ou non) Ref(`P`) de la procédure `P` de la façon suivante :

Soit `P` = $\mathtt{P}_0, \mathtt{P}_1, \ldots, \mathtt{P}_n$ = `Pr` le chemin qui relie `P` au programme principal `Pr` (avec $n = d$(`P`)) et posons Ref_i(`P`) = *loc*($\mathtt{P}_i$) pour chaque i, alors :

Ref(`P`) est réunion de tous ces Ref_i(`P`).

Le degré d'ascendance d'un nom `X` $\in Ref$(`P`) *par rapport à* `P` est l'entier i tel que `X` $\in Ref_i$(`P`).

La primitive `rechercher( X, P )` qui détermine le degré d'ascendance `dasc` de `X` par rapport à `P` peut s écrire :

```
dasc := 0 ;
Y := P ;
tantque dasc <= d( P ) et X ∉ loc( Y ) faire
    dasc := dasc + 1 ;
    Y := ascendant immédiat de Y
fin
```

Commentaires.

- Si la valeur trouvée est $dasc = d(\mathtt{P}) + 1$ alors `X` n'est pas une référence possible pour `P`,
- sinon, notons plus précisément $dasc_{\mathtt{P}}(\mathtt{X})$ le degré d'ascendance de `X` par rapport à `P` : il est clair que l'on a

$$dasc_{\mathtt{P}}(\mathtt{X}) = d(\mathtt{P}) - d(\mathtt{X}) + 1.$$

- La primitive `rechercher` ne considère que la première occurrence de `X` qu'elle "voit" : elle satisfait donc la condition de visibilité réelle qui nous intéresse.
- Nous parlons depuis le début de la table des symboles comme d'un immeuble parfaitement achevé mais, on la construit au fur et à mesure de l'analyse des "déclarations" du programme. Or, la primitive `rechercher` est appelée à chaque fois qu'une instruction fait mention d'un identificateur : si celui–ci n'est pas dans la table au moment de sa recherche notre primitive signalera que la référence n'est pas valable, ou pire, trouvera une référence fausse. Il y a deux types de réponses à ce problème :
  1) ou bien on fait une première passe qui ne fait pas appel à `rechercher` : il faut alors en faire une seconde ... On peut aussi envisager une méthode de "reprise" analogue à celle que nous avons vue plus haut.
  2) ou bien on applique la maxime bien connue : n'utiliser un identificateur qu'après sa déclaration.

Etablissement du lien statique.

Lors d'un appel de la procédure `X` le lien statique de l'EA correspondant doit pointer sur l'EA de la dernière activation encore dans la pile d'exécution de son ascendant statique immédiat. Soit `P` la procédure qui fait cet appel : l'EA correspondant, comme tous ceux qui sont dans la pile, dispose de son lien statique

Soit $a = d(\mathtt{P}) - d(\mathtt{X}) + 1$ le degré d'ascendance de `X` par rapport à `P` et considérons le chemin $\mathtt{P} = \mathtt{P}_0, \mathtt{P}_1, \ldots, \mathtt{P}_n = \mathtt{Pr}$ qui relie `P` au nom du programme principal : par définition de a on a $\mathtt{X} \in loc(\mathtt{P}_a)$ et le lien que nous recherchons doit donc pointer sur l'EA correspondant au a–ième ascendant de `P`. On peut exprimer ceci par la recette suivante :

En partant de l'EA de `P` *qui vient de faire l'appel de* `X`*, remonter a liens statiques : l'EA ainsi pointé est celui sur lequel le lien statique de l'EA de* `X` *doit pointer.*

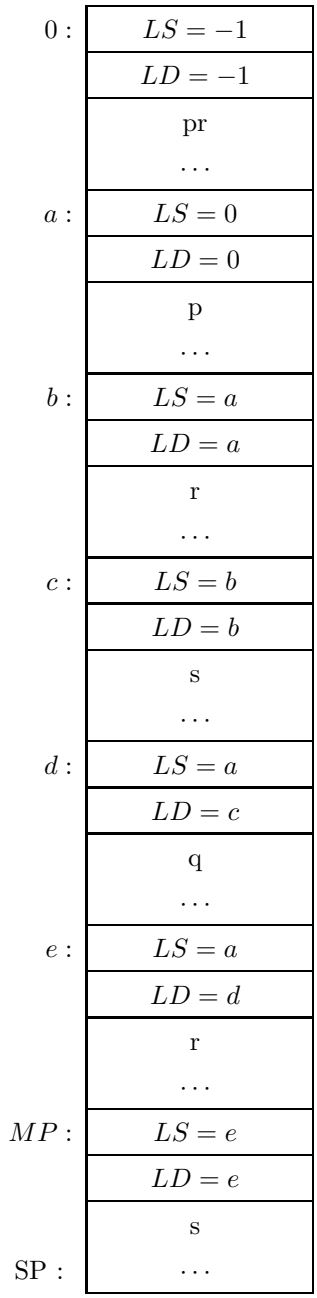
Ceci s'applique aux références à des noms de paramètres ou de variables. Lorsque `P` fait référence à **id**`.nom`, la valeur de **id**`.dasc` (relativement à `P`) est calculable à la compilation et on accèdera à l'emplacement où la donnée en cause se trouve dans la pile

- en suivant **id**`.dasc` liens statiques,
- puis en faisant le déplacement **id**`.depl` dans l'EA ainsi atteint.

Le codage de chaque référence à un tel identificateur est donc le couple

id`.dasc` : **id**`.depl`.

Afin de distinguer les éventuelles variables temporaires des autres, on pourra leur attribuer un degré d'ascendance particulier, par exemple -1.



La figure représente un état de la pile de l'exécution du programme `pr` qui nous a déjà servi d'exemple.

La représentation des EA a été simplifiée : seuls les liens statiques et dynamiques ont été figurés. Par commodité, les adresses sont représentées sous la forme d'étiquettes.

Le degré, ou le degré d'ascendance, est noté en commentaire.

```
Programme pr ;              /* 0 */
    var x : Tx y : Ty1
    procedure p ;           /* 1 */
        procedure q ;       /* 2 */
            ... y ...       /* 2 */
            r
        fin ;
        procedure r ;       /* 2 */
            var y : Ty2
            procedure s ;   /* 3
            */
                ... x ...  /* 3 */
                ... y ...  /* 1 */
                q
            fin ;
            ... y ...       /* 0 */
            s
        fin ;
        r
    fin ;
    p
fin.
```

Un état de la pile d'exécution du programme `pr`.

6 – Annexe.

Voici quelques attributs et quelques primitives qui sont utiles à nos traductions. Ces listes ne sont pas du tout exhaustives et les définitions supposent souvent la connaisance du texte qui précède.

6.1 – Attributs.

Les attributs qui sont définis ici sont principalement *statiques*, c'est–à–dire "calculables" à partir du seul programme source.

`nom` : chaîne de caractères ne commençant pas par un chiffre, qui sert à désigner un objet (procédure, fonction, paramètre formel, variable, variable temporaire, ...).

Une règle sémantique relative à une production comportant un identificateur devra comporter, pour chaque occurrence de **id**, la recherche et éventuellement le classement de **id**`.nom` dans la table des symboles. Dans les traductions qui sont données ici, nous utilisons cet attribut de façon générique. En réalité, la table des symboles cesse d'exister à la fin de la traduction : celle–ci doit donc comporter des opérations tendant à remplacer un nom par une donnée utilisable lors de l'exécution (voir les attributs `degre` et `deplacement` ci–dessous).

`degre` : c'est le degré d'imbrication (profondeur dans l'arbre statique que constitue la table des symboles). Il est nécessaire pour mettre en œuvre la notion de portée statique, c'est–à–dire pour calculer le degré d'ascendance `dasc` qui sert à installer la liaison statique dans la pile d'exécution.

`debut` : l'adresse ou l'étiquette de la première ligne du code propre à une procédure ou à une fonction.

`deplacement` : distance qu'il faut parcourir, dans l'enregistrement d'activation convenable, pour accéder au début de la donnée en cause : c'est la somme des tailles des données qui précèdent !

`val` : la valeur. C'est un attribut statique pour tout élément de l'unité lexicale **nb** (qui ne possède que cet attribut).

`type` : le type au sens habituel du terme. Sa connaissance permet de calculer la `taille`.

`code` : désigne la partie de code permettant l'évaluation de l'objet auquel il est attribué.

`indice` : cet attribut désigne l'adresse de la prochaine ligne du code. Il est utile pour les instructions de branchement.

`ls` : c'est une liste de numéros de lignes de code d'où l'on doit faire un branchement vers une ligne de code dont l'adresse ne sera déterminée qu'ultérieurement.

`lv` : idem, dans le cas où ces branchements sont déterminés par l'évaluation d'un objet booléen dont le résultat est **vrai**.

`lf` : idem, dans le cas où ces branchements sont déterminés par l'évaluation d'un objet booléen dont le résultat est **faux**.

6.2 – Primitives.

`creertable(precedent)` : fonction qui crée une nouvelle table, nécessaire lors de la déclaration d'une nouvelle procédure ou d'une nouvelle fonction et renvoie un pointeur sur celle–ci. Dans l'application que nous en faisons, ce pointeur est empilé sur une pile réservée à cet effet. L'argument `precedent` est un pointeur sur la table active au moment de l'appel.

`entrerpara(ptrtable, nom, type, ...)` : installe le `nom`, le `type`, ... d'un paramètre formel, dans la table pointée par `ptrtable` de la procédure ou de la fonction en cause.

`entrervar(ptrtable, nom, type, ...)` : idem, dans le cas d'une variable locale.

`entrertemp(ptrtable, nom, type, ...)` : idem, dans le cas d'une variable temporaire si l'on en fait usage, par exemple pour écrire un code "à trois adresses".

`entrerprocedure(ptrtable, nom, nouvelletable, ...)` : idem, dans le cas d'une procédure. Le pointeur sur la table créée lors de la déclaration d'une procédure est entregistrée dans la table de la procédure appelante

`entrerfonction(ptrtable, nom, type, nouvelletable, ...)` : idem, dans le cas d'une fonction.

`Indice` : est une variable globale désignant le numéro de la ligne de code qui suit la dernière instruction écrite (c'est donc le numéro de la ligne où viendra s'inscrire la prochaine instruction dudit code !). Sa valeur initiale est par exemple `0`.

`instruction( x )` : affecte `x` au champ `instruction` d'une ligne de code.

`etiquette( x )` : affecte `x` au champ `étiquette` d'une ligne de code.

`Netiquette` : fonction qui renvoie un nouveau nom d'étiquette (la visibilité d'une étiquette est globale : il faut donc prendre ici le qualificatif "nouveau" dans un sens strict !

`Ntemporaire` : fonction qui renvoie le nom d'une nouvelle variable temporaire (chaîne de caractères dans le style `'t0'`, `'t1'`, `'t2'`, ...). Une variable temporaire a une portée strictement locale : il faut donc prendre l'expression "nouveau" dans un sens tout aussi local.

`emettre( x )` : écrit l'instruction `x`, suivant le format correct, à la ligne de code de numéro `Indice` puis, incrémente `Indice`.

`reprendre(l, i)` : insère le numéro `i` (indice) comme destination des branchements qui se trouvent aux lignes dont les numéros sont les éléments de la liste `l`. Les listes en question sont construites par des primitives usuelles : les fonctions `listevide( )` qui renvoie une liste vide, `creerliste( i )` une liste à un seul élément `i` et `concatener( l1, l2, ...)` la concaténée des listes `l1, l2, ...`

EXERCICES.

L'objet de ces exercices est de construire, par exemple à l'aide de Lex et Yacc, un traducteur du langage *Galileo* en l'un des codes dont il a été question dans les généralités (P–code, Code à registres ou Code à trois adresses) et éventuellement, d'écrire un programme (dans votre langage favori) capable d'exécuter un tel code.

Présentation de la grammaire.

Galileo est, en français, un langage impératif très simple inspiré de Pascal, dont la sémantique est par conséquent connue de tous (deux points, inspirés de C sont signalés dans ce qui suit).

- La définition précise des unités lexicales est donnée à la fin, dans un style Lex.
- La grammaire exclut l'imbrication des procédures et des fonctions (ceci tient à l'absence de Dp dans les règles 8 et 9) ainsi, un nom ne peut–il être que local ou global : cet attribut est suffisant pour déterminer l'adresse d'une référence et le lien statique n'a pas lieu d'être explicité dans la pile d'exécution.
- L'usage du mot réservé **fin** permet d'éviter un cas classique d'ambiguïté (règles 22 et 23), plus généralement, **fin** joue le rôle d'une parenthèse fermante qui donne à la grammaire une structure très simple. Les "parenthèses ouvrantes" se présentent sous différents aspects, par exemple **Programme**, **var**, ...
- Les types autorisés sont les entiers et les tableaux d'entiers (on n'a donc pas jugé utile de mentionner **entier** dans les règles 9 et 15). En réalité, l'emploi du type tableau est limité à la définition des listes d'identificateurs (règles 4 et 5) et donc, des listes de paramètres (règle 13) et de certaines expressions. L'introduction d'une expression de type tableau ne peut se produire que par application de la règle 39, mais il n'est pas question de faire des calculs sur un tel objet (les calculs sur un tableau se font sur ses composantes !)
- Pour en finir avec les tableaux : le domaine d'un tableau de type **tableau** `[ n ]` est l'ensemble des entiers `0, ..., n - 1` comme en C.
- Comme en C, les entiers sont interprétés comme des booléens lorsque c'est nécessaire : 0 pour faux et $\neq 0$ pour vrai ; la négation transforme respectivement ces valeurs en 1 et 0.
L'évaluation des booléens se fera par un calcul complet mais les instructions de branchement provenant des règles 22, 23 et 24 nécessitent l'emploi d'une méthode de reprise : il vous faudra marquer les terminaux ";", **alors**, **sinon**, **tantque** et **faire** de façon convenable.
- **La portée est statique**.
- **Les paramètres sont tous transmis par valeur** et le langage ne comporte pas de pointeur.
- Une liste d'instructions ne peut pas être vide :
  - le retour d'une fonction se fait toujours par une instruction **retourne** (E),
  - le retour du programme ou d'une procédure se fait nécessairement par une instruction **retourne**
- Il peut arriver, pour certaines valeurs des paramètres, que l'exécution n'aboutisse à aucune instruction de retour : ceci est le fruit d'une erreur grave dans le programme mais qui n'est pas décelable pendant la traduction.
- Une autre erreur classique peut se produire dans la transmition des paramètres : lors de la traduction, il est nécessaire (et évidemment possible) de vérifier que le nombre et le type des paramètres utilisés dans un appel de fonction ou de procédure correspond bien à la déclaration de celle–ci.

La grammaire de Galileo

Programmes.

1 : $P \rightarrow$ **programme id** $Dv\ Dp\ Li$ **fin**

Déclarations de **v**ariables.

2 : $Dv \rightarrow \varepsilon$

3 : $Dv \rightarrow$ **var** Lid **fin**

Listes d'**id**entificateurs.

4 : $Lid \rightarrow$ **id** : Ty

5 : $Lid \rightarrow Lid$, **id** : Ty

Déclarations de **p**rocédures et de fonctions.

6 : $Dp \rightarrow \varepsilon$
7 : $Dp \rightarrow Dp\ R$
8 : $R \rightarrow Pro$ (Lp) $Dv\ Li$ **fin**
9 : $R \rightarrow Fon$ (Lp) $Dv\ Li$ **fin**
10 : $Pro \rightarrow$ **procedure id**
11 : $Fon \rightarrow$ **fonction id**

Listes de **p**aramètres.

12 : $Lp \rightarrow \varepsilon$

13 : $Lp \rightarrow Lid$

Types.

14 : $Ty \rightarrow$ **entier**

15 : $Ty \rightarrow$ **tableau** [**nb**]

Listes d'**i**nstructions.

16 : $Li \rightarrow I$

17 : $Li \rightarrow Li$; I

Instructions.

18 : $I \rightarrow V\ :=\ E$
19 : $I \rightarrow$ **id** (Le)
20 : $I \rightarrow$ **retourne**
21 : $I \rightarrow$ **retourne** (E)
22 : $I \rightarrow$ **si** E **alors** Li **fin**
23 : $I \rightarrow$ **si** E **alors** Li **sinon** Li **fin**
24 : $I \rightarrow$ **tantque** E **faire** Li **fin**
25 : $I \rightarrow$ **ecrire** (E)

Variables.

26 : $V \rightarrow$ **id**

27 : $V \rightarrow$ **id** [E]

Listes d'**e**xpressions.

28 : $Le \rightarrow \varepsilon$
29 : $Le \rightarrow Le'$
30 : $Le' \rightarrow E$
31 : $Le' \rightarrow Le'$, E

Expressions (expressions **S**imples, **T**ermes, **F**acteurs).

32 : $E \rightarrow S$
33 : $E \rightarrow S$ **oprel** S
34 : $S \rightarrow T$
35 : $S \rightarrow S$ **opadd** T
36 : $S \rightarrow S\ -\ T$
37 : $T \rightarrow F$
38 : $T \rightarrow T$ **opmult** F
39 : $F \rightarrow$ **id**
40 : $F \rightarrow$ **id** (Le)
41 : $F \rightarrow$ **lire** ( )
42 : $F \rightarrow$ **id** [E]
43 : $F \rightarrow$ (E)
44 : $F \rightarrow -\ F$
45 : $F \rightarrow$ **non** F
46 : $F \rightarrow$ **nb**

Les unités lexicales.

- Les mots clefs (**programme**, **var**, **procedure**, **fonction**, **entier**, **tableau**, **retourne**, **si**, **alors**, **sinon**, **tantque**, **faire**, **lire**, **ecrire**, **fin**) sont évidemment réservés et chacun d'eux peut constituer une unité lexicale à soi seul.
- **id** = $[a-zA-Z][a-zA-Z0-9]*$ est l'unité lexicale des identificateurs : des chaînes non vides de chiffres et de lettres, commençant par une lettre.
- **nb** = $[-+]?[0-9]+$ est l'unité lexicale des nombres : des chaînes non vides de chiffres, éventuellement précédées d'un signe.
- **oprel** = " = " | " <> " | " <= " | " < " | " >= " | " > "
- **opadd** = " + " | **ou**
- **opmult** = " * " | **et**
- Chacun des autres symboles (le signe moins " − ", la négation **non**, l'opérateur d'affectation " := ", les parenthèses "(", ")", "[", "]" et les séparateurs ",", ";", " : ") constitue une unité lexicale à lui seul.
- **bl** = $[\ \backslash n\backslash t]+$ est l'unité lexicale des "blancs" : des chaînes constituées d'un nombre quelconque d'espaces, de fins de lignes et de tabulations. Ceux qui figurent dans la grammaire ne sont nécessaires qu'avant et après un mot clef, sauf si un symbole (parmi ceux qui sont décrits dans les points précédents) peut jouer le même rôle.
- **comm** = "%".$*\backslash n$ est l'unité lexicale des commentaires : tout ce qui, dans une ligne, suit le symbole % est considéré comme un commentaire et doit être négligé dans l'analyse syntaxique (ceci explique que les commentaires ne figurent pas dans la grammaire précédente).

Exemple : P–code.

Ce type de code intermédiaire est intéressant car il met bien en evidence un mode très courant d'exécution d'un programme.

Un P–code est une liste de P–instructions, dont la plupart agissent directement sur la pile d'exécution, qui se présente généralement sous la forme d'un simple fichier.

- Une P–instruction comporte un opérateur (représenté par un mnémonique de quelques lettres, qui devra être codé par un entier) et éventuellement un argument (noté `p` dans les descriptions ci–dessous).
- On dispose de "registres" (qui pourront être implantés sous la forme de variables globales de catégorie `registre`) :

`CP` (Code Pointer)
`SP` (Stack Pointer)
`MP` (Mark Pointer)
`DP` (Data Pointer)
`RCP` (Return CP).

- Le registre `CP` est l'adresse, dans le P–code, de l'instruction à exécuter (le déplacement à effectuer, en partant du début du fichier, pour atteindre le début de la P–instruction en question).

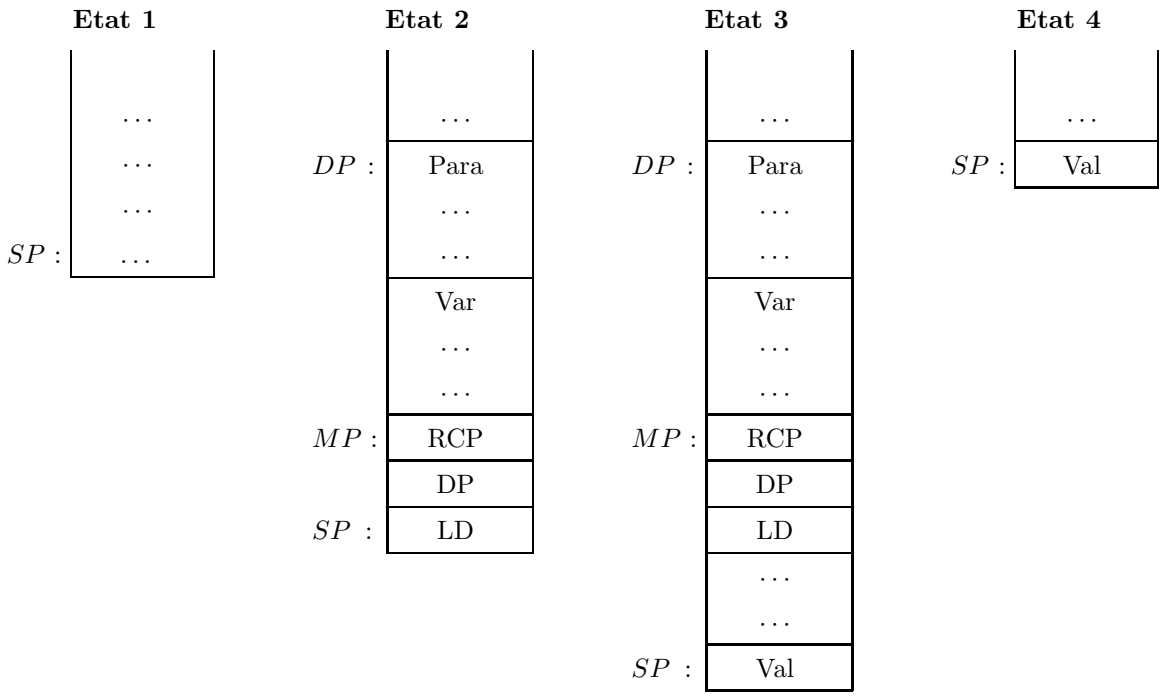
Les instructions d'un *P*–code s'exécutent séquentiellement, à partir de la première, sauf dans le cas des *P*–instructions qui peuvent déterminer un saut (`UJP`, `TJP`, `FJP`, `CALL`, `RETF`, `RETP`) et l'instruction de sortie pour cause d'erreur `ERREUR`; il faut donc incrémenter en conséquence le registre `CP` après l'exécution d'une *P*–instruction. Cette incrémentation peut être représentée schématiquement par `CP := CP + 1` mais il faut comprendre ce `1` comme étant la longueur de la P–instruction que l'on vient d'exécuter (il en est de même pour la variable globale `Indice` utilisée dans la traduction). Si l'on veut que l'incrémentation de `CP` ait toujours le même pas, il faut choisir de réserver un espace constant pour toute P–instruction (un nombre entier de mots machine).

- La valeur maximale de `CP` est désignée par *Codemax*

La pile d'exécution.

La pile d'exécution est désignée comme une partie initiale d'un tableau `PILE` : l'indice de son sommet est la valeur du registre `SP`.

• La valeur maximale de `SP` est désignée par $Maxstr$: cette valeur doit être choisie de façon judicieuse ! Toute P–instruction incrémentant `SP` doit commencer par un test de non débordement de pile (qui n'est pas mentionné dans la liste qui est donnée plus bas).



Evolution de la pile lors d'un appel (1 → 2) et lors d'un retour (3 → 4) de fonction.

- Le principe d'exécution d'une opération sur une pile est bien connu :
  - les arguments de l'opération sont empilés au sommet, dans l'ordre correspondant à la déclaration de cette opération,
  - l'exécution de l'opération a pour effet global de dépiler les arguments et d'empiler le résultat, si l'opération en produit un.
- Ce principe s'étend au cas des fonctions et des procédures.
  - APPEL : les arguments sont les arguments proprement dits (valeur actuelle des paramètres) qui doivent être évalués avant l'appel de la fonction et les variables locales. Ceci est figuré par le passage de l'état 1 à l'état 2 :
    - La P–instruction `PARA` détermine le début du bloc des arguments proprement dits,
    - la P–instruction `IEA` achève l'installation de l'enregistrement d'activation en empilant le bloc réservé aux variables locales et le bloc des sauvegardes.
  - RETOUR : l'enregistrement d'activation est dépilé et une valeur de retour est empilée, dans le cas d'une fonction. Ceci est figuré par le passage de l'état 3 à l'état 4.
    Les P–instructions `RETF` et `RETP` se chargent de plus de réactualiser les registres `SP`, `MP` et `CP`.

Les P–instructions.

Dans la liste des P–instructions qui suit, la mention d'un type de base qui vient souvent qualifier l'opérateur, (`E` pour entier, `B` pour booléen, `A` pour adresse, ...) ne sera pas utile ici puisque les objets qui sont manipulés sont toujours des entiers.

Le test de non débordement de pile, qu'il est indispensable d'effectuer avant l'exécution de toute P–instruction qui incrémente `SP`, n'est pas mentionné.

Expressions arithmétiques.

`ADD E` : $(\mathtt{E},\mathtt{E}) \rightarrow \mathtt{E}$.

```
PILE[SP - 1] := PILE[SP - 1] +E PILE[SP] ;
SP := SP - 1 ;
```

...

Expressions booléennes.

`AND` : $(\mathtt{B},\mathtt{B}) \rightarrow \mathtt{B}$.

```
PILE[SP - 1] := PILE[SP - 1] et PILE[SP] ;
SP := SP - 1 ;
```

...

`EQU T` : $(\mathtt{T},\mathtt{T}) \rightarrow \mathtt{B}$.

```
PILE[SP - 1] := PILE[SP - 1] =T PILE[SP] ;
SP := SP - 1 ;
```

...

Adressage direct des données.

`LOC p : vide` $\rightarrow$ `A`.

```
SP := SP + 1 ;
PILE[SP] := PILE[MP + 1] + p ;
```

`GLO p : vide` $\rightarrow$ `A`.

```
SP := SP + 1 ;
PILE[SP] := p ;
```

Adressage indexé des données.

`IXA` : $(\mathtt{E},\mathtt{A}) \rightarrow \mathtt{A}$.

```
PILE[SP - 1] := PILE[SP] + PILE[SP - 1] ;
SP := SP - 1 ;
```

Test de domaine.

`CHK p : E` $\rightarrow$ `E, Type(p) = E`.

```
si ( PILE[SP] < 0 ou PILE[SP] ≥ p ) alors
          erreur('hors du domaine')
fin ;
```

Lecture (Load) et écriture (Store) en mémoire.

LDO T p : vide $\rightarrow$ T, p $\in [0, Maxstr]$.

```
SP := SP + 1 ;
PILE[SP] := PILE[p] ;
```

LDC T p : vide $\rightarrow$ T, Type(p) = T.

```
SP := SP + 1 ;
PILE[SP] := p ;
```

SRO T p : T $\rightarrow$ vide, p $\in [0, Maxstr]$.

```
PILE[p] := PILE[SP] ;
SP := SP - 1 ;
```

STO T : (A, T) $\rightarrow$ vide.

```
PILE[PILE[SP - 1]] := PILE[SP] ;
SP := SP - 2 ;
```

Copie de blocs (tableaux).

MOV p : A $\rightarrow$ (T, . . . , T).

```
pour i décroissant de p - 1 à 0 faire
          PILE[SP + i] := PILE[PILE[SP] + i]
fin ;
```

Branchement.

UJP p : p $\in [0, Codemax]$.

```
CP := p ;
```

TJP p : B $\rightarrow$ vide, p $\in [0, Codemax]$.

```
si PILE[SP] alors CP := p fin ;
SP := SP - 1 ;
```

FJP p : B $\rightarrow$ vide, p $\in [0, Codemax]$.

```
si ( non PILE[SP] ) alors CP := p fin ;
SP := SP - 1 ;
```

Initialisation des registres.

INIT.

```
SP := -1 ;
MP := -1 ;
DP := 0 ;
RCP := -1 ;
```

Appel d'une procédure ou d'une fonction.

CALL p.

```
RCP := CP + 1 ;
CP := p ;
```

Installation d'un enregistrement d'activation.

```
PARA p.
          DP := SP - p + 1 ;

IEA p : vide → (U, ..., U, A, A, A).
          SP := SP + p + 3 ;
          PILE[SP - 2] := RCP ;
          PILE[SP - 1] := DP ;
          PILE[SP] := MP ;
          MP := SP - 2 ;
```

Retour de fonction ou de procédure.

```
RETF : 'EA' → T.
           CP := PILE[MP] ;
           PILE[PILE[MP + 1]] := PILE[SP] ;
           SP := PILE[MP + 1] ;
           MP := PILE[MP + 2] ;

RETP : 'EA' → vide.
           CP := PILE[MP] ;
           SP := PILE[MP + 1] - 1 ;
           MP := PILE[MP + 2] ;

ERREUR.
          erreur('sortie imprévue') ;
```

Entrées et sorties.

```
ECRIRE : E → vide.
          écrire l'entier PILE[SP] à l'écran ;
           SP := SP - 1 ;

LIRE : vide → E.
          lire un entier e à l'écran ;
           SP := SP + 1 ;
           PILE[SP] := e ;
```