Travaux Dirigés et Travaux Pratiques pour la Construction d'un Petit Compilateur de Pascal

Philippe MARQUET[1]

marquet@lifl.fr

31 août 1993
révisé le 17 juillet 1995
impression du 28 octobre 1999

1. Université des sciences et technologies de Lille. Laboratoire d'informatique fondamentale de Lille. Cité scientifique — Bâtiment M3. 59655 Villeneuve d'ascq cedex. France.

Table des matières

Table des figures

Liste des tableaux

Introduction

Le but de ce document est l'écriture d'un simple compilateur d'un langage « à la Pascal ». L'approche n'est pas de traiter des différentes méthodes de compilation, mais de détailler la construction d'un compilateur selon une méthode donnée : l'analyse récursive descendante. Le grand avantage de celle-ci est que le compilateur peut être construit directement à partir de la syntaxe du langage ; l'analyse sémantique et la génération de code sont « ajoutées » à cette analyse dirigée par la syntaxe.

Pourquoi écrire un compilateur ? Il est à peu près certain que la majorité des informaticiens n'auront jamais à écrire de compilateur. Cependant, mener à bien l'écriture d'un « gros » programme, le tester, documenter... a déjà un intérêt en soi. De plus, les algorithmes employés dans un compilateur sont repris dans la résolution d'autres problèmes : manipulation de listes de recherche (gestion de la table des symboles), vérification de la correction d'enregistrements (analyse syntaxique), traitement de caractères, de mots (grammaire BNF). Enfin, il semble intéressant pour un programmeur de connaître les dessous des choses ; comprendre comment les programmes se compilent et s'exécutent aide à la compréhension des limitations, obligations... imposées par les langages.

Chacun des chapitres introduit une nouvelle notion, propose des exercices, et amène, à travers une série de travaux pratiques, à la réalisation d'une partie du compilateur. L'ensemble des travaux pratiques doit aboutir à

- la réalisation effective d'un compilateur pour un sous-ensemble du langage Pascal ;
- le test du compilateur ;
- un manuel de maintenance du compilateur destiné à l'informaticien chargé de maintenir/étendre le produit ;
- un manuel d'utilisation détaillant la mise en œuvre du produit, la syntaxe et la sémantique du langage, etc.

Chapitre 1

Traducteurs et interpréteurs

1.1 Les traducteurs

La difficulté de programmer directement en *langage machine*, c'est-à-dire en introduisant des mots binaires à la main dans la mémoire de l'ordinateur, a fait naître plusieurs générations d'outils appelés *traducteurs*. Ces outils permettent de programmer en utilisant des structures, souvent conceptuellement complexes, formulées de façon relativement lisible.

Un traducteur est un *programme* qui accepte en entrée une représentation textuelle d'un algorithme décrit dans un *langage source* et qui produit en sortie une représentation du même algorithme dans un autre langage appelé *langage cible*, ou *langage objet*. De nos jours, les traducteurs les plus modernes produisent souvent du langage cible aussi « bon » que celui que produirait un spécialiste humain.

On distingue différents types de traducteurs :

Assembleur Le langage source des assembleurs utilise une représentation symbolique des instructions machines. Le langage cible est alors le langage machine. Un assembleur traduit généralement une instruction symbolique en une instruction machine. Les *macro-assembleurs* utilisent un mécanisme quelquefois sophistiqué de remplacement de texte pour permettre à l'utilisateur de créer ses propres instructions symboliques.

Compilateur Le langage source des compilateurs est dit de *haut niveau*. Une instruction du langage source est souvent traduite en un nombre important d'instructions du langage objet de *bas niveau*. Lorsque le langage produit n'est pas destiné à la machine sur laquelle tourne le compilateur, on parle de *cross-compilateur*.

Préprocesseur Le langage source et le langage objet des préprocesseurs sont des langages de haut niveau. Un préprocesseur fournit à l'utilisateur un sur-ensemble d'un langage donné en ajoutant certaines possibilités absentes du langage. Par exemple dans le langage C, il n'y a pas de déclarations de constantes, c'est le préprocesseur `cpp` qui les prend en charge.

Décompilateur et désassembleur Le langage source des décompilateurs et des désassembleurs est de plus bas niveau que le langage objet. En général, ces outils servent à regénérer un programme source à partir d'un programme objet. Les désassembleurs produisent une forme lisible d'un code en langage machine. Les décompilateurs de langages comme Pascal ne peuvent se concevoir que dans le cadre d'environnements de compilation appropriés (par exemple Mentor) dans lesquels les informations habituellement absentes d'un programme objet (par exemple les symboles utilisés dans le texte source) sont conservées.

1.2 T-diagramme et bootstrapping

Un traducteur est un programme lui-même écrit dans un langage informatique. Il est maintenant rare que ce langage soit de bas niveau, le plus souvent il s'agit aussi d'un langage de haut niveau. Ce langage est appelé *langage hôte* du traducteur. Une bonne notation pour un traducteur est le *T-diagramme* (figure 1.1).

Une traduction est alors notée par un T-diagramme (figure 1.2). Par exemple, la compilation C de myprog.c sur Sun 3 sera notée par le T-diagramme de la figure 1.3.

Le *bootstrapping* est une technique de construction de compilateurs portables au cours de laquelle on va écrire le compilateur dans son propre langage source.

Prenons l'exemple du premier compilateur Pascal développé par le créateur du langage : Nicklaus Wirth. Wirth a commencé par écrire un compilateur d'un sous-ensemble de Pascal, Pascal-1, en code CDC (le langage machine des Control Data), produisant du code CDC (figure 1.4).

Il écrivit ensuite un programme Pascal qui faisait la même chose que son premier compilateur. De plus, il inclua dans son programme des fonctionnalités absentes de Pascal-1. La compilation de ce programme par son premier compilateur donna un nouveau compilateur plus performant (figure 1.5).

Cette chaîne peut être notée par l'imbrication des T-diagrammes comme à la figure 1.6.

Lorsque Wirth voulut porter son compilateur sur machine ICL, il réécrivit un compilateur en Pascal générant du code ICL. Ce travail n'était pas trop difficile, puisque qu'il l'avait déjà fait pour CDC. En le compilant avec le compilateur Pascal du CDC (PAS), il fabriqua un cross-compilateur (Cross-PAS, figure 1.7).

Un dernière compilation produisit un compilateur fonctionnant sur ICL (PAS-ICL, figure 1.8).

1.3 Structure d'un compilateur

Un compilateur est un programme complexe divisé en *phases*. On peut imaginer que chaque phase communique avec la précédente et la suivante grâce à un langage approprié. En réalité les différentes phases sont le plus souvent entrelacées.

On peut mettre en évidence les phases suivantes :

Interface d'entrée Cette phase est responsable de la communication avec l'extérieur, il s'agit souvent du système d'exploitation, pour lire et écrire dans les fichiers d'entrée et de sortie. Cette phase est dépendante de la machine.

Analyseur lexical Son rôle est de reconnaître dans le flux de caractères fourni par l'interface d'entrée les mots élémentaires du langage tels les identificateurs, les mots clés, les opérateurs, etc. Les mots reconnus sont codés sous forme de *tokens* qui seront fournis à la phase suivante.

Analyseur syntaxique Son rôle est de reconnaître dans le flux de tokens fourni par l'analyseur lexical des *phrases* vérifiant la syntaxe du langage. La syntaxe du langage, précisée par une *grammaire algébrique*, permet à l'analyseur syntaxique de construire un *arbre de dérivation* ou *arbre de syntaxe abstraite* qui sera utilisé par les phases suivantes.

```
+------------------------------+
| langage   traducteur  langage|
| source                cible  |
+---------+          +---------+
          |  langage |
          |  hote    |
          +----------+
```

FIG. 1.1 – *T-diagramme pour un traducteur.*

```
          +------------------------------+
programme | langage   traducteur  langage|  programme
source    | source                cible  |  objet
          +---------+          +---------+
                    |  langage |
                    |  hote    |
                    +----------+
                       machine
                       hote
```

FIG. 1.2 – *T-diagramme pour une traduction.*

```
         +------------------------------+
myprog.c |    C    compilateur C  code  |  a.out
         |             (cc)       68000 |
         +---------+          +---------+
                   |   code   |
                   |   68000  |
                   +----------+
                       Sun 3
```

FIG. 1.3 – *T-diagramme pour* `cc myproc.c`.

```
+------------------------------+
| Pascal-1     PAS-1    code   |
|                       CDC    |
+---------+          +---------+
          |    code  |
          |    CDC   |
          +----------+
```

FIG. 1.4 – *Compilateur PAS-1.*


```
+------------------------------+
| Pascal-2              code   |
|                       CDC    |
+---------+          +---------+--------------------+
          | Pascal-1 | Pascal-1     PAS-1    code   |
          |          |                       CDC    |
          +----------+---------+          +---------+
                               |    code  |
                               |    CDC   |
                               +----------+
```

FIG. 1.5 – *Compilation produisant PAS-2.*

```
+------------------------------+          +------------------------------+
| Pascal-2              code   |          | Pascal-2    PAS-2     code   |
|                       CDC    |          |                       CDC    |
+---------+          +---------+----------+---------+          +---------+
          | Pascal-1 | Pascal-1     PAS-1    code   |   code    |
          |          |                       CDC    |   CDC     |
          +----------+---------+          +---------+----------+
                               |    code  |
                               |    CDC   |
                               +----------+
```

FIG. 1.6 – *Chaîne de compilation produisant PAS-2.*

```
+------------------------------+          +------------------------------+
| Pascal                code   |          | Pascal   Cross-PAS    code   |
|                       ICL    |          |                       ICL    |
+---------+          +---------+----------+---------+          +---------+
          |  Pascal  | Pascal       PAS      code   |   code    |
          |          |                       CDC    |   CDC     |
          +----------+---------+          +---------+----------+
                               |    code  |
                               |    CDC   |
                               +----------+
```

FIG. 1.7 – *Production d'un cross-compilateur pour ICL.*

```
+------------------------------+          +------------------------------+
| Pascal                code   |          | Pascal    PAS-ICL     code   |
|                       ICL    |          |                       ICL    |
+---------+          +---------+----------+---------+          +---------+
          |  Pascal  | Pascal    Cross-PAS   code   |   code    |
          |          |                       ICL    |   ICL     |
          +----------+---------+          +---------+----------+
                               |    code  |
                               |    CDC   |
                               +----------+
```

FIG. 1.8 – *Production d'un compilateur pour ICL.*

Analyseur sémantique Son rôle est de déterminer si les phrases reconnues par l'analyseur syntaxiques ont un sens. En pratique son rôle est essentiellement d'assurer que le texte source vérifie les règles non syntaxiques qui pourraient être imposées (concordance de type, visibilité des variables, présence des déclarations nécessaires...).

Générateur de code intermédiaire On va souvent, pour des raisons de portabilité, traduire le texte source dans un langage de plus bas niveau dit intermédiaire car ne correspondant pas au but final de la traduction, à savoir produire du code exécutable sur une machine. Ce code intermédiaire a la propriété d'être général et de ne pas supposer une architecture de machine précise (nombre de registres, disposition de la mémoire...). Pour passer d'une machine à une autre, il suffit en général de réécrire les phases ultérieures à la génération de code intermédiaire.

Optimisation du code intermédiaire Cette phase optionnelle réalise une amélioration de la qualité du code intermédiaire. Son rôle est d'éliminer le code inutile ou de simplifier ce code.

Génération de code et optimisation finale Cette phase est bien sûr dépendante de la machine cible. Le code intermédiaire va être traduit en instructions machines, les emplacements mémoires vont être choisis, les registres vont être affectés à des tâches particulières... Cette phase étant très importante pour la future efficacité du compilateur, une dernière optimisation peut être entreprise.

De plus, au cours de toutes ces phases, il est nécessaire de maintenir une *table des symboles* qui garde trace des symboles utilisés dans le texte source et les attributs qui leur sont associés (type, adresse, valeur...).

Une autre tâche que doit réaliser un compilateur est la *gestion des erreurs* avec des techniques qui le plus souvent permettent au compilateur de continuer « sainement » le travail d'analyse après la détection d'erreurs, et qui quelquefois permettent la correction d'erreurs simples.

La structure d'un compilateur est schématisée par la figure 1.9.

Exemple

Prenons en exemple le texte source suivant :

```
while (1<P) and (P<3) do
   P:=P+Q
```

L'interface d'entrée va fournir une suite de caractères significatifs (l'espace blanc est noté par le caractère _) :

```
w h i l e _ ( 1 < P ) _ a n d _ ( P < 3 ) _ d o _ P : = P + Q _
```



```
                       +-------------+
                       | Code source |
                       +------|------+
                              V
                       Interface d'entree    <-|
                              |                |
                              V                |
                   |->  Analyseur lexical    <-|
                   |          |                |
                   |          V                |
                   |-> Analyseur syntaxique  <-|
                   |          |                |
                   |          V                |
                   |-> Analyseur semantique  <-|
                   |          |                |
                   |          V                |
Table des symboles |->  Generation de code   <-|  Gestion des erreurs
                   |      intermediaire        |
                   |          |                |
                   |          V                |
                   |-> Optimisation de code  <-|
                         intermediaire         |
                              |                |
                              V                |
                       Generation de code    <-|
                              |
                              V
                      Optimisation de code
                              |
                        +-----V------+
                        | Code objet |
                        +------------+
```

FIG. 1.9 – *Organisation générale d'un compilateur.*

L'analyseur syntaxique va reconnaître les mots (tokens) suivants :

```
mot clef                while
symbole                 parenthèse ouvrante
constante numérique     1
opérateur relationnel   <
identificateur          P
symbole                 parenthèse fermante
mot clef                and
symbole                 parenthèse ouvrante
identificateur          P
opérateur relationnel   <
constante numérique     3
symbole                 parenthèse fermante
mot clef                do
identificateur          P
symbole                 affectation
identificateur          P
opérateur               +
identificateur          Q
```

L'analyseur syntaxique va construire un arbre tel celui de la figure 1.10(page 21). Le code intermédiaire généré sera de la forme :

```
L0    if 1<P goto L1
      goto L3
L1    if P<3 goto L2
      goto L3
L2    P:=P+Q
      goto L0
L3    ...
```

Il pourra être optimisé en :

```
L0    if 1<P goto L1
      goto L3
      if P>=3 goto L1
      P:=P+Q
      goto L0
L1    ...
```

1.4 Interpréteur

Certains compilateurs ne possèdent pas de phase de génération de code en langage machine. Ils s'arrêtent à la production de code intermédiaire. L'exécution du programme source consiste alors en une *interprétation* du code intermédiaire par un programme appelé *interprète* ou *interpréteur*. Quelquefois, l'interprète exécute directement le langage source, comme en Basic ou en APL. Le plus souvent, il exécute un code intermédiaire fabriqué par un compilateur, par exemple le P-Code fabriqué par le compilateur Pascal UCSD, ou le A-Code du compilateur ADA-Artek.

L'intérêt d'une interprétation du langage intermédiaire est la portabilité du compilateur ainsi créé : le groupe de Wirth à Zurich diffusait le langage Pascal en fournissant le source d'un compilateur Pascal en P-Code et le source Pascal d'un interprète P-Code. Pour porter Pascal sur sa machine, il suffisait de réécrire un programme équivalent à l'interprète fourni.

Le désavantage d'une interprétation est une baisse de performance des programmes ; l'interprétation étant souvent plus lente que ne le serait l'exécution de code machine optimisé d'un compilateur complet.

1.5 Exemple de code intermédiaire pour une machine à pile

Nous présentons ici un langage intermédiaire qui est une forme simplifiée du P-Code, et étudions la réalisation d'un interprète pour ce langage à la section 1.6.

Le P-Code est le langage intermédiaire utilisé pour le Pascal UCSD. Il est associé à la machine abstraite P-Machine. Cette machine n'a qu'une zone mémoire gérée en pile (en particulier, elle n'a donc pas de registres, mis à part le pointeur de sommet de pile (SP) et le compteur d'instructions (PC)). La plupart des instructions du P-Code prennent donc leurs opérandes sur la pile et laissent leur résultat sur cette pile.

La structuration de la mémoire en pile facilite la compilation de langages de haut niveau autorisant la récursion et la compilation d'expressions arithmétiques complexes. Cette structuration n'a pas que des avantages, en particulier pour l'utilisation optimale de registres de la machine physique.

Le bas de la pile est réservé pour l'allocation des variables globales, c'est-à-dire que la première instruction d'un programme P-Code réserve une partie de la pile correspondant à la zone d'allocation des variables utilisées dans le programme ; cette réservation est faite par incrémentation du pointeur de pile de la taille mémoire nécessaire.

Les éléments de la pile sont de simples entiers ; les adresses sont des adresses absolues dans la pile.

Le jeu d'instruction du P-Code simplifié que nous considérerons est donné à la figure 1.11.

L'exemple suivant :

```
begin
   repeat
      read (A) ;
      B := A + B ;
   until A = 0 ;
   write (B) ;
end
```

produira le P-code de la figure 1.12.

1.6 Écriture d'un interprète de P-Code

Les structures de données nécessaires lors de l'écriture d'un interprète simplifié pour le P-Code sont 1–un tableau `MEM` représentant la pile de la machine et un pointeur de pile associé (`SP`, stack pointer) :

```
var MEM : array [0 .. TAILLEMEM] of integer ;
    SP : integer ;
```

```
                        while_inst
                v-----------|----------------v
          cond_compose                     inst_simple
      v---------|--------v                      |
     cond      and     cond                affectation
 v-----|------v    v-----|-----v            v---|---v
num  op_inf  id   id  op_inf  num          id      exp
                                               v---|------v
                                              id  op_add  id
```

FIG. 1.10 – *Exemple d'arbre de dérivation.*

`ADD`	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour `SUB`, `MUL`, `DIV`)
`EQL`	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour `NEQ`, `GTR`, `LSS`, `GEQ`, `LEQ`)
`PRN`	imprime le sommet, dépile
`INN`	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
`INT c`	incrémente de la constante `c` le pointeur de pile (la constante `c` peut être négative)
`LDI v`	empile la valeur `v`
`LDA a`	empile l'adresse `a`
`LDV`	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
`STO`	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
`BRN i`	branchement inconditionnel à l'instruction `i`
`BZE i`	branchement à l'instruction `i` si le sommet = 0, dépile
`HLT`	halte

FIG. 1.11 – *Jeu d'instructions du P-Code.*

2–un tableau `PCODE` stockant les pseudo-instructions, auquel sont associés l'instruction courante `INST` et un pointeur d'instruction `PC` ; une instruction est un couple (mnémonique, opérande éventuelle) :

```
type MNEMONIQUES = (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,LSS,GEQ,LEQ,
        PRN,INN,INT,LDI,LDA,LDV,STO,BRN,BZE,HLT);
     INSTRUCTION = record
        MNE : MNEMONIQUES;
        SUITE : integer
     end
var  PCODE : array [0 .. TAILLECODE] of INSTRUCTION;
     PC : integer;
     INST : INSTRUCTION;
```

L'algorithme se divise en deux parties ; le « chargeur » et l'interpréteur proprement-dit.

Le chargeur remplit la table `PCODE` à partir d'un fichier de code P-Code. L'interprète est basé sur le modèle de la figure 1.13

1.7 Travaux dirigés et travaux pratiques

1. Écrire un programme P-Code dont l'algorithme est défini par

```
begin
   read (A) ; SMALLEST:=A ; LARGEST:=A ;
   while A <> 0 do begin
      if A > LARGEST then LARGEST:=A ;
      if A < SMALLEST then SMALLEST:=A ;
      read (A)
   end
   write (SMALLEST,LARGEST)
end.
```

2. Le jeu d'instructions du P-Code vous semble-t-il suffisant pour définir une « machine à entiers » ?
3. Écrire le « chargeur » de P-Code ; il est préalablement nécessaire de définir la syntaxe du source P-Code (label d'instruction, commentaires...).
4. Compléter le programme de la figure 1.13 pour définir un interprète P-Code.
5. Réaliser une série de tests de l'interpréteur ainsi défini.
6. Un programme P-Code peut-il être incorrect ? Deux types d'incorrections sont distinguées : celles qui peuvent être détectées *statiquement*, et celles qui ne peuvent être détectées qu'à l'exécution. Proposer un outil d'analyse statique d'un programme P-Code qui en vérifie la correction. Ajouter dans l'interpréteur la détection des erreurs survenant lors de l'exécution.
7. Quels outils peuvent être utiles pour la mise au point de programmes P-Code ? En expliquer la mise en place.
8. Développer une variante du P-code avec des instructions de *branchements relatifs* plutôt qu'*absolus*. Quels sont les avantages de cette approche ?
9. Quelles optimisations peuvent être mises en place au niveau du P-Code. Expliciter la structure d'un optimiseur. Comment l'intégrer dans l'interpréteur ?

```
0   INT 2   réserve 2 emplacements pour A (0) et B (1)
1   LDA 0   empile l'adresse de A
2   INN     lit une valeur la range dans A
3   LDA 1   empile l'adresse de B
4   LDA 0   empile l'adresse de A
5   LDV     déréférence. Valeur de A au sommet
6   LDA 1   empile l'adresse de B
7   LDV     déréférence. Valeur de B au sommet
8   ADD     addition de A et B
9   STO     stocke le résultat dans B
10  LDA 0   empile l'adresse de A
11  LDV     déréférence. Valeur de A au sommet
12  LDI 0   empile la valeur 0
13  EQL     test d'égalité
14  BZE 1   branchement en 1 si A <>0
15  LDA 1   empile l'adresse de B
16  LDV     déréférence. Valeur de B au sommet
17  PRN     écrit le résultat
18  HLT     fin
```

Fig. 1.12 – *Exemple de programme P-Code.*

```
begin
   (* initialise pointeur d'instruction PC et pointeur de pile SP *)
   (* initialise program status PS a EXECUTION *)
   repeat
      (* chargement *)
      INST = PCODE [PC]
      incremente PC
      (* execution *)
      with INST do
         case MNE of
            ADD : begin SP:=SP-1; MEM[SP]:=MEM[SP]+MEM[SP+1] end ;
            ...
            HLT : PS:=FINI ;
         end
   until PS <> EXECUTION
end.
```

Fig. 1.13 – *Modèle d'un interprète P-Code.*

Première partie

Construction d'un compilateur simple

Chapitre 2

Analyse syntaxique

On définit un petit langage à la Pascal, PP. La première version s'appelle PP1. Le but est la construction d'un compilateur qui génère du P-Code (cf. section 1.5) à partir d'un programme PP1 ; ce compilateur s'appellera `pp1` et sera développé en C (figure 2.1). Ce chapitre se limite à l'analyse lexicale et syntaxique du langage ; la construction du compilateur sera complétée dans les chapitres suivants.

2.1 Syntaxe d'un langage

La syntaxe d'un langage peut être donnée par des exemples ou une description informelle. Cependant, il est plus aisé de décrire une syntaxe par une *grammaire algébrique*.

Une telle grammaire est formée de *règles de production*. Chaque règle indique en partie gauche un *non-terminal* qui peut être dérivé en l'une des formes apparaissant en partie droite (les formes sont séparées par des |).

Une forme est composée de *terminaux* (en minuscule, par exemple `while`, ou caractères spéciaux, par exemple `+` ; ces terminaux apparaissent tel quel dans le langage) et/ou de *non-terminaux* (en majuscule, par exemple `BLOCK`).

Une forme entre accolades `{` et `}` signifie qu'elle peut être présente zéro, une ou plusieurs fois. Le symbole ε signifie que la partie gauche peut se dériver en une forme vide.

Le but de l'analyse syntaxique est de montrer que l'axiome de la grammaire (un non-terminal particulier, en général la partie gauche de la première règle, ici `PROGRAM`) peut être dérivé en une suite de terminaux formant le texte à analyser.

2.2 Syntaxe du langage PP1

La syntaxe du langage PP1 est donnée à la figure 2.2.

Les mots clés sont réservés ; ils apparaissent en minuscule dans la grammaire.

Certains non-terminaux ne sont pas décrits par cette grammaire. Il s'agit des non-terminaux pris en charge par l'analyse lexicale :

`ID` représente les identificateurs, c'est-à-dire toute suite de lettres ou chiffres commençant par une lettre et qui ne représente pas un mot clé (qui sont les terminaux présents dans la grammaire) ;

`NUM` représente les constantes numériques, c'est-à-dire toute suite de chiffres.


```
+---------------------------------+
|   PP1         pp1      P-Code   |
|                                 |
+----------+           +----------+
           |     C     |
           |           |
           +-----------+
```

FIG. 2.1 – *T-diagramme du compilateur* `pp1`.

```
PROGRAM   ::=  program ID ; BLOCK .
BLOCK     ::=  CONSTS VARS INSTS
CONSTS    ::=  const ID = NUM ; { ID = NUM ; } | ε
VARS      ::=  var ID { , ID } ; | ε
INSTS     ::=  begin INST { ; INST } end
INST      ::=  INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | ε
AFFEC     ::=  ID := EXPR
SI        ::=  if COND then INST
TANTQUE   ::=  while COND do INST
ECRIRE    ::=  write ( EXPR { , EXPR } )
LIRE      ::=  read ( ID { , ID } )
COND      ::=  EXPR RELOP EXPR
RELOP     ::=  = | <>| <| >| <= | >=
EXPR      ::=  TERM { ADDOP TERM }
ADDOP     ::=  + | -
TERM      ::=  FACT { MULOP FACT }
MULOP     ::=  * | /
FACT      ::=  ID | NUM | ( EXPR )
```

FIG. 2.2 – *Grammaire du langage PP1.*

2.2.1 Méta-règles

Une série de règles définissent la forme d'un programme PP1 :

1. Un *commentaire* est une suite de caractères encadrés des parenthèses (* et *) ;
2. Un *séparateur* est un *caractère séparateur* (espace blanc, tabulation, retour chariot) ou un *commentaire* ;
3. Deux ID ou *mots clés* qui se suivent doivent être séparés par au moins un *séparateur* ;
4. Des *séparateurs* peuvent être insérés partout, sauf à l'intérieur de *terminaux*.

2.2.2 Sémantique

La sémantique de PP1 est claquée sur celle de Pascal. Les identificateurs doivent être déclarés pour être utilisés. Toutes les variables sont implicitement déclarées de type entier.

2.3 Analyseur syntaxique de PP1

L'analyseur lexical peut être vu comme une procédure appelée par l'analyseur syntaxique. Chaque appel à la procédure d'analyse lexical NEXT_TOKEN met à jour les variables TOKEN, SYM, et VAL :

TOKEN contient le dernier token lu ;

SYM contient la forme textuelle du dernier token lu ;

VAL est la valeur du dernier token lu.

```
type TOKENS = (ID_TOKEN, NUM_TOKEN, PLUS_TOKEN, MOINS_TOKEN, MUL_TOKEN
         DIV_TOKEN, EGAL_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN,
         INF_EGAL_TOKEN, SUP_EGAL_TOKEN, PAR_OUV_TOKEN,
         PAR_FER_TOKEN, VIRG_TOKEN, PT_VIRG_TOKEN, POINT_TOKEN,
         AFFEC_TOKEN, BEGIN_TOKEN, END_TOKEN, IF_TOKEN ,WHILE_TOKEN,
         THEN_TOKEN, DO_TOKEN, WRITE_TOKEN, READ_TOKEN,
         CONST_TOKEN, VAR_TOKEN, PROGRAM_TOKEN, TOKEN_INCONNU) ;
     ALFA = packed array [1 .. 8] of char ;
var  TOKEN : TOKENS ;
     SYM : ALFA ;
     VAL : integer ;
```

Une procédure TESTE teste si le prochain token (TOKEN) est bien celui passé en paramètre à la procédure ; on s'arrête sur une erreur sinon (procédure ERREUR) :

```
procedure TESTE (T:TOKENS) ;
begin
   if TOKEN = T
      then NEXT_TOKEN
      else ERREUR
end ;
```

Le langage PP1 ayant de bonnes propriétés (se référer au cours d'Analyse Syntaxique), il est possible d'écrire un analyseur récursif descendant simple (sans retour arrière). L'idée est que chaque règle de la grammaire est associée à une procédure qui « vérifie » la concordance du

texte à analyser avec une de ses parties droites. Voici la fonction principale d'un tel analyseur :

```
procedure PROGRAM;
begin
   TESTE(PROGRAM_TOKEN) ;
   TESTE (ID_TOKEN) ;
   TESTE (PT_VIRG_TOKEN) ;
   BLOCK ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

La procédure BLOCK est la suivante :

```
procedure BLOCK ;
begin
   if TOKEN = CONST_TOKEN then CONSTS ;
   if TOKEN = VAR_TOKEN then VARS ;
   INSTS
end
```

Les autres procédures sont :

```
procedure CONSTS ;
begin
   TESTE (CONST_TOKEN) ;
   repeat
      TESTE (ID_TOKEN) ;
      TESTE (EGAL_TOKEN) ;
      TESTE (NUM_TOKEN) ;
      TESTE (PT_VIRG_TOKEN)
   until TOKEN <> ID_TOKEN
end ;

procedure VARS ;
begin
   TESTE (VAR_TOKEN) ;
   TESTE (ID_TOKEN) ;
   while TOKEN = VIRG_TOKEN do
      begin NEXT_TOKEN ; TESTE (ID_TOKEN) end ;
   TESTE (PT_VIRG_TOKEN)
end ;


procedure INSTS ;
begin
   TESTE (BEGIN_TOKEN) ;
   INST ;
   while TOKEN = PT_VIRG_TOKEN do
      begin NEXT_TOKEN ; INST end ;
   TESTE (END_TOKEN)
end ;
```

```
procedure INST ;
begin
   case TOKEN of
      ID_TOKEN : AFFEC ;
      IF_TOKEN : SI ;
      WHILE_TOKEN : TANTQUE ;
      BEGIN_TOKEN : INSTS ;
      WRITE_TOKEN : ECRIRE ;
      READ_TOKEN : LIRE
   end
end ;


procedure AFFEC ;
begin
   TESTE (ID_TOKEN) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR
end ;


procedure SI ;
begin
   TESTE (IF_TOKEN) ;
   COND ;
   TESTE (THEN_TOKEN) ;
   INST
end ;


procedure TANTQUE ;
begin
   TESTE (WHILE_TOKEN) ;
   COND ;
   TESTE (DO_TOKEN) ;
   INST
end ;


procedure ECRIRE ;
begin
   TESTE (WRITE_TOKEN) ;
   TESTE (PAR_OUV_TOKEN) ;
   EXPR ;
   while TOKEN = VIRG_TOKEN do
      begin NEXT_TOKEN ; EXPR end ;
   TESTE (PAR_FER_TOKEN) ;
end ;
```



```
procedure LIRE ;
begin
   TESTE (READ_TOKEN) ;
   TESTE (PAR_OUV_TOKEN) ;
   TESTE (ID_TOKEN) ;
   while TOKEN = VIRG_TOKEN do
      begin NEXT_TOKEN ; TESTE (ID_TOKEN) end ;
   TESTE (PAR_FER_TOKEN)
end ;

procedure EXPR ;
begin
   TERM ;
   while TOKEN in [PLUS_TOKEN, MOINS_TOKEN] do
      begin NEXT_TOKEN ; TERM end
end ;

procedure COND ;
begin
   EXPR ;
   if TOKEN in [EGAL_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN,
      INF_EGAL_TOKEN, SUP_EGAL_TOKEN ]
      then begin
         NEXT_TOKEN ;
         EXPR
      end
end ;

procedure TERM ;
begin
   FACT ;
   while TOKEN in [MULT_TOKEN, DIV_TOKEN] do
      begin NEXT_TOKEN ; FACT end
end ;

procedure FACT ;
begin
   if TOKEN in [ID_TOKEN, NUM_TOKEN]
      then NEXT_TOKEN
   else begin
      TESTE (PAR_OUV_TOKEN) ;
      EXPR ;
      TESTE (PAR_FER_TOKEN)
   end
end ;
```

Cet analyseur s'arrête à la première erreur détectée. Nous verrons dans un chapitre suivant comment améliorer cette situation.

2.4 Travaux dirigés et travaux pratiques

1. Quelle est l'utilité des points virgules dans la grammaire du langage PP1 (figure 2.2)? Peux-t-on les supprimer?
2. Dans la grammaire de PP1 (comme en Pascal), une instruction `INST` peut se dériver en ε. Quelle peut être l'utilité d'une instruction vide?
3. En PP1, les déclarations de constantes (`const`) doivent précéder les déclarations de variables (`var`). Quelle est l'utilité d'une telle règle? Comment s'en affranchir?
4. Modifier la grammaire de PP1 (figure 2.2) pour inclure une branche `else` facultative à l'alternative `if`. Quel problème survient alors? Considérer le code :

```
begin
   if A then
   ...
   if B then
   ...
   else
   ...
end
```

   Comment le résoudre?
5. Montrer que la méta-règle définissant les commentaires à la page 29 est ambiguë. Proposer une modification pour lever l'ambiguïté.

   La possibilité d'imbrication de commentaires est-elle souhaitable? Considérer le code :

```
begin
   (* Somme de A et B
   A := A + B ;
   (* Produit de A et B *)
   A := A * B
end
```

6. Écrire une fonction `NEXT_CHAR` qui retourne le prochain caractère du flux d'entrée en effectuant le filtrage suivant : tout *séparateur* sera renvoyé comme le caractère *espace*. (Tenir compte de la convention adoptée dans l'exercice précédent pour les commentaires).
7. Écrire la fonction `NEXT_TOKEN` d'analyse lexicale en utilisant la fonction `NEXT_CHAR`.
8. Écrire les fonctions d'analyse syntaxique du langage PP1 défini par la grammaire de la figure 2.2 sur le modèle donné dans ce chapitre.
9. à l'aide des fonctions précédentes, écrire un programme qui vérifie la correction syntaxique d'un code PP1. En cas d'erreur, on affiche le numéro de la ligne à laquelle la première erreur apparaît.
10. Comment imaginer la prise en compte de la possibilité d'inclure des blancs au milieu des mots clés et des identificateurs? (Cette possibilité est, par exemple, offerte par le langage Fortran.)

Chapitre 3

Analyse sémantique

Une fois le texte source reconnu syntaxiquement correct, il y a lieu de voir si le texte vérifie des règles non-exprimées par la syntaxe. C'est le rôle de l'analyse sémantique.

3.1 Table des symboles

Pour le langage PP1, cela consiste essentiellement à vérifier que les identificateurs utilisés sont bien déclarés. Pour cela, il est nécessaire de mémoriser les identificateurs déclarés pour tester la déclaration ou non des identificateurs utilisés. Cette mémorisation se fait dans une *table des symboles*.

Dans cette table, on associe à chaque entrée (chaque identificateur) toutes les informations connues sur lui, pour le moment :

- sa forme textuelle (son nom) ;
- sa classe, à savoir s'il désigne un programme, une constante, ou une variable.

```
type
   TABLEINDEX = 1 .. INDEXMAX ;
   CLASSES = (PROGRAMME, CONSTANTE, VARIABLE) ;
   CLASSET = set of CLASSES ;
var
   TABLESYM : array [TABLEINDEX] of record
      NOM : ALFA ;
      CLASSE : CLASSES
   end ;
   DERNIERSYM : TABLEINDEX ; (* indice du dernier symbole entre *)
```

La table des symboles est manipulée par deux fonctions :

`ENTRERSYM` ajoute le symbole `SYM` (cf. déclaration page 29) dans la table des symboles avec la classe passée en paramètre ; une version de base peut être la suivante :

```
procedure ENTRERSYM (C:CLASSES) ;
begin
   if DERNIERSYM = INDEXMAX then ERREUR ;
   DERNIERSYM := DERNIERSYM + 1 ;
   with TABLESYM [DERNIERSYM] do
```



```
      begin NOM := SYM ; CLASSE := C end
end ;
```

`CHERCHERSYM` recherche l'identificateur `SYM` dans la table des symboles parmi les classes permises passées en paramètre et retourne l'index de l'entrée correspondante dans la table des symboles, ou s'arrête sur `ERREUR` ; on donne ici une version de base :

```
procedure CHERCHERSYM (var INDEX:TABLEINDEX ; PERMIS:CLASSET) ;
var FIN : booleen ;
begin
   INDEX := DERNIERSYM ; FIN := false ;
   while not FIN do begin
      if INDEX = 0 then FIN := true
      else if TABLESYM [INDEX].NOM = SYM
         then FIN := true
         else INDEX := INDEX - 1
   end ;
   if INDEX = 0 then ERREUR (* SYM absent *)
   else if not (TABLESYM[INDEX].CLASSE in PERMIS)
      then ERREUR ;  (* SYM pas de bonne classe *)
end ;
```

3.2 Entrée dans la table des symboles

La procédure `ENTRERSYM` est utilisée lors de la déclaration de symboles par l'intermédiaire d'une procédure `TESTE_ET_ENTRE` qui vérifie que le prochain token est celui attendu et qui met à jour la table des symboles :

```
procedure TESTE_ET_ENTRE (T:TOKENS ; C:CLASSES) ;
begin
   if TOKEN = T then
      begin
         ENTRERSYM (C) ;
         NEXT_TOKEN
      end
   else ERREUR
end ;
```

Cette fonction `TESTE_ET_ENTRE` est appelée lors de la déclaration de symboles ; c'est-à-dire dans les fonctions `CONSTS`, `VARS` et `PROGRAM`, par exemple :

```
procedure PROGRAM ;
begin
   TESTE (PROGRAM_TOKEN) ;
   TESTE_ET_ENTRE (ID_TOKEN, PROGRAMME) ;
   TEST (PT_VIRG_TOKEN) ;
   BLOCK ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

3.3 Consultation de la table des symboles

De manière symétrique, une procédure TESTE_ET_CHERCHE est une évolution de la procédure TESTE qui de plus recherche le symbole SYM dans la table des symboles par un appel à CHERCHESYM, l'index du symbole recherché dans la table des symboles est retourné par la variable PLACESYM :

```
var PLACESYM : TABLEINDEX ;

procedure TESTE_ET_CHERCHE (T:TOKENS; PERMIS:CLASSET) ;
begin
   if TOKEN = T then begin
      CHERCHERSYM (PLACESYM, PERMIS) ;
      NEXT_TOKEN ;
   else ERREUR ;
end ;
```

Cette fonction est appelée lors de l'utilisation d'identificateur, c'est-à-dire par FACT, AFFEC, et LIRE. On a par exemple :

```
procedure AFFEC ;
begin
   TESTE_ET_CHERCHE (ID_TOKEN, VARIABLE) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR ;
end ;

procedure FACT ;
begin
   if TOKEN = ID_TOKEN
      then TESTE_ET_CHERCHE (ID_TOKEN, [CONSTANTE, VARIABLE])
   else if TOKEN = NUM_TOKEN
      then NEXT_TOKEN
   else begin
      TESTE (PAR_OUV_TOKEN) ;
      EXPR ;
      TESTE (PAR_FER_TOKEN) ;
   end
end ;
```

3.4 Travaux dirigés et travaux pratiques

1. Les versions proposées de ENTRERSYM et CHERCHERSYM ne tiennent pas compte des doubles déclarations. Comment prendre ces déclarations fautives en compte ?
   Serait-il imaginable d'accepter des déclarations telles que :
   ```
   var A, B, A ;
   ```
   et telles que
   ```
   const A = 3 ;
   var A ;
   ```


2. La recherche de symboles est telle que deux symboles de classes différentes ne peuvent avoir le même nom. Cette règle est-elle nécessaire ?
3. Écrire l'analyseur sémantique de PP1. On testera que les identificateurs ne sont pas déclarés plusieurs fois.
4. Il est intéressant de proposer à l'utilisateur la production d'une table des symboles. Cette option peut être activée par le passage d'un paramètre au compilateur, par exemple
   ```
   pp1 -tablesymb
   ```
   ou par l'utilisation d'une directive de compilation incluse dans un commentaire :
   ```
   (*$T+ *)
   ```
   pour l'activation de la production de la table des symboles, et
   ```
   (*$T- *)
   ```
   pour la désactivation. Quelles sont les avantages de l'une et l'autre des méthodes. Sont-elles incompatibles ? Proposer une implantation.
5. Il est possible d'imaginer que les identificateurs non déclarés doivent être traités comme des variables (déclaration implicite ; c'est le cas en Fortran) et mis dans la table des symboles au moment de leur première référence. Est-ce une bonne idée ? Comment l'implémenter ?
6. La table des symboles proposée en exemple est une simple table. Il est possible d'implanter cette table par des arbres binaires de recherches ou d'aider la recherche par une table de hashage. Est-ce une bonne idée ? Quel problème pourrait survenir ?

Chapitre 4

Gestion des erreurs

Jusqu'à maintenant, le compilateur pp1 s'arrête sur la première erreur rencontrée. Il est bien évidemment intéressant de pouvoir poursuivre la compilation aussi loin que faire se peut. On peut imaginer que la procédure ERREUR n'arrête pas l'exécution du compilateur, mais se limite à reporter une erreur. Il est cependant peu vraisemblable que la suite de la compilation puisse se dérouler sans engendrer la détection d'erreurs curieuses et étranges. La gestion des erreurs doit être plus fine et autoriser la reprise après erreur dans de « bonnes » conditions.

Les programmes PP1 (comme la plupart des programmes) peuvent contenir des erreurs à différents niveaux :

- erreurs lexicales, comme l'écriture erronée d'un identificateur, d'un mot clé ou d'un opérateur ;
- erreurs syntaxiques, comme une expression arithmétique mal parenthésée ;
- erreurs sémantiques, comme un opérateur appliqué à un opérande non compatible ;
- erreurs logiques, comme une boucle sans fin.

Souvent, dans un compilateur, la part la plus importante dans la détection et la récupération sur erreur est centrée autour de l'analyse syntaxique.

On distingue :

la récupération après erreur dont le but est, lors de la détection d'une erreur, de positionner le compilateur dans un état lui permettant de continuer sainement ; et

la correction d'erreur dont le but est de corriger des erreurs et de continuer la compilation malgré la présence d'erreurs dans le programme.

4.1 Messages d'erreurs

Il semble important que des messages d'erreurs informatifs soient fournis par le compilateur à la rencontre d'une erreur. Pour ce faire, on définit une liste d'erreurs, se référer par exemple à la table 4.1.

D'autres erreurs peuvent survenir, nous les qualifierons d'erreurs d'administration ; ce sont par exemple l'impossibilité d'ouvrir le fichier contenant le code à compiler, le dépassement de la capacité d'un tableau.... Une autre erreur pouvant être détectée par l'analyseur lexicale est la fin du fichier de programme dans un commentaire (EOF_IN_COMM_ERR).

L'émission des messages d'erreurs est assurée par la procédure ERREUR à l'aide du tableau MESSAGES_ERREUR :

```
type
```


TAB. 4.1 – *Messages d'erreurs PP1.*

1	identificateur attendu	ID_ERR
2	PROGRAM attendu	PROGRAM_ERR
3	) parenthèse fermante attendue	PAR_FER_ERR
4	end attendu	END_ERR
5	; attendu	PT_VIRG_ERR
6	= attendu	EGAL_ERR
7	begin attendu	BEGIN_ERR
8	erreur dans la partie déclaration	ERR_IN_DECL
9	, attendue	VIRG_ERR
10	erreur dans une constante	ERR_IN_CONST
11	:= attendu	AFFEC_ERR
12	then attendu	THEN_ERR
13	do attendu	DO_ERR
14	erreur dans un facteur (expression erronée)	ERR_IN_EXPR
15	identificateur déclaré deux fois	ERR_DBL_ID
16	identificateur non déclaré	ERR_NO_ID
17	nombre attendu	NUM_ERR
18	affectation non permise	ERR_NO_AFFEC
19	constante entière dépassant les limites	ERR_NUM_DEPASS
20	division par zéro	ERR_DIV_ZERO
21	. point attendu	POINT_ERR
22	(parenthèse ouvrante attendue	PAR_OUV_ERR

```
   ERREURT = (ID_ERR, PROGRAM_ERR, PAR_FER_ERR, END_ERR,
      PT_VIRG_ERR, EGAL_ERR, BEGIN_ERR, ERR_IN_DECL, VIRG_ERR
      ERR_IN_CONST, AFFEC_ERR, THEN_ERR, DO_ERR, ERR_IN_EXPR
      ERR_DBL_ID, ERR_NO_ID, NUM_ERR, ERR_NO_AFFEC,
      ERR_NUM_DEPASS, ERR_DIV_ZERO, POINT_ERR) ;
var
   MESSAGES_ERREUR : array [ERREURT] of STRING ;
```

La procédure ERREUR accepte maintenant un paramètre :

```
procedure ERREUR (ERRNUM:ERREURT) ;
```

La majorité des appels à ERREUR se font par TESTE, que l'on modifie ainsi :

```
procedure TESTE (T:TOKENS ; ERRNUM:ERREURT) ;
begin
   if TOKEN = T
      then NEXT_TOKEN
      else ERREUR (ERRNUM)
end ;
```

On modifie de même TESTE_ET_ENTRE et TESTE_ET_CHERCHE. Ces procédures ne produisent directement des erreurs que parce que le prochain token n'est pas un identificateur (pour l'instant), on a donc :

```
procedure TESTE_ET_CHERCHE (T:TOKENS ; PERMIS:CLASSET) ;
begin
   if TOKEN = T then
      begin
         CHERCHERSYM (PLACESYM, PERMIS) ;
         NEXT_TOKEN
      end
   else
      case T of :
         ID_TOKEN : ERREUR (ID_ERR) ;
      end
end ;

procedure TESTE_ET_ENTRE (T:TOKENS ; C:CLASSES) ;
begin
   if TOKEN = T then
      begin
         ENTRERSYM (C) ;
         NEXT_TOKEN
      end
   else
      case T of :
         ID_TOKEN : ERREUR (ID_ERR) ;
      end
end ;
```

La manipulation de la table des symboles peut produire des erreurs : identificateur déjà déclaré (ERR_DBL_ID) pour ENTRERSYM et identificateur non trouvé (non déclaré : ERR_NO_ID)

TAB. 4.2 – *Liste de corrections d'erreurs.*

token recherché	token trouvé	traitement
THEN_TOKEN	DO_TOKEN	substitution
DO_TOKEN	THEN_TOKEN	substitution
VIRG_TOKEN	PT_VIRG_TOKEN	substitution
PT_VIRG_TOKEN	VIRG_TOKEN	substitution
POINT_TOKEN	PT_VIRG_TOKEN	substitution
EGAL_TOKEN	AFFEC_TOKEN	substitution
AFFEC_TOKEN	EGAL_TOKEN	substitution
RELOP_TOKEN [a]	AFFEC_TOKEN	substitution par EGAL_TOKEN
PAR_FER_TOKEN	PT_VIRG_TOKEN	insertion
PAR_FER_TOKEN	ADDOP_TOKEN [b]	insertion
PAR_FER_TOKEN	MULOP_TOKEN [c]	insertion
PAR_OUV_TOKEN	ID_TOKEN	insertion

[a] En fait, un des tokens EGAL_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN, INF_EGAL_TOKEN, SUP_EGAL_TOKEN.
[b] Un des tokens PLUS_TOKEN, MOINS_TOKEN.
[c] Un des tokens MUL_TOKEN, DIV_TOKEN.

pour CHERCHERSYM. On modifie donc ces deux procédures en conséquence par un appel adéquat à ERREUR.

4.2 Récupération après erreur

La *récupération après erreur* est basée sur le modèle suivant. Quand on découvre une erreur, l'analyseur syntaxique élimine les symboles d'entrée les uns après les autres jusqu'à en rencontrer un qui appartienne à un ensemble de synchronisation. Usuellement, les tokens de synchronisation sont des délimiteurs tels que le point virgule ou le end dont le rôle dans le texte source est bien défini.

La récupération sur erreur peut être implantée comme suit. à chaque appel de procédure analysant un non-terminal de la grammaire, on passe un ensemble des tokens de synchronisation (paramètre SYNCHRO_TOKENS) ; en cas d'erreur, la procédure est chargée de se synchroniser sur un de ces tokens. De plus, au retour, la procédure appelée doit informer l'appelante de la bonne analyse ou de la synchronisation réalisée (paramètre ETAT).

4.3 Correction d'erreurs

Selon l'approche de la *correction d'erreur*, quand une erreur est découverte, l'analyseur syntaxique peut effectuer des corrections locales, c'est-à-dire qu'il peut modifier un token afin de permettre la poursuite de l'analyse. Une correction locale typique consisterait à remplacer une virgule par un point-virgule, à détruire un point-virgule excédentaire, ou à insérer un point-virgule manquant.

Il est préalablement nécessaire de définir une liste des corrections, suppressions et ajouts envisageables. Pour PP1, on peut commencer par la liste de la table 4.2.

Il est nécessaire de vérifier que les substitutions ainsi réalisées sont valides dans tous les cas ; on modifie la procédure TESTE en conséquence, cf. figure 4.1. La procédure UNGET_TOKEN

```
procedure TESTE (T:TOKENS ; ERRNUM:ERREURT) ;
var CORRECTION : booleen ;
begin
   CORRECTION := false ;
   if TOKEN = T then NEXT_TOKEN
   else begin
      case T of
         THEN_TOKEN :
            if TOKEN = DO_TOKEN
            then begin
               (* substitution *)
               CORRECTION := true ;
               TOKEN = THEN_TOKEN ;
               ERREUR_MESS (THEN_ERR)
            end ;
         DO_TOKEN : (* ... *)
         VIRG_TOKEN : (* ... *)
         PT_VIRG_TOKEN : (* ... *)
         POINT_TOKEN : (* ... *)
         AFFEC_TOKEN : (* ... *)
         (* RELOP_TOKEN *)
         EGAL_TOKEN,
         DIFF_TOKEN,
         INF_TOKEN,
         SUP_TOKEN,
         INF_EGAL_TOKEN,
         SUP_EGAL_TOKEN : (* ... *)
         PAR_FER_TOKEN :
            if TOKEN in [PT_VIRG_TOKEN, ADDOP_TOKEN, MULOP_TOKEN]
            then begin
                (* insertion du token recherche *)
                CORRECTION :=  true ;
                UNGET_TOKEN ;
                TOKEN = PAR_FER_TOKEN ;
                ERREUR_MESS (PAR_FER_ERR)
            end ;
         PAR_OUV_TOKEN : (* ... *)
      end ; (* case *)
      if not CORRECTION
      then ERREUR (ERRNUM)
   end (* else *)
end ;
```

Fig. 4.1 – *Procédure* `TESTE`*, prise en compte de la correction d'erreurs.*

met à jour une structure telle que le prochain appel de `NEXT_TOKEN` retourne le token `TOKEN` et non un token lu sur le programme traité.

Lors de la mise en place de la correction d'erreurs, le concepteur du compilateur doit assurer de ne pas tomber dans une boucle infinie par des insertions répétées de tokens.

4.4 Travaux dirigés et travaux pratiques

1. Identifier les erreurs possibles d'un programme PP1 et dresser une liste du type de celle de la table 4.1. Compléter chacun des appels de la procédure `ERREUR` par un type d'erreur. Modifier la procédure `ERREUR` pour prendre en compte cette information. Pour chacune des erreurs identifiées, définir un programme PP1 la contenant et le compiler.
2. La facilité offerte par `UNGET_TOKEN` et `NEXT_TOKEN` de revenir sur la lecture d'un token peut-elle nécessiter, lors de la compilation d'un programme PP1, la mémorisation de plusieurs tokens? Implanter cette facilité.
3. Définir, pour chaque appel de procédure d'analyse, l'ensemble de synchronisation qui convient. Implanter le mécanisme de récupération sur erreur décrit dans ce chapitre.
4. Assurez vous de la validité de la correction d'erreurs donnée dans ce chapitre. Implanter cette correction sur le modèle proposé.
5. Il a été dit que l'insertion de token dans une phase de corrections d'erreurs pouvait mener à une récursion infinie. Trouver un tel exemple de correction et le programme erroné correspondant entraînant une telle situation.

Chapitre 5

Génération de code

Nous allons compléter l'analyseur sémantique construit au chapitre 3 pour générer le P-Code correspondant au programme PP1 analysé.

Une première chose à faire est de décider de l'allocation des variables. Ensuite, chaque reconnaissance d'une règle de la grammaire déclenche la génération de P-Code à l'aide des procédures `GENERER1` (génération d'une instruction P-Code sans opérande) et `GENERER2` (génération d'une instruction P-Code à un opérande) dont l'écriture est détaillée à la fin de ce chapitre.

5.1 Allocation mémoire

Au niveau du langage PP1 (comme dans les autres langages de haut niveau), on ne se préoccupe pas de la gestion de la mémoire ; on manipule celle-ci par l'intermédiaire de symboles.

C'est le rôle du compilateur d'*allouer* ces symboles en mémoire. à chaque symbole, le compilateur doit associer un emplacement mémoire dont la taille dépend du type du symbole.

Une manière simple et naturelle de faire est de choisir les adresses au fur et à mesure de l'analyse des déclarations en incrémentant un *offset* qui indique la place occupée par les déclarations précédentes (variable `OFFSET`).

à la fin des déclarations, il est possible de déterminer l'emplacement mémoire à réserver dans la pile au début de l'exécution du programme (instruction P-Code `INT`).

Pour chaque symbole, son adresse d'allocation est stockée dans la table des symboles :

```
var
   TABLESYM : array [TABLEINDEX] of record
      NOM : ALFA ;
      CLASSE : CLASSES ;
      ADRESSE : integer
   end ;
   OFFSET : integer ;
```

On modifie la procédure `ENTRERSYM` pour tenir compte de cette allocation mémoire :

```
procedure ENTRERSYM(C:CLASSES) ;
begin
   if DERNIERSYM = INDEXMAX then ERREUR ;
   DERNIERSYM := DERNIERSYM + 1 ;
```



```
   with TABLESYM [DERNIERSYM] do begin
      NOM := SYM ;
      CLASSE := C ;
      if C = VARIABLE then begin
         ADRESSE := OFFSET ;
         OFFSET := OFFSET + 1
      end
   end
end ;
```

Le champ `ADRESSE` n'est utilisé que pour les variables ; néanmoins, on l'utilisera pour stocker la valeur des constantes ; on modifie la procédure `CONSTS` en conséquence (à faire en exercice).

5.2 Génération de début et fin de programme

Une fois l'allocation des données réalisée, il est nécessaire de réserver l'emplacement suffisant dans la pile P-Code. Cette réservation est faite lors de l'analyse d'un `BLOCK` par la génération d'une instruction P-Code `INST` :

```
procedure BLOCK ;
begin
   OFFSET := 0 ;
   if TOKEN = CONST_TOKEN then CONSTS ;
   if TOKEN = VAR_TOKEN then VARS ;
   GENERER2 (INT, OFFSET) ;
   INSTS
end ;
```

Lors de la terminaison de l'analyse d'un programme, il est nécessaire de générer une instruction P-Code d'arrêt du programme, `HLT` :

```
procedure PROGRAM ;
begin
   TESTE (PROGRAM_TOKEN) ;
   TESTE_ET_ENTRE (ID_TOKEN, PROGRAMME) ;
   TEST (PT_VIRG_TOKEN) ;
   BLOCK ;
   GENERER1 (HLT) ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

5.3 Génération des expressions

Dans un analyseur récursif descendant comme celui que nous construisons, lorsque l'on termine une procédure d'analyse c'est que l'on a déjà analysé correctement les phrases correspondant aux procédures appelées dans cette procédure.

Par exemple lors de l'analyse de l'expression `a + b`, `EXPR` va appeler `TERM` deux fois, une fois pour analyser `a` et une fois pour analyser `b` ; l'analyse du `+` se fait au niveau de `EXPR`.

Si les deux appels réussissent (c'est le cas dans notre exemple) et que le + est bien reconnu, EXPR réussit.

Lorsque la génération est dirigée par la syntaxe, on adopte le même raisonnement : lorsqu'une procédure se termine, on considère que la génération des phrases analysées par les procédures appelées est terminée. Il ne reste plus qu'à générer le code pour l'addition, c'est-à-dire simplement l'instruction P-Code ADD.

5.3.1 Génération des facteurs (FACT)

On commence donc par le génération des facteurs pour lesquels on laisse sur la pile P-Code une valeur :

```
procedure FACT ;
begin
   if TOKEN = ID_TOKEN then begin
      TESTE_ET_CHERCHE (ID_TOKEN, [CONSTANTE, VARIABLE]);
      with TABLESYM [PLACESYM] do
         case CLASSE of
            CONSTANTE : GENERER2 (LDI, ADRESSE) ;
            VARIABLE : begin
               GENERER2 (LDA, ADRESSE) ;
               GENERER1 (LDV)
            end ;
            PROGRAMME : ;
         end
      end
   end ;
   else if TOKEN = NUM_TOKEN then begin
      GENERER2 (LDI, VAL) ;
      NEXT_TOKEN
   end
   else begin
      TESTE (PAR_OUV_TOKEN) ;
      EXPR ;
      TESTE (PAR_FER_TOKEN)
   end
end ;
```

5.3.2 Génération des termes (TERM)

De même, un terme laisse sur la pile P-Code la valeur du terme. Il est nécessaire de mémoriser le token correspondant à l'opération avant l'analyse de l'opérande gauche du terme (variable OP) :

```
procedure TERM ;
var OP : TOKENS ;
begin
   FACT ;
   while TOKEN in [MULT_TOKEN, DIV_TOKEN] do
```



```
      begin
         OP := TOKEN ; (* memorise l'operation *)
         NEXT_TOKEN ;
         FACT ;
         if OP = MUL_TOKEN
            then GENERER1 (MUL)
            else GENERER1 (DIV)
      end
end ;
```

5.3.3 Génération des expressions (EXPR)

L'analyse d'une expression laisse aussi une valeur sur la pile P-Code ; le code est similaire à celui de l'analyse des termes :

```
procedure EXPR ;
var OP : TOKENS ;
begin
   TERM ;
   while TOKEN in [PLUS_TOKEN, MOINS_TOKEN] do
      begin
         OP := TOKEN ; (* memorise l'operation *)
         NEXT_TOKEN ;
         TERM ;
         if OP = PLUS_TOKEN
            then GENERER1 (ADD)
            else GENERER1 (SUB)
      end
end ;
```

5.4 Génération des conditions

La génération des conditions (procédure COND) est calquée sur celle des expressions. à faire en exercice.

5.5 Génération des instructions simples

Il n'y pas de code à générer lors de l'analyse du bloc d'instructions (INSTS) ou d'une instruction (INST). On détaille ici la génération à réaliser lors de l'analyse d'une instruction d'affectation et des instructions d'entrée/sortie.

5.5.1 Génération d'une affectation

Une affectation `A := expression` est générée suivant le modèle

```
LDA <adresse de A>   empile l'adresse de A
<code>               empile la valeur de l'expression
STO                  stocke la valeur de l'expression dans A
```

Le P-Code `<code>` dépose la valeur de `expression` sur la pile ; il est généré lors de l'analyse de `expression` par l'appel `EXPR`. On a donc :

```
procedure AFFEC ;
begin
   TESTE_ET_CHERCHE (ID_TOKEN, VARIABLE) ;
   GENERER2 (LDA, TABLESYM [PLACESYM]. ADRESSE) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR ;
   GENERER1 (STO)
end ;
```

5.5.2 Génération des instructions d'entrée/sortie

Le code à générer pour une instruction d'écriture telle `write (e1, e2, e3)` est le suivant :

```
<code1>   empile la valeur de l'expression e1
PRN       imprime cette valeur
<code2>   empile la valeur de l'expression e2
PRN       imprime cette valeur
<code3>   empile la valeur de l'expression e3
PRN       imprime cette valeur
```

Les P-Codes `<code1>`, `<code2>` et `<code3>` sont générés lors de l'analyse des expressions `e1`, `e2` et `e3` par des appels à `EXPR`. On modifiera la procédure `ECRIRE` en conséquence.

Le code à générer pour une instruction de lecture telle `read (v1, v2, v3)` est le suivant :

```
LDA <adresse de v1>   empile l'adresse de la variable v1
INN                   lit un entier, le stocke dans v1
LDA <adresse de v2>   empile l'adresse de la variable v2
INN                   lit un entier, le stocke dans v2
LDA <adresse de v3>   empile l'adresse de la variable v3
INN                   lit un entier, le stocke dans v3
```

Modifier la procédure `LIRE` en conséquence.

5.6 Génération des instructions avec rupture de séquence

Nous étudions ici la génération de code pour les instructions de rupture de séquence (`if COND then INST` et `while COND do INST`). Cette génération est faite sur les modèles suivants :

```
        code généré pour COND
        if not COND then goto LABEL
        code généré pour INST
LABEL   suite...
```

pour l'instruction `if` et

```
DEBUT   code généré pour COND
        if not COND then goto LABEL
        code généré pour INST
        goto DEBUT
LABEL   suite...
```


pour l'instruction `while`.

Le problème est que lors de la génération du saut conditionnel à `LABEL`, on ne connaît pas encore la valeur de ce label puisqu'on ne connaît pas a priori la longueur du code généré pour `INST`.

On va donc générer des instructions de saut incomplètes (sans numéro d'instruction) et mémoriser les numéros des instructions incomplètes pour pouvoir les compléter lorsque l'information sera disponible[1].

Cette facilité de compléter une instruction préalablement générée est offerte par l'intermédiaire des deux procédures `NEXT_INST` et `REMPLIR_INST` :

```
procedure NEXT_INST (var COMPT:integer) ;
procedure REMPLIR_INST (NUM_INST, DEP:integer) ;
```

la procédure `NEXT_INST` retourne dans le compteur indiquant le numéro de la prochaine instruction qui sera générée ; la procédure `REMPLIR_INST` complète l'instruction de saut `NUM_INST` avec le numéro d'instruction `DEP`.

On modifie donc la procédure `SI` en conséquence :

```
procedure SI ;
var SAUT, SUITE : integer ;
begin
   TESTE (IF_TOKEN) ;
   COND ;
   TESTE (THEN_TOKEN) ;
   NEXT_INST (SAUT) ;
   GENERER2 (BZE, 0) ; (* 0 car incomplet *)
   INST ;
   NEXT_INST (SUITE) ;
   REMPLIR_INST (SAUT, SUITE)
end ;
```

On fait de même pour la procédure `TANTQUE` :

```
procedure TANTQUE ;
var DEBUT, SAUT, SUITE ; integer ;
begin
   TESTE (WHILE_TOKEN) ;
   NEXT_INST (DEBUT) ;
   COND ;
   TESTE (DO_TOKEN) ;
   NEXT_INST (SAUT) ;
   GENERER2 (BZE, 0) ; (* 0 car incomplet *)
   INST ;
   GENERER2 (BRN, DEBUT) ;
   NEXT_INST (SUITE) ;
```

1. La technique de génération de code détaillée ici est directement liée au langage cible utilisé, le P-Code défini au chapitre 1. Dans le cas de génération d'un langage cible dans lequel les labels d'instruction sont définis par le programmeur et non implicites (n° d'instruction), la technique de génération de code est différente. L'instruction de saut peut être générée complètement avec un label défini au sein du compilateur ; ce label est mémorisé et sera utilisé pour l'écriture explicite du label de l'instruction sur laquelle est effectuée le branchement.

```
   REMPLIR_INST (SAUT, SUITE)
end ;
```

5.7 Procédures de génération de P-Code

On reprend les déclarations de la section 1.6 :

```
type MNEMONIQUES = (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,LSS,GEQ,LEQ,
        PRN,INN,INT,LDI,LDA,LDV,STO,BRN,BZE,HLT);
     INSTRUCTION = record
        MNE : MNEMONIQUES;
        SUITE : integer
     end
var  PCODE : array [0 .. TAILLECODE] of INSTRUCTION;
     PC : integer;
```

Les fonctions de génération de code GENERER1 et GENERER2 s'écrivent simplement :

```
procedure GENERER1 (M:MNEMONIQUES) ;
begin
   if PC = TAILLECODE then ERREUR ;
   PC := PC + 1 ;
   with PCODE [PC] do
      MNE := M
end ;


procedure GENERER2 (M:MNEMONIQUES ; A:integer) ;
begin
   if PC = TAILLECODE then ERREUR ;
   PC := PC + 1 ;
   with PCODE [PC] do begin
      MNE := M ;
      SUITE := A
   end
end ;
```

Les procédures NEXT_INST et REMPLIR_INST s'écrivent :

```
procedure NEXT_INST (var COMPT:integer ) ;
begin
   COMPT := PC + 1
end ;


procedure REMPLIR_INST (NUM_INST, DEP : integer) ;
begin
   PCODE [NUM_INST]. SUITE := DEP
end ;
```


5.8 Travaux dirigés et travaux pratiques

1. Implanter l'allocation des variables en mémoire et stockage des valeurs des constantes dans la table des symboles : modification des procédures `ENTRERSYM` et `CONSTS` (section 5.1). On testera la réalisation par l'extension de l'affichage de la table des symboles (exercice 4 page 38).
2. Ajouter la possibilité de déclarer des constantes avec une valeur signée et des constantes définies par des constantes précédemment déclarées, comme par exemple
   ```
   const
      MAX = +456 ;
      MIN = -456 ;
      PETIT = -MAX ;
   ```
3. Implanter les procédures de génération de P-Code `GENERER1` et `GENERER2` ainsi que les procédures utilitaires de générations de P-Code associées (section 5.7). Sous quelle forme générer le code ? Si le code généré doit être rechargé par le chargeur de P-Code, il paraît plus intéressant de passer par une forme non lisible (par un humain !), mais plus compacte du code. Proposer un tel codage.
   En complément, et pour faciliter le test du compilateur ou la mise au point de programmes PP1, il est aussi intéressant de pouvoir obtenir une forme lisible du P-Code généré. Quelles solutions sont envisageables pour ce faire ? En implanter une.
4. Lorsqu'une erreur est détectée, il est nécessaire de poursuivre la compilation. Il est cependant normal de ne pas continuer la génération de code. Pourquoi ? Mettre en œuvre cette interruption de la génération.
5. Implanter la génération de code pour les expressions `EXPR` et les instructions simples (affectation, lecture écriture). Tester cette implantation.
6. On implantera maintenant les conditions `COND` et les structures de contrôles présentées : **`if`** et **`while`**.
7. On complétera les structures de contrôle **`if`** et **`while`** par des constructions **`repeat`** ... **`until`** et **`if`** ... **`then`** ... **`else`**. En définir la syntaxe, la sémantique et l'implantation.
8. Définir et implanter une structure à choix multiples telle le `case` Pascal. Que faire lorsqu'aucune des « branches » du case ne convient ? Envisager la possibilité d'une alternative **`otherwise`** qui est exécutée dans ce cas.
9. Comment introduire une boucle `for` à la Pascal ? En Pascal, la variable de contrôle de la boucle ne peut pas être modifiée à l'intérieur de la boucle. Comment imposer cette contrainte ?
   Pourquoi, à votre avis, la valeur de l'indice des boucles `for` n'est pas définie à la sortie de la boucle en Pascal ?
   Certains pensent que la variable de contrôle de la boucle doit être une vraie variable locale et donc être implicitement déclarée au début de la boucle. Comment prendre cela en compte ?
10. Comment intégrer une instruction `goto` en PP1 ?

Deuxième partie

Extensions à des types complexes

Cette partie propose différentes extensions de PP1 pour la manipulation de types complexes. Nous introduisons tout d'abord des variables de types tableaux dans le langage ; puis nous considérons des variables de type enregistrement. Enfin nous étendons le langage par la manipulation de types construits.

Chapitre 6

Variables de type tableau

6.1 Grammaire et sémantique

On définit le langage PP2 comme une extension de PP1 autorisant la déclaration et la manipulation de tableaux simples. On modifie la règle `VARS` (page 28) pour autoriser la déclaration de tableaux ;

```
VARS ::= var DECL { , DECL } ; | ε
DECL ::= ID | ID [ NUM ]
```

Sémantiquement, cela permet la déclaration de tableaux d'entiers de dimension 1 dont la borne inférieure est toujours zéro et la borne supérieure est indiquée par la constante `NUM` (qui doit être strictement positive). Le nombre d'éléments est immédiatement calculé à partir de ces deux informations ; il n'existe pas de tableau à un élément.

On modifie les règles dans lesquelles figure le non-terminal `ID` (à savoir `AFFEC`, `LIRE` et `FACT`) pour autoriser la référence à un élément de tableau en introduisant une règle `VAR` :

```
VAR   ::= ID | ID [ EXPR ]
AFFEC ::= VAR := EXPR
LIRE  ::= read ( VAR { , VAR } )
FACT  ::= VAR | NUM | ( EXPR )
```

Sémantiquement, une variable `VAR` doit toujours référencer une variable simple ou un élément de tableau. Cela revient à dire que les noms de tableaux sont obligatoirement indicés dans une instruction. (En particulier, l'affectation directe de tableau n'est pas permise.)

6.2 Table des symboles

Lors de l'analyse d'une déclaration, on mémorise dans la table des symboles la taille du tableau (nombre d'éléments). Pour les symboles dénotant des variables simples, on mettra une taille à 1 (il n'y a pas de conflits avec les tableaux, les tableaux d'un élément n'existant pas en PP2).

```
var
   TABLESYM : array [TABLEINDEX] of record
      NOM : ALFA ;
      CLASSE : CLASSES ;
      ADRESSE : integer ;
      TAILLE : integer
```



```
   end ;
```

On modifie la procédure `TESTE_ET_ENTRE` pour mettre à jour cette table des symboles :

```
procedure TESTE_ET_ENTRE (T:TOKENS; C:CLASSES) ;
begin
   if TOKEN = T
   then if C <> VARIABLE
      then begin ENTRERSYM (C) ; NEXT_TOKEN end
      else ENTRERVARIABLE (C)
   else ERREUR
end ;
```

La procédure `ENTRERVARIABLE` se charge de l'entrée dans la table des symboles d'une variable simple ou d'un tableau.

```
procedure ENTRERVARIABLE (C:CLASSES) ;
begin
   if DERNIERSYM = INDEXMAX then ERREUR ;
   DERNIERSYM := DERNIERSYM + 1 ;
   with TABLESYM [DERNIERSYMB] do begin
      NOM := SYM ;
      CLASSE := C ;
      TAILLE := 1 ;
      ADRESSE := OFFSET
   end ; (* with *)
   NEXT_TOKEN ;
   if TOKEN = CROCH_OUV_TOKEN then begin
      (* c'est un tableau *)
      NEXT_TOKEN ;
      TESTE (NUM_TOKEN) ;
      TESTE (CROCH_FER_TOKEN) ;
      TABLESYM [DERNIERSYM]. TAILLE  := VAL + 1
   end ;
   OFFSET := OFFSET + TABLESYM [DERNIERSYM]. TAILLE
end ;
```

6.3 Génération de code

La génération de code pour accéder à un élément de tableau doit inclure la génération du code pour calculer l'adresse de cet élément. En effet, cette adresse ne peut être calculée à la compilation car l'élément accédé dépend de l'exécution.

Le code généré pour `t [expression]` doit permettre d'obtenir sur la pile l'adresse du (`expression`+1)$^{\text{ième}}$ élément du tableau `t` ; c'est-à-dire `adresse (t) + expression` !

Pour cela on doit générer le P-Code :

```
LDA adresse (t)
<code pour expression>
ADD
```

Donc, par exemple, pour l'affectation `t [expression] := expr2`, on génère:

```
LDA adresse (t)
<code pour expression>
ADD
<code pour expr2>
STO
```

D'où la modification de la procédure `AFFEC`:

```
procedure AFFEC ;
begin
   TESTE_ET_CHERCHE (ID_TOKEN, VARIABLE) ;
   GENERER2 (LDA, TABLESYM [PLACESYM]. ADRESSE) ;
   if TABLESYM [PLACESYM]. TAILLE > 1
      then begin
      (* On a un tableau, il faut un [ *)
         TESTE (CROCH_OUV_TOKEN) ;
         EXPR ;
         TESTE (CROCH_FER_TOKEN) ;
         GENERER1 (ADD) ;
      end ;
   TESTE (AFFEC_TOKEN) ;
   EXPR ;
   GENERER1 (STO)
end ;
```

On modifie de même les procédures `FACT` et `LIRE` qui sont les seules à référencer des noms de tableaux.

6.4 Travaux dirigés et travaux pratiques

1. Modifier le compilateur `pp1` (qui devient `pp2`) pour autoriser la manipulation de tableaux simples.
2. Modifier la grammaire de PP2 pour autoriser l'utilisation de constantes dans la déclaration de tableaux; par exemple:

   ```
   program tab_const ;
   const TAILLE = 64 ;
   var tab [ TAILLE ] ;
   ```

   Prendre cette nouvelle grammaire en compte pour modifier le compilateur `pp2`.

   Il est aussi possible d'autoriser les tailles à être des expressions simples et évaluables à la compilation formées à base de constantes déclarées et/ou de littéraux. Qu'en pensez-vous. Comment l'implémenter? (Se reporter à l'exercice 2, page 52.)
3. Lors de l'accès à un élément de tableau, on génère le code nécessaire au calcul de son adresse car, de manière générale, celle-ci ne peut pas être définie à la compilation. Cependant, dans certains cas, il est possible de déterminer cette adresse dès la compilation. Il n'est donc plus nécessaire de générer un tel code. Quels sont les cas où cela est possible? Comment prendre en compte cette optimisation dans le compilateur `pp2`?

4. On aurait pu étendre la syntaxe de PP1 de la manière suivante:

   ```
   VARS ::= var DECL { , DECL } ; | ε
   DECL ::= ID | ID [ NUM ] | ID [ NUM .. NUM ]
   ```

   Cela permet la déclaration de tableaux d'entiers indicés par un intervalle de valeurs.

   (a) Que doit-on mémoriser dans la table des symboles?

   (b) Réécrire les procédures `ENTRERVARIABLE` et `AFFEC` pour de tels tableaux.
5. On peut généraliser la manipulation de tableaux à la manipulation de tableaux à plusieurs dimensions.

   (a) Proposer une syntaxe pour de tels tableaux.

   (b) Quelle est la structure en mémoire pour de tels tableaux? Comment obtient-on l'adresse d'un élément d'un tel tableau?

   (c) Quelles informations doivent donc être rangées dans la table des symboles lors de la déclaration d'un tableau multi-dimensionnel?
6. Lors de l'utilisation des tableaux de dimension 1 décrits dans ce chapitre, on ne se soucie pas d'éventuels débordements d'indices. Faites en sorte que cette erreur soit signalée. Dans certains cas on peut signaler l'erreur dès la compilation; envisagez aussi cette solution (cf. l'exercice 3 ci-dessus).

   Dans le cas général, lorsque l'on dispose de la valeur de l'indice d'un tableau sur la pile, il est nécessaire de comparer cette valeur avec la borne supérieure du tableau et de laisser cette valeur sur la pile pour la suite du traitement. Comment réaliser cette opération en P-Code? Est-il imaginable d'étendre le P-Code par une instruction appropriée?

   En cas d'erreur pour débordement, on s'arrêtera sur une instruction P-Code d'arrêt sur erreur `OOB` (out of bounds).
7. Il est possible d'étendre la grammaire de PP2 pour autoriser l'affectation directe entre tableaux de même taille. Pour permettre cette affectation entre tableau, on introduit une nouvelle instruction P-Code:

   ```
   MOV n
   ```

   Cette instruction attend deux adresses sur la pile (`a1` au sous-sommet et `a2` au sommet) et réalise la recopie de `n` valeurs lues à partir de l'adresse `a2` dans les adresses successives à partir de `a1`.

Chapitre 7

Variables de type enregistrement

7.1 Grammaire et sémantique

On définit le langage PP3 comme une (nouvelle) extension de PP1 autorisant la déclaration et la manipulation de structures (enregistrements) simples ; on abandonne donc temporairement les tableaux. On modifie la règle `VARS` (page 28) pour autoriser la déclaration de structures ;

```
VARS  ::= var DECLS ; | ε
DECLS ::= DECL {, DECL}
DECL  ::= ID | ID ( DECLS )
```

Sémantiquement, cela permet la déclaration de structures composées d'enregistrements qui sont eux-mêmes soit des variables simples, soit des structures.

Par exemple,

```
var A, B (C, D), E, F (A, B (C, D)) ;
```

définit deux variables simples `A` et `E`, et deux variables structures `B` et `F`. La structure `B` déclare deux variables entières simples tandis que l'enregistrement `F` déclare une variable simple et un enregistrement de deux variables.

On modifie les règles dans lesquelles figure le non-terminal `ID` (à savoir `AFFEC`, `LIRE` et `FACT`) pour autoriser la référence à un champ d'enregistrement par une notation pointée (règle `VAR`) :

```
VAR   ::= ID {. ID}
AFFEC ::= VAR := EXPR
LIRE  ::= read ( VAR { , VAR } )
FACT  ::= VAR | NUM | ( EXPR )
```

Sémantiquement, une variable `VAR` doit toujours référencer une variable simple ; en particulier, les affectations directes entre structures ne sont pas permises.

7.2 Table des symboles

Les déclarations sont des *arbres*. Aussi, on les mémorisera dans la table des symboles sous forme d'arbres. Pour faciliter l'implantation dans la table des symboles, on associe à chaque variable un lien horizontal et un lien vertical. Le lien horizontal référence un frère dans la déclaration, le lien vertical référence le fils aîné. La figure 7.1 est une représentation


```
A ---> B --------------> E ---> F ---> *
|      |                 |      |
V      V                 V      V
*      C ---> D ---> *   *      A ---> B ---> *
       |      |                 |      |
       V      V                 V      V
       *      *                 *      C ---> D ---> *
                                       |      |
                                       V      V
                                       *      *
```

FIG. 7.1 – *Représentation d'une déclaration d'enregistrements.*

	NOM	ADRESSE	LH	LV
0	A	0	1	*
1	B	1	4	2
2	C	0	3	*
3	D	1	*	*
4	E	3	5	*
5	F	4	*	6
6	A	0	7	*
7	B	1	*	8
8	C	0	9	*
9	D	1	*	*

FIG. 7.2 – *Table des symboles pour une déclaration d'enregistrements.*

de l'exemple de déclaration donné ci-dessus. On remarque que les feuilles n'ont pas de lien vertical. On code cette structure dans la table des symboles par l'ajout de deux champs `LV` et `LH`.

```
var
   TABLESYM : array [TABLEINDEX] of record
      NOM : ALFA ;
      CLASSE : CLASSES ;
      ADRESSE : integer ;
      LV : integer ;
      LH : integer
   end ;
```

On prend comme convention que le champ `ADRESSE` représente un déplacement par rapport à l'adresse du père dans l'arbre et représente une adresse absolue (en fait relative à 0 !) pour le premier niveau de déclaration (dont les variables simples). Pour l'exemple

```
var A, B (C, D), E, F (A, B (C, D)) ;
```

on aura la table des symboles de la figure 7.2. Dans cette table, une `*` signifie qu'un lien est terminal. On le codera par une valeur invalide, par exemple -1. La déclaration de l'exemple nécessite une allocation de 7 entiers (variable `OFFSET`).

Il faut écrire les deux procédures d'analyse des déclarations `DECLS` et `DECL`. Ces procédures retournent par un paramètre `BASE` l'adresse dans la table des symboles de la première entrée analysée. Cette adresse est utilisée pour rattacher la déclaration à son père dans l'arbre. Le code des deux procédures est donné aux figures 7.4 et 7.3.

7.3 Génération de code

La génération de code pour accéder à un champ de structure utilise l'adresse de ce champ. Cette adresse peut être définie lors de la compilation. Étant donnée la manière dont on a construit la table des symboles, pour obtenir l'adresse d'une variable il suffit d'*additionner* les champs `ADRESSE` des `ID` rencontrés.

On utilisera la procédure `CHERCHERSYM` modifiée :

```
procedure CHERCHERSYM (var INDEX: TABLEINDEX ;
                       PERMIS: CLASSET ;
                       DEBUT: TABLEINDEX) ;
```

Le paramètre `DEBUT` indique l'entrée dans la table des symboles à partir de laquelle rechercher le symbole en suivant les liens `LH`. Comme par le passé, cette procédure retourne dans son premier paramètre `INDEX` l'indice dans la table des symboles du symbole trouvé.

On écrit ensuite la procédure `VAR` d'analyse d'une variable qui laisse sur la pile P-Code l'adresse de la variable :

```
procedure VAR ;
var BASE : integer ; ADR : integer ;
begin
   if TOKEN <> ID_TOKEN then ERREUR ;
   else NEXT_TOKEN ;
   CHERCHERSYM (INDEX, [VARIABLE], 0) ; (* niveau 0 *)
   ADR := TABLESYM [INDEX]. ADRESSE ;
   while TOKEN = PT_TOKEN do
      begin
         NEXT_TOKEN ;
         if TOKEN <> ID_TOKEN then ERREUR
         else NEXT_TOKEN ;
         CHERCHESYM (INDEX, [VARIABLE], INDEX) ;
         ADR := ADR + TABLESYM [INDEX]. ADRESSE
      end ;
   GENERER2 (LDA, ADR)
end ;
```

Cette procédure est appelée par les trois procédures `FACT`, `AFFEC`, et `LIRE` que l'on modifiera en conséquence.

7.4 Travaux dirigés et travaux pratiques

1. Construire le compilateur `pp3` comme une modification de `pp1` implantant la manipulation d'enregistrements telle qu'elle est décrite dans ce chapitre. On veillera de plus à modifier la production de la table des symboles (exercice 4, page 38).

2. Les procédures `DECL` et `DECLS` données dans ce chapitre ne testent pas la présence éventuelle de doubles déclarations. Quelle information doit passer entre ces procédures pour détecter de telles déclarations. Implanter ce mécanisme de détection des déclarations fautives.
3. La manipulation d'enregistrements peut être la source d'autres erreurs. Dresser une liste de ces erreurs et en ajouter la détection dans le compilateur.

   Modifier le code de recouvrement d'erreurs pour l'étendre au langage PP3.

   De nouveaux cas de figures de corrections d'erreurs apparaissent-ils ? Les prendre en compte.

```
procedure DECLS (var BASE:integer) ;
var START : integer ; TBASE : integer ;
begin
   START := DERNIERSYMB ;
   DECL (TBASE) ;
   while TOKEN = VIRG_TOKEN do
      begin
         NEXT_TOKEN ;
         DECL (TBASE)
      end ;
   TABLESYM [TBASE]. LH := -1 ; (* fin du niveau de declaration *)
   BASE := START
end ;
```

FIG. 7.3 – *Procédure* DECLS, *analyse de déclarations d'enregistrements.*


```
procedure DECL (var BASE:integer) ;
var START : integer ; OLD_OFFSET : integer ;
begin
   if DERNIERSYM = INDEXMAX then ERREUR ;
   START := DERNIERSYMB ;
   DERNIERSYMB := DERNIERSYMB + 1 ;
   if TOKEN <> ID_TOKEN then ERREUR ;
   with TABLESYM [START] do begin
      NOM := SYM ;
      CLASSE := VARIABLE ;
      ADRESSE := OFFSET ;
      LH := -1 ; (* provisoirement *)
      LV := -1 (* provisoirement *)
   end ;
   NEXT\_TOKEN ;
   if TOKEN = PAR_OUV_TOKEN then begin
   (* les offset sont comptes a partir du pere *)
      OLD_OFFSET := OFFSET ; OFFSET := 0 ;
      NEXT_TOKEN ;
      DECLS (TABLESYM [START]. LV) ;
      if TOKEN <> PAR_FER_TOKEN then ERREUR
      else NEXT_TOKEN ;
      OFFSET := OLD_OFFSET + OFFSET
   end else
      OFFSET := OFFSET + 1 ;
   TABLESYM [START]. LH := DERNIERSYMB ;
   BASE := START
end ;
```

FIG. 7.4 – *Procédure* DECL, *analyse de déclarations d'enregistrements.*

Chapitre 8

Définition de types

8.1 Grammaire et sémantique

8.1.1 Définition et utilisation de types

On définit le langage PP4 comme une extension de PP1 incluant la définition de types. On modifie la grammaire de PP1 (page 28)

```
VARS ::= var DECL { ; DECL } ;
DECL ::= ID { , ID } : TYPE
TYPE ::= ID | int
```

Les `ID` de `TYPE` doivent correspondre à un type déclaré précédemment :

```
BLOCK    ::= CONSTS TYPES VARS INSTS
TYPES    ::= type TYPEDECL ; { TYPEDECL ; } | ε
TYPEDECL ::= ID = array [ CONST ] of TYPE
          |  ID = record CHAMPS { ; CHAMPS } end
CHAMPS   ::= ID { , ID } : TYPE
CONST    ::= ID | NUM
```

Cela permet la définition de types à partir des types de bases. Ces types sont utilisés pour la définition d'autres types et pour la déclaration de variables.

Sémantiquement, les types utilisés doivent être préalablement déclarés.

Les tableaux sont des tableaux à une dimension[1] de taille constante indiquée par la constante `CONST` représentant une valeur immédiate ou le nom d'une constante précédemment déclarée.

8.1.2 Exemple

Définition d'une pile de complexes :

```
const
   TAILLE = 100 ;
type
   COMPLEX = record
      RE, IM : int
```

1. Cependant il est possible de manipuler des tableaux de plusieurs dimensions par la déclaration de tableaux de tableaux...


```
   end ;
   COMPLEXTAB = array [ TAILLE ] of COMPLEX ;
   COMPLEXPILE = record
      SOMMET : int ;
      PILE : COMPLEXTAB
   end ;
var
   MAPILE : COMPLEXPILE ;
```

8.1.3 Utilisation des variables de types complexes

On modifie les règles de la grammaire de PP1 dans lesquelles figure le non-terminal `ID` pour autoriser la référence aux champs des enregistrements et aux éléments des tableaux :

```
AFFEC  ::= VAR := EXPR
LIRE   ::= read ( VAR { , VAR } )
FACT   ::= VAR | NUM | ( EXPR )
```

Le non-terminal `VAR` doit référencer une variable scalaire :

```
VAR ::= ID
     |  VAR . ID
     |  VAR [ EXPR ]
```

La notation `.` accède à un champ d'une structure ; la notation `[ ]` à un élément d'un tableau.

8.2 Table des symboles

Pour mémoriser les informations contenues dans de telles déclarations de types, le compilateur ne peut plus se contenter d'une table des symboles aussi simple que précédemment. Il est nécessaire de construire une table des symboles qui reflète la structure du type construit ; les déclarations de types pour une telle table des symboles sont données à la figure 8.1.

La déclaration de la pile de complexes définie à la section 8.1.2 créerait une structure de données interne au compilateur telle celle décrite à la figure 8.2.

La table des symboles est une liste de nœuds de type `IDENTIFICATEUR` ; la liste est chaînée par les champs `GLIEN` et `DLIEN` (liens droit et gauche).

Chaque `IDENTIFICATEUR` correspond au nom d'un type (`TYPECL`), d'une constante (`CONSTANTE`), ou d'une variable (`VARIABLE`) ; ce nom est contenu dans le champ `NOM`.

Le champ `IDTYPE` indique le type de l'objet. Pour une constante, on stocke sa valeur (`VALEUR`); pour une variable son adresse d'allocation dans la pile[2] si l'objet est un objet de premier niveau (déclaré en temps que variable dans le programme); si l'objet est un champ de structure, on stocke son adresse relative au début de l'enregistrement le contenant[3].

Un type est représenté par un nœud de type `TIPE`. Un type est `SCALAIRE`, `TABLEAU`, ou `ENREGISTREMENT`. Le champ `TAILLE` contient le nombre d'éléments d'un tableau, 1 pour un scalaire, la somme de la taille des champs du type pour un enregistrement.

Pour un scalaire, on stocke son `NOM` (ici uniquement `int`). Pour un tableau, on stocke le type de ses éléments (`TYPEELT`). Pour un enregistrement, on établit une liste chaînée de ses champs (`LISTECHAMPS`) qui sont des nœuds de type `IDENTIFICATEUR` reliés par le champ `SUIVANT`. Le champ `ADRESSE` est alors l'adresse relative du champ dans l'enregistrement.

2. Cette adresse est déterminée par le compilateur lors de l'allocation des variables.
3. Cette adresse est déterminée par la taille des champs précédant le champ en question dans la structure.

```
type
   TYPEPT = ^TIPE ;
   IDPT = ^IDENTIFICATEUR ;

   TYPECLASSES = (SCALAIRE, TABLEAU, ENREGISTREMENT) ;
   TIPE =
      record
         TAILLE : integer ;
         case CLASSE : TYPECLASSES of
            SCALAIRE : (
               NOM : IDPT
               ) ;
            TABLEAU : (
               TYPEELT : TYPEPT
               ) ;
            ENREGISTREMENT : (
               LISTECHAMPS : IDPT
               )
      end;

   IDCLASSES = (TYPECL, CONSTANTE, VARIABLE) ;
   IDENTIFICATEUR =
      record
         NOM : ALFA ;
         GLIEN, DLIEN, SUIVANT : IDPT ;
         IDTYPE : TYPEPT ;
         case CLASSE : IDCALSSES of
            TYPECL : () ;
            CONSTANTE : (
               VALEUR : integer
               ) ;
            VARIABLE : (
               ADRESSE : integer
               )
      end ;
```

Fig. 8.1 – *Déclarations de types pour la table des symboles.*

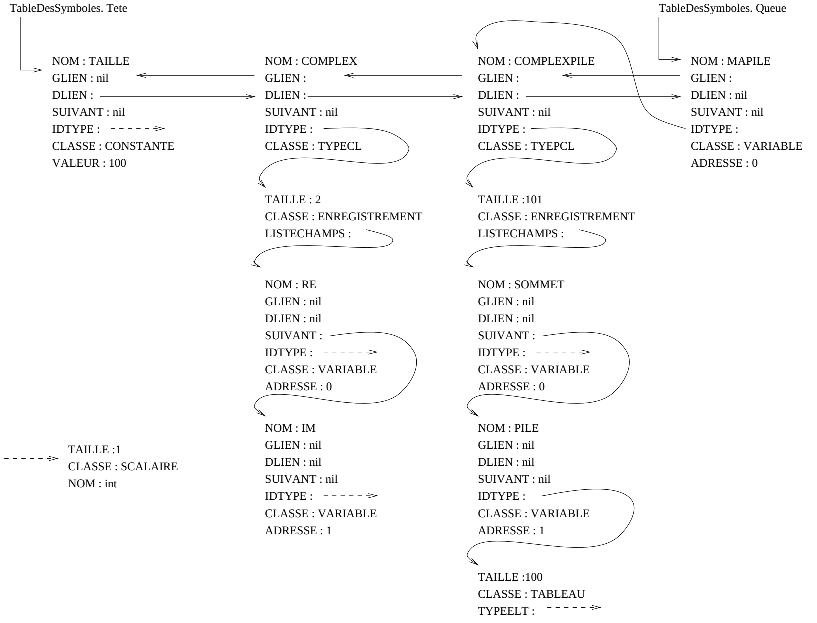

Fig. 8.2 – *Structure de la table des symboles pour la pile de complexes.*

```
1    LDA 0   empile l'adresse de base de MAPILE
2    LDA 1   empile l'adresse relative de PILE/MAPILE
3    ADD     → adresse de base de MAPILE. PILE
4    LDA 0   empile l'adresse de base de MAPILE
5    LDA 0   empile l'adresse relative de SOMMET/MAPILE
6    ADD     → adresse de MAPILE. SOMMET
7    LDV     → valeur de MAPILE. SOMMET
8    LDI 1   |
9    SUB     | enleve 1 pour l'indicage
10   LDI 2   empile la taille des elements de MAPILE. PILE
11   MUL     → deplacement pour acceder MAPILE. PILE [MAPILE. SOMMET]
12   ADD     → adresse de MAPILE. PILE [MAPILE. SOMMET]
13   LDA 1   empile l'adresse relative de IM/COMPLEX
14   ADD     → adresse de MAPILE. PILE [MAPILE. SOMMET]. IM
```

FIG. 8.3 – *P-code pour accéder à* `MAPILE. PILE [MAPILE. SOMMET]. IM`.

8.3 Génération de code

La génération de code doit prendre en compte la structure ainsi définie pour accéder aux différents éléments des objets du type.

Par exemple, lorsque le programmeur écrit

```
MAPILE. PILE [MAPILE. SOMMET]. IM
```

le compilateur doit générer le code calculant l'adresse de cette partie imaginaire du complexe en sommet de pile, et cela à partir de l'adresse de base de `MAPILE` et de la valeur de `MAPILE. SOMMET`, cette dernière n'étant pas connue avant l'exécution. Le code de la figure 8.3 réalise un tel adressage.

Il apparaît que le code généré peut être long et donc coûteux en temps d'exécution. Des langages offrent la possibilité de « factoriser » ce calcul d'adresse. C'est le cas de l'instruction `with` de Pascal :

```
with MAPILE. PILE [MAPILE. SOMMET] do
   begin
      RE := ...
      IM := ...
   end ;
```

Les douze premières instructions du P-Code de la figure 8.3 ne sont générées qu'une fois, l'adresse obtenue est utilisée pour chacun des accès à `RE` et `IM`.

8.4 Travaux dirigés et travaux pratiques

1. La table des symboles est représentée par une liste doublement chaînée : liens `DLIEN` et `GLIEN` (figures 8.1 et 8.2). Quelle est l'utilité d'un tel double chaînage?
2. Modifier la grammaire de PP1 pour accepter les définitions de types introduites ici. Prendre en compte cette nouvelle grammaire pour l'écriture de l'analyseur syntaxique de `pp4`. Implanter la gestion de la table des symboles et la modification des règles

sémantiques. Modifier la gestion des erreurs en conséquence. Prendre en compte les accès aux éléments de structure et de tableaux lors de la génération de code.
3. La grammaire proposée pour PP4 oblige la déclaration préalable d'un type nommé à chaque niveau. Pascal accepte aussi des déclarations de types « anonymes »; par exemple la déclaration

```
var
   MAPILE : record
      SOMMET : int ;
      PILE : array [TAILLE] of record
         RE, IM : int
      end
   end ;
```

Proposer une grammaire pour une telle déclaration. Modifier `pp5` en conséquence.
4. Soient les déclarations

```
type
   LISTS : array [10] of int ;
var
   X : LISTS ;
   A : array [10] of int ;
   B : array [10] of int ;
   Y : LISTS ;
```

Il peut sembler évident que les deux variables `A` et `B` soient de même type, et même que `X` et `Y` soient de même type que `A` et `B`. Cependant, cette équivalence, nommée *équivalence structurelle de types*, n'est pas facile à contrôler. L'*équivalence nominale de types* reconnaît uniquement les variables `X` et `Y` comme ayant le même type. Cette équivalence nominale est facile à vérifier : deux objets sont nominalement équivalent si leurs champs `IDTYPE` référence un même nœud `TIPE`[4].
Cette notion d'équivalence est primordiale si l'on désire autoriser l'affectation directe de structures complexes (enregistrements ou tableaux) [5]
   (a) Implanter ces affectations en considérant uniquement les affectations entre objets nominalement équivalents comme valides ;
   (b) Qu'en est-il des variables `A` et `B` d'une déclaration
   ```
   var
      A, B : array [10] of int ;
   ```
   (c) Considérer maintenant l'équivalence structurelle des types.
5. D'autres constructeurs de types sont proposés en Pascal ; considérons les intervalles :

```
const
   TAILLE = 100 ;
type
   INDICE = 1 .. TAILLE ;
   TAB = array [INDICE] of int ;
```

4. On notera, en particulier sur l'exemple, que le nœud correspondant au type prédéfini scalaire entier n'existe qu'en un exemplaire référencé par les différents `IDENTIFICATEUR`s de ce type.
5. Se reporter à l'exercice 7, page 60 pour les problèmes liés à de telles affectations.

Y a-t-il une difficulté majeure à leur implantation dans `pp5` ? Quelle garantie peut attendre le programmeur de leur utilisation ? Comment assurer cette garantie ?
Ces types sont parfois critiqués car ils mènent à des ambiguïtés. Par exemple, une valeur `34` peut être de type `0..45` et aussi `30..90`. Comment contrer cette objection ?

6. Pascal propose des enregistrements avec variantes. Quel est l'avantage d'une telle construction. Considérer l'allocation de ces enregistrements.
7. Les ensembles.
8. Pourquoi Pascal n'oblige-t-il pas la définition d'un type à apparaître avant la définition du type pointeur vers ce type ?
   Comment ces définitions peuvent-elles être prises en compte ? Comment construire la table des symboles ?
9. Comment implanter l'instruction `with` de Pascal (se référer à la section 8.3).
10. Proposer une implantation des tableaux de taille dynamique. Ces tableaux sont explicitement alloués et désalloués par le programmeur ; par exemple :

```
var
   TAB : array [dynamic] of int ;
   I : int ;

   read (I) ;
   allocate (TAB, I) ;
   ...
   desallocate (TAB) ;
```

   Il peut être intéressant d'associer un « entête » à chaque tableau dynamique. Quelles contraintes doivent vérifier l'allocation de tels tableaux ?

Troisième partie

Les procédures et fonctions

On introduit dans cette partie les procédures du genre Pascal. Cette introduction se fait en plusieurs phases. Dans un premier temps nous ne considérons que les procédures sans paramètres, ni déclarations locales et non imbriquées. Nous introduisons ensuite les déclarations locales. Puis nous considérons les procédures imbriquées. Enfin nous prenons en compte les paramètres. Le dernier chapitre traite des fonctions.

Chapitre 9

Procédures simples

On introduit donc ici les procédures simples, c'est-à-dire sans paramètres, sans déclarations locales, et non imbriquées. L'extension de PP1 ainsi construite se nomme PP5.

9.1 Grammaire et sémantique

Nous étendons la grammaire de PP1 par l'introduction d'une règle `PROCS` autorisant la définition de procédures :

```
PROGRAM ::= program ID ; BLOCK .
BLOCK   ::= CONSTS VARS PROCS INSTS
PROCS   ::= { proc ID ; INSTS ; }
```

On modifie également la règle `INST` pour inclure l'appel de procédure dans les instructions :

```
INST  ::= ... | APPEL | ...
APPEL ::= ID
```

l'identificateur `ID` de la règle `APPEL` doit être le nom d'une procédure précédemment définie.

9.2 Extension du P-Code

Si un *appel* de procédure peut se voir comme un `BRN` (branchement inconditionnel) vers la première instruction de la procédure, le *retour* de procédure ne peut être un `BRN` car on ne connaît jamais à l'exécution le numéro de l'instruction (chez l'appelant) suivant l'appel. Plus précisément, ce numéro dépend de l'appel ; il peut y en avoir plusieurs dans un programme.

On introduit donc deux nouvelles instructions P-Code :

`CAL i` empile le compteur d'instructions et réalise le branchement à l'instruction `i`

`RET` dépile un numéro d'instruction et continue à ce numéro

On étend donc l'interpréteur P-Code (section 1.6 , page 20et figure 1.13) en conséquence :

```
with INST do
   case MNE of
      ...
      CAL : begin SP := SP + 1 ; MEM [SP] := PC ; PC := SUITE end ;
      RET : begin PC := MEM [SP] ; SP := SP - 1 end ;
   end
```


9.3 Table des symboles

Les noms des procédures doivent être rangés dans la table des symboles. On ajoute une nouvelle classe, `PROCEDURE` (section 3.1, page 35).

```
type
   CLASSES = (PROGRAMME, CONSTANTE, VARIABLE, PROCEDURE) ;
   CLASSET = set of CLASSES ;
```

À chacune des entrées de classe `PROCEDURE` de la table des symboles, on associe le nom de la procédure (champ `ALFA`, cf. déclaration page 45) et le numéro de la première instruction P-Code générée pour cette procédure (champ `ADRESSE`).

9.4 Génération de code

La génération de code doit inclure la génération des différentes procédures. Après la génération de la première instruction de la procédure, il est nécessaire de mémoriser dans la table des symboles le numéro de cette instruction P-Code. à la fin de la procédure, on génère une instruction P-Code `RET`.

La génération pour une instruction PP5 d'appel de procédure consiste à rechercher le nom de la procédure dans la table des symboles et à générer l'instruction `CAL <adresse>`.

Deux remarques peuvent être faites sur la génération de code :

D'une part, les procédures sont traduites avant le programme principal. Le code généré pour un programme doit donc commencer par un saut inconditionnel vers le programme principal. On génère un `BRN` incomplet qui sera complété lorsqu'on va rencontrer le début du programme principal.

D'autre part, l'appel d'une procédure dans une procédure ne pose pas de problèmes particuliers. Il faut simplement que la procédure appelée soit définie avant la procédure appelante. Cette remarque peut être étendue à la récursivité (non croisée !) : une procédure peut s'appeler sans problèmes. Néanmoins, la récursivité de procédures sans paramètres ni déclarations locales n'est pas un outil très puissant.

9.5 Exemples

On donne ici deux exemples de programmes PP5 et leur traduction en P-Code.

9.5.1 Exemple simple

La compilation du programme PP5 (figure 9.1) utilise une table des symboles telles que celle-ci :

ALFA	CLASSE	ADRESSE
UN	PROGRAM	—
A	VARIABLE	0
B	VARIABLE	1
MUL	PROCEDURE	2
DEBUT	PROCEDURE	10

pour produire le P-Code de la figure 9.2.

```
program UN ;
var A, B ;
proc MUL ;
   begin
      A := A * B
   end ;
proc DEBUT ;
   begin
      read (A) ; read (B) ;
      MUL
   end ;
begin
   DEBUT ;
   write (A)
end.
```

FIG. 9.1 – *Exemple de code PP5 pour les appels de procédures.*

```
0   INT 2    réserve 2 emplacements pour A et B
1   BRN 16   saut au début du programme principal
2   LDA 0    début de la fonction MUL
3   LDA 0
4   LDV
5   LDA 1
6   LDV
7   MUL
8   STO
9   RET      fin de la fonction MUL
10  LDA 0    début de la fonction DEBUT
11  INN
12  LDA 1
13  INN
14  CAL 2    appel de la fonction MUL
15  RET      fin de la fonction DEBUT
16  CAL 9    début du programme principal (appel de la fonction DEBUT)
17  LDA 0
18  LDV
19  PRN
20  HLT      fin du programme principal
```

FIG. 9.2 – *Exemple de P-Code pour les appels de procédures.*

9.5.2 Exemple récursif

Ce deuxième exemple de programme PP5 illustre les appels récursifs (figure 9.3). La table des symboles utilisées lors de la compilation est la suivante :

ALFA	CLASSE	ADRESSE
RECUR	PROGRAM	—
T	VARIABLE	0
N	VARIABLE	1
CALCUL	PROCEDURE	2

Cette compilation produit le P-Code de la figure 9.4.

9.6 Travaux dirigés et travaux pratiques

1. Modifier l'interpréteur de P-Code pour intégrer les instructions `CAL` et `RET`.
2. Modifier les procédures `TEST_ET_ENTRE` et `TEST_ET_CHERCHE` de gestion de la table des symboles pour intégrer la gestion des entrées de type `PROCEDURE`.
3. Implanter la compilation des procédures telles que définies dans ce chapitre. En particulier, écrire le code de la procédure `PROCS`, et modifier la procédure `INST` pour intégrer l'appel de procédure.

```
program RECUR ;
var T, N ;
proc CALCUL ;
   begin
      read (N) ;
      T := T * 10 + N ;
      if N <> 0 then CALCUL
   end ;
begin
   T := 0 ;
   CALCUL ;
   write (T)
end.
```

Fig. 9.3 – *Exemple de code PP5 illustrant les appels récursifs de procédures.*


```
0   INT 2    réserve 2 emplacements pour T et N
1   BRN 20   saut au début du programme principal
2   LDA 1    début de la fonction CALCUL
3   INN
4   LDA 0
5   LDA 0
6   LDV
7   LDI 10
8   MUL
9   LDA 1
10  LDV
11  ADD
12  STO
13  LDA 1
14  LDV
15  LDI 0
16  NEQ
17  BZE 19
18  CAL 2    appel récursif à CALCUL
19  RET      fin de la fonction CALCUL
20  LDA 0    début du programme principal
21  LDI 0
22  STO
23  CAL 2    appel de CALCUL
24  LDA 0
25  LDV
26  PRN
27  HLT
```

Fig. 9.4 – *Exemple de P-Code pour les appels récursifs de procédures.*

Chapitre 10

Déclarations locales

On introduit donc ici les déclarations de variables locales aux procédures ; l'extension de PP5 ainsi construite se nomme PP6.

10.1 Grammaire et sémantique

Nous étendons la grammaire de PP5 par l'introduction d'une règle `PROCS` autorisant la définition de procédures pouvant inclure des déclarations de constantes et variables locales :

```
PROGRAM ::= program ID ; BLOCK .
BLOCK   ::= CONSTS VARS PROCS INSTS
PROCS   ::= { proc ID ; CONSTS VARS INSTS ; }
```

De même que dans PP5, on autorise l'appel de procédure dans les instructions :

```
INST  ::= ... | APPEL | ...
APPEL ::= ID
```

l'identificateur `ID` de la règle `APPEL` doit être le nom d'une procédure précédemment définie.

Sémantiquement, nous définissons une notion de *visibilité* des variables. Une variable locale n'est visible que dans la procédure qui la déclare. Une variable globale est visible dans une procédure s'il n'existe pas de variable locale portant le même nom.

10.2 Visibilité des variables

Pour respecter cette règle de visibilité des variables locales, il est nécessaire que les déclarations locales apparaissent dans la table des symboles *uniquement* durant l'analyse de la procédure correspondante.

On va donc gérer une table des symboles à deux niveaux. Le premier niveau contient les déclarations globales (y compris les déclarations de procédures, qui sont globales) ; le deuxième niveau contient les déclarations locales à la procédure en cours d'analyse. La limite entre ces deux niveaux est indiquée par la variable `LIMIT` ; les entrées dans la table des symboles d'indice supérieure à `LIMIT` sont locales.

On obtient la procédure `PROCS` d'analyse de déclarations de procédures de la figure 10.1.

10.3 Allocation des variables

Deux solutions sont possibles pour l'allocation des variables locales : l'allocation *statique* et l'allocation *dynamique sur la pile*.

10.3.1 Allocation statique

L'allocation statique associe un emplacement mémoire fixe dans la pile à chaque variable, que celle-ci soit globale ou locale. à chaque fois qu'une variable (globale ou locale) est déclarée, on incrémente l'`OFFSET`, l'adresse suivante est donc allouée à la variable. Pour le programme

```
program X ;
   var A, B ;
   proc Y ;
      var C, D ;
         ...
   proc Z ;
      var E, F ;
         ...
   begin
      ...
   end .
```

on obtient alors les adresses d'allocation dans la pile suivantes pour les différentes variables

Variable	Adresse
`A` de `X`	0
`B` de `X`	1
`C` de `Y`	2
`D` de `Y`	3
`E` de `Z`	4
`F` de `Z`	5

Toute variable a donc une adresse différente et toutes les adresses sont connues à la compilation.

On obtient le schéma de P-Code suivant :

```
   INT N       réservation pour toutes les variables (globales et locales)
   BRN PP
   <proc>
   ...
   RET
   <proc>
   ...
   RET
PP ...
   <program>
   ...
   HLT
```

Cette solution est celle choisie par le langage FORTRAN. Cette solution ne permet pas la récursivité, car à chaque appel d'une procédure ce sont toujours les mêmes adresses qui sont utilisées. Si l'on désire autoriser la récursivité (comme en Pascal ou en PP6 !), il faut des adresses différentes pour chaque appel récursif.

10.3.2 Allocation dynamique sur la pile

Afin d'associer des adresses différentes aux variables à chaque appel récursif, il devient impossible de connaître les adresses lors de la compilation (le nombre d'appels récursifs ne pouvant être connu). On va donc allouer dynamiquement sur la pile les variables locales. On obtient le schéma de P-Code suivant :

```
    INT N        réservation pour les variables globales
    BRN PP
    INT N1       réservation dynamique pour les variables de P1
    <proc>
    ...
    RET
    INT N2       réservation dynamique pour les variables de P2
    <proc>
    ...
    RET
PP  ...
    <program>
    HLT
```

Les adresses associées aux variables locales sont alors des adresses relatives au début de la zone allouée dynamiquement dans la pile. Pour le même programme

```
program X ;
   var A, B ;
   proc Y ;
      var C, D ;
         ...
   proc Z ;
      var E, F ;
         ...
   begin
      ...
   end .
```

on obtient alors les adresses d'allocation suivantes pour les différentes variables

Variable	Adresse
A de X	globale 0
B de X	globale 1
C de Y	relative 0
D de Y	relative 1
E de Z	relative 0
F de Z	relative 1

Modifier la procédure PROCS pour prendre en compte cette allocation des variables.

10.4 Accès aux variables locales

L'accès à une variable globale se fait par un `LDA i` avec `i` un déplacement par rapport à la base de la mémoire, c'est-à-dire par rapport à la base de la pile dans la zone réservée par le `INT n` du programme principal.

L'accès à une variable locale se fera par `LDL i` avec `i` un déplacement par rapport à la base de la zone mémoire réservée par le `INT n` exécuté à l'appel de la procédure.

Il se pose alors le problème suivant : comment connaître cette adresse de base lors de l'exécution ?

1. Lorsque l'on appelle une procédure, c'est simple : la base pour la procédure appelée est le pointeur de pile au moment de l'appel ;
2. Au retour de la procédure, il faut retrouver la base de la procédure appelante.

L'idée est de chaîner les zones de la pile. Par exemple, lorsque le programme principal a appelé la procédure `P1` qui a elle-même appelé la procédure `P2`, l'état de la pile est celui de la figure 10.2.

`BASE` est une variable de l'interpréteur de P-Code qui pointe sur la base de la zone de la procédure en cours d'exécution. `BASE` vaut zéro lorsque l'exécution est dans le programme principal.

On modifie en conséquence les instructions `CAL`, `RET` de l'interpréteur et on ajoute l'instruction `LDL`.

10.5 Travaux dirigés et travaux pratiques

1. Quels sont les avantages et inconvénients d'une allocation statique des variables locales telle qu'elle est réalisée en Fortran ?
2. Donner le P-Code à générer et l'évolution, lors de la compilation, de la table des symboles utilisée pour l'exemple de programme PP6 de la figure 10.3.
3. Même chose pour le programme récursif de la figure 10.4.
4. On modifiera le code de la procédure `PROCS` donnée à la figure 10.1 pour prendre en compte l'allocation des variables locales relativement à la base de la zone locale de mémoire.
5. Donner le nouveau code des instructions P-Code `CAL`, `RET`, et `LDL`.
6. Implémenter les références aux variables en fonction de leur classe : variable locale ou variable globale.

```
procedure PROCS ;
   var LIMIT : integer ;
   begin
      while TOKEN = PROC_TOKEN do begin
         NEXT_TOKEN ;
         (* on entre le nom de la procedure en global *)
         TESTE_ET_ENTRE (ID_TOKEN, PROCEDURE) ;
         TESTE (PT_VIRG_TOKEN) ;
         (* on memorise le dernier global *)
         LIMIT := DERNIERSYM ;
         if TOKEN = CONST_TOKEN then CONSTS ;
         if TOKEN = VAR_TOKEN then VARS ;
         INSTS ;
         TESTE (PT_VIRG_TOKEN) ;
         (* on oublie les declarations locales *)
         DERNIERSYM := LIMIT ;
         GENERER1 (RET)
      end
   end ;
```

FIG. 10.1 – *Procédure* `PROCS` *d'analyse de déclarations de procédures.*

```
|            |
|------------|
| variables  |
| locales de |
|    P2      |
|------------|
|   -------------+
|------------|   |
|    AR      |<--|-------- BASE
|------------|   |
| variables  |   |
| locales de |   |
|    P1      |   |
|------------|   |
|   -------------|----+
|------------|   |    |
|    AR      |<--+    |
|------------|        |
| variables  |        |
| globales   |        |
+------------+<-------+
```

FIG. 10.2 – *État de la pile aprés l'appel de la procédure* `P2`*, appelée par la procédure* `P1`*, elle-même appelée par le programme principal.*


```
program X ;
   var A ;
   proc Y ;
      var B ;
      begin
         read (B) ;
         A : = A + B
      end ;
   proc Z ;
      var B ;
      begin
         read (B) ;
         A : = A * B ;
         Y
      end ;
   begin
      read (A) ;
      Z ;
      write (A)
   end .
```

FIG. 10.3 – *Exemple de programme PP6.*

```
program INVERT ;
   var OK ;
   proc INV ;
      var N ;
      begin
         read (N) ;
         if N <> OK then INV ;
         write (N)
      end ;
   begin
      OK := 0 ;
      INV
   end .
```

FIG. 10.4 – *Exemple de programme PP6 récursif.*

Chapitre 11

Procédures imbriquées

On introduit donc ici les procédures imbriquées ; la version ainsi modifiée de PP6 s'appelle PP7.

11.1 Grammaire et sémantique

La grammaire de PP6 est modifiée de la manière suivante :

```
PROGRAM ::= program ID ; BLOCK .
BLOCK   ::= CONSTS VARS PROCS INSTS
PROCS   ::= { proc ID ; BLOCK ; }
```

Les procédures peuvent maintenant être imbriquées. Sémantiquement, la règle de visibilité est définie comme suit : une déclaration locale n'est visible que dans le bloc qui la déclare et les blocs englobés dans celui-ci.

11.2 Accès aux variables

Il est donc nécessaire de gérer la table des symboles en pile :

```
|               |
|---------------|
|               |   declarations du bloc en cours
|               |       d'analyse
|               |
|---------------|
|               |   declarations du bloc englobant
|               |
|---------------|
|               |
|               |
|               |   declarations globales
|               |
|               |
|               |
+---------------+
```

De plus, on associe à chaque entrée de la table des symboles le niveau de la déclaration (dans un champ `NIVEAU`, cf. déclaration page 35) :

```
+---------------+
```



```
|          niv0 |
| +-----------+ |
| |      niv1 | |
| |           | |
| | +-------+ | |
| | |  niv2 | | |
| | |       | | |
| | |       | | |
| | +-------+ | |
| +-----------+ |
|               |
| +-----------+ |
| |      niv1 | |
| |           | |
| |           | |
| +-----------+ |
+---------------+
```

L'accès aux variables est donc différent selon leur niveau :

- Si le numéro de niveau est 0, la variable est une variable globale. On y accède par un `LDA i`.
- Si le numéro de niveau est le numéro du niveau courant, la variable est locale à la procédure en cours. On y accède par une instruction `LDL i`.
- Si le numéro du niveau est différent de zéro et du niveau courant. De combien de zones faut-il descendre pour trouver la variable?

En solution à ce problème, la première idée peut être : il faut descendre de $nc - nv$ (nc : niveau courant, nv : niveau de la variable). C'est une idée fausse ; nous allons détailler l'accès à une telle variable sur un exemple.

Tout d'abord quelques éléments de réponse :

Il est impossible de connaître lors de la compilation l'adresse d'une telle variable. Par exemple, à cause de la récursivité.

D'autre part, les niveaux de procédures empilés sur la pile sont croissants, mais pas strictement croissants.

Regardons sur l'exemple suivant : on considère le programme de la figure 11.1 et l'accès à `A`, `C` et `B` dans l'instruction `C := B + A` au cours de l'analyse de `Y1` (niveau 2) ; on a la table des symboles suivante :

ALPHA	CLASSE	ADRESSE	NIVEAU
C	VARIABLE	0	2
Y1	PROCEDURE	adr2	1
B	VARIABLE	0	1
Y	PROCEDURE	adr1	0
A	VARIABLE	0	0
X	PROGRAMME	—	0

Comme le niveau de `A` est 0, on génère un `LDA 0`. Comme le niveau de `C` est égal au niveau courant, on génère un `LDL 0`.

à l'exécution, l'état de la pile sera celui de la figure 11.2. Le `B` qui doit être accédé est celui de `Y`, il est donc nécessaire de descendre de 2 zones (ce qui est différent de la différence des niveaux de déclaration : $2 - 1 = 1$)

Pour ce faire, on va chaîner (par un lien dit *statique*), à l'exécution, les positions de la pile qui correspondent à un niveau de déclaration. Dans ce cas, la différence de niveau correspond

```
program X ;            (* niveau 0 *)
   var A;              (* niveau 0 *)
   proc Y ;            (* niveau 0 *)
      var B ;          (* niveau 1 *)
      proc Y1 ;        (* niveau 1 *)
         var C ;       (* niveau 2 *)
         begin
            C := B + A
         end ;
      proc Y2 ;
         var B ;
         begin
            Y1
         end ;
      begin
         Y2
      end ;
   begin
      Y
   end .
```

FIG. 11.1 – *Exemple d'imbrication des niveaux dans un programme PP7.*

```
                                   |          |
                                   |----------|
                                   |    C     |   |
                                   |----------|   |
                       +-----------|-----     |   |-- appel de Y1 (n1)
                       |           |----------|   |
                       |           |    AR    |   |
                       |           |==========|
                       |           |    B     |   |
                       |           |----------|   |
                       | +---------|-----     |   |-- appel de Y2 (n1)
                       | |         |----------|   |
                       +-|-------->|    AR    |   |
                         |         |==========|
c'est ce B la qui ---->  |         |    B     |   |
doit etre accede         |         |----------|   |
                         | +-------|-----     |   |-- appel de Y (n0)
                         | |       |----------|   |
                         +-|------>|    AR    |   |
                           |       |==========|
                           |       |    A     |   |-- globale (n0)
                           +------>+----------+
```

FIG. 11.2 – *État de la pile lors de l'exécution de* `C := B + A` *dans le programme de la figure 11.1.*

bien au nombre de liens statiques qu'il faut parcourir (en descendant dans la pile) pour accéder à la zone voulue.

On introduit donc une modification de l'instruction P-Code `LDI` :

```
LDL  i  n
```

Les paramètres sont `i`, l'adresse relative au début de la zone, et `n`, le nombre de liens statiques à parcourir pour accéder à cette zone. Remarquons que ces deux paramètres sont calculables à la compilation.

L'ancienne version de `LDL i` est équivalente à `LDL i 0`.

On reprend l'exemple de la figure 11.1. La pile de la figure 11.3 indique les liens de chaînage dynamique et les liens de chaînage statique sur la pile. Le P-Code généré est donné à la figure 11.4.

11.3 Travaux dirigés et travaux pratiques

1. Modifier l'interprétation de l'instruction `LDL` pour prendre en compte les deux paramètres. (Cela peut amener des changements dans le format des sources P-Code, les instructions acceptaient auparavant au plus un paramètre).
2. Intégrer la gestion du niveau dans les procédures de génération de code. On introduit une variable globale `NIVEAU` qui contient le numéro du niveau courant. Cette variable `NIVEAU` est utilisée lors de l'entrée dans la table des symboles.
3. Modifier les procédures de génération de l'accès à une variable (c'est-à-dire `AFFEC`, `LIRE`, et `FACT`) en fonction du niveau courant et du niveau de la variable.

```
                              |           |
                              |-----------|
                              |    C      |
                              |-----------|
                              |     ------|----------+
                              |-----------|          |
                 +------------|-----      |          |
                 |            |-----------|          |
                 |            |    AR     |          |
                 |            |===========|          |
                 |            |    B      |          |
                 |            |-----------|          |
                 |            |     ------|----------+
                 |            |-----------|          |
                 | +----------|-----      |          |
chainage         | |          |-----------|          |     chainage
dynamique        +-|--------->|    AR     |          |     statique
                   |          |===========|          |
                   |          |    B      |          |
                   |          |-----------|          |
                   |          |     ------|------+   |
                   |          |-----------|      |   |
                   | +--------|-----      |      |   |
                   | |        |-----------|      |   |
                   +-|------->|    AR     |<-----|---+
                     |        |===========|      |
                     |        |    A      |      |
                     +------->+-----------+<-----+
```

FIG. 11.3 – *Liens de chaînage statique et dynamique dans la pile.*


```
0   BRN 17
1   BRN 14
2   INT 1
3   LDL 0 0    accès à C de Y1
4   LDL 0 1    accès à B de Y
5   LDV
6   LDA 0      accès à A de X
7   LDV
8   ADD
9   STO
10  RET
11  INT 1
12  CAL 2
13  RET
14  INT 1
15  CAL 11
16  RET
17  INT 1
18  CAL 1
19  HLT
```

FIG. 11.4 – *P-Code généré pour le programme de la figure 11.1.*

Chapitre 12

Procédures paramétrées

On introduit donc ici les procédures paramétrées; l'extension de PP6 ainsi construite se nomme PP8. (PP8 est une extension de PP6, on ne reprend donc pas les procédures imbriquées.)

12.1 Grammaire et sémantique

12.1.1 Définition de procédures

Les règles suivantes de la grammaire de PP6 sont modifiées

```
PROGRAM   ::= program ID ; BLOCK .
BLOCK     ::= CONSTS VARS PROCS INSTS
PROCS     ::= { proc ID PARAMLIST ; CONSTS VARS INSTS ; }
PARAMLIST ::= ( PARAMDECL { , PARAMDECL } ) | ε
PARAMDECL ::= var ID | ID
```

La règle `PARAMLIST` définit la liste des paramètres formels de la procédure.

12.1.2 Appel de procédure

L'appel de procédure peut maintenant comporter une liste de paramètres effectifs :

```
APPEL        ::= ID PARAMEFFLIST
PARAMEFFLIST ::= ( PARAMEFF { , PARAMEFF } ) | ε
PARAMEFF     ::= EXPR
```

Sémantiquement, le nombre de paramètres effectifs de l'appel de procédure doit correspondre au nombre de paramètres déclarés dans la définition de la procédure. Il est possible au compilateur de vérifier le respect de cette règle sémantique en associant le nombre de paramètres de la procédure à son entrée dans la table des symboles.

12.1.3 Passages de paramètre par valeur et par adresse

Contrairement à d'autres langages, deux types de passage de paramètres existent en Pascal. En Fortran, tous les passages de paramètres se font par adresse; en C tous les passages de paramètres se font par valeur[1].

1. Cette limitation apparente est contournée par la possibilité de passer par valeur des adresses; le passage par adresse est donc à la charge du programmeur.


```
                |----------|
                |          |
                |          |<--- zone allouee pour
                |          |     l'appel (AR
                |          |     + liens statiques
BASE ---------> |          |     + variables locales)
                |----------|
                |          |<--- parametres effectifs
                |          |     de l'appel
                |----------|
                |          |
                |          |
                +----------+
```

FIG. 12.1 – *État de la pile juste après l'appel d'une procédure.*

Afin de différencier les deux types de passages de paramètres, les définitions de paramètres formels `PARAMDECL` peuvent être étiquetées par le mot clé `var` pour indiquer un passage par adresse; dans le cas contraire , il s'agit d'un passage par valeur.

Le paramètre effectif correspondant à une définition de paramètre formel déclaré `var` est une expression `EXPR` limitée à un identificateur `ID` (cf. figure 2.2, page 28).

Un paramètre effectif correspondant à un passage par valeur est une expression `EXPR`. Cette expression est évaluée et sa valeur est passée à la procédure.

12.2 Implémentation du passage de paramètre par valeur

12.2.1 Allocation mémoire des paramètres

Les paramètres d'une procédure ne peuvent être connus avant l'exécution. De plus, ils sont différents à chaque appel. L'idée de base est qu'une procédure retrouve ses paramètres sur la pile *sous* la zone qui lui est allouée, cf figure 12.1

Ainsi, les paramètres seront accédés grâce à un déplacement *négatif* relatif à `BASE`.

12.2.2 Génération de code

Le code à générer pour un appel de procédure doit inclure

1. Le P-Code empilant les valeurs des paramètres effectifs.
2. Une instruction P-Code `CAL i` réalisant le branchement nécessaire à l'appel de la procédure.

On modifie l'instruction P-Code `RET` de retour de procédure qui accepte maintenant un paramètre :

```
RET  n
```

Ce paramètre correspond au nombre de paramètres devant être dépilés pour mettre à jour la variable `SP` du compilateur. On modifie donc l'interpréteur P-Code (section 9.2, page 79) en conséquence :

```
with INST do
```

```
case MNE of
   ...
   RET : begin
      PC   := MEM [BASE] ;
      SP   := SP - SUITE ;
      BASE := MEM [BASE + 1]
   end ;
end
```

Prenons en exemple le code PP8 suivant :

```
program X ;
var A ;
proc Y (N) ;
   begin
      write (N)
   end ;
begin
   read (A) ;
   Y (A+2)
end .
```

On générera le P-Code suivant :

```
0   INT 1    réserve 1 emplacement pour A
1   BRN 6    saut au début du programme principal
#            début de la fonction Y
2   LDL -3   empile l'adresse de N
3   LDV      accède la valeur de N
4   PRN
5   RET 1
#            début du programme
6   LDA 0    adresse de A
7   INN      lecture d'une valeur dans A
8   LDA 0    adresse de A
9   LDV      valeur de A
10  LDI 2    2
11  ADD      addition : la valeur est sur la pile
12  CAL 2    appel de Y
13  HLT
```

La pile avant l'exécution de l'instruction `LDL -3` de la ligne 2 est schématisée à la figure 12.2.

Le paramètre de l'instruction `LDL` a pour valeur -3 car l'instruction `LDL i` référence l'emplacement mémoire `BASE+2-i` (cf. exercice 5, page 88).

12.3 Implémentation du passage de paramètre par adresse

Le mécanisme mis en place pour accéder les paramètres passés par adresse est semblable à celui mis en place pour les passages par valeur.

L'appelant laisse sur la pile non pas la valeur du paramètre effectif, mais son adresse.


```
|          |<-------- variables locales
|          |
|----------|
|     -----|---+
|----------|   |
|    AR    |<-------- BASE
|----------|   |
|    N     |   |
|----------|   V
|          |
```

Fig. 12.2 – *État de la pile juste avant l'exécution de* `LDL -3`.

L'accès à un paramètre formel se fait de la manière suivante :

- Pour accéder à l'adresse du paramètre effectif, on génère le P-Code

```
LDL -n
LDV
```

- Pour accéder à sa valeur, on génère :

```
LDL -n
LDV
LDV
```

12.4 Travaux dirigés et travaux pratiques

1. Quels sont les avantages de proposer les deux types de passage de paramètre par adresse et par valeur au sein d'un même langage. Comparer avec les approches de Fortran (passage par adresse uniquement) et C (passage par valeur uniquement).
2. Qu'advient-il si, dans un code Pascal, on passe une expression non réduite à un nom de variable comme paramètre effectif dont le paramètre correspondant est déclaré `var` ? Comment un compilateur Pascal peut-il détecter une telle erreur ?
3. Implanter dans un premier temps le passage de paramètre par valeur. Étendre cette implantation au passage par adresse.
4. Quels problèmes nouveaux surviennent si l'on désire étendre le passage de paramètre à des variables et valeurs de type différents (entiers et réels, puis tableaux, structures et types construits) ?

Chapitre 13

Fonctions

On introduit donc ici les fonctions ; l'extension de PP8 ainsi construite se nomme PP9.

13.1 Grammaire et sémantique

La grammaire est modifiée pour accepter les définitions de fonctions. Ces définitions sont distinguées de celles des procédures par l'introduction du mot clé `func` :

```
PROCS    ::= { PROCTYPE ID PARAMLIST ; CONSTS VARS INSTS ; }
PROCTYPE ::= proc | func
```

Les fonctions retournent toutes une valeur entière. Cette valeur est celle qui est « affectée » à l'identificateur de la fonction au sein de celle-ci.

L'appel de fonction se fait par son nom, éventuellement suivi de ses paramètres. On modifie la règle `FACT` pour ce faire :

```
FACT ::= ID | NUM | ( EXPR ) | APPEL
```

Dans cette règle, un `ID` désigne le nom d'une variable ; les appels de fonctions se font par dérivation du non-terminal `APPEL`.

13.2 Génération de code

Un appel de fonction laisse sur la pile la valeur retournée par la fonction. Il est donc nécessaire que l'appelant réserve un emplacement sur la pile pour le retour de cette valeur par une instruction `INT 1` avant d'empiler les paramètres de la fonction et de faire l'appel.

La pile à l'entrée d'une fonction est bâtie sur le modèle de la figure 13.1

S'il y a N paramètres, le résultat de la fonction est accédé (dans la fonction) par

`LDL -`N`-2-1`

Cette adresse est donc celle dans laquelle il faut ranger le résultat de l'affectation de l'identificateur correspondant au nom de la fonction.

13.3 Exemple

Le P-Code du programme de la figure 13.2 est donné à la figure 13.3.


```
         |----------|
         |          |
         |          |<--- variables locales
         |----------|
+--------|------    |
|        |----------|
|        |    AR    |<--- BASE
|        |----------|
|        |          |<--- parametres effectifs
|        |          |     de l'appel
V        |----------|
         |          |<--- retour
         |----------|
         |          |
```

Fig. 13.1 – *État de la pile juste après l'appel d'une fonction.*

```
program X ;
var N ;
   func F (M) ;
   begin
      if M = 1 then F := 1 ;
      if M > 1 then F := M * F (M-1)
   end ;
begin
   read (N) ;
   write (F(N))
end .
```

Fig. 13.2 – *Exemple de programme PP9.*

```
0   INT 1     reserve 1 emplacement pour N
1   BRN 27
#             debut de la fonction F
2   LDL -3    |
3   LDV       | M:=1
4   LDI 1     |
5   EQL       |
6   BZE 10

7   LDL -4    |
8   LDI 1     | F:=1
9   STO       |

10  LDL -3    |
11  LDV       | M>1
12  LDI 1     |
13  GTR       |
14  BZE 26

15  LDL -4    | F

16  LDL -3    | M
17  LDV       |

18  INT 1     |
19  LDL -3    |
20  LDV       | F (M-1)
21  LDI 1     |
22  SUB       |
23  CAL 2     |

24  MUL       | *

25  STO       | :=

26  RET 1

#             programme X
27  LDA 0
28  INN
29  INT 1     |
30  LDA 0     | F (N)
31  LDV       |
32  CAL 1     |

33  PRN
34  HLT
```

Fig. 13.3 – *P-Code généré pour le programme de la figure 13.2.*

13.4 Travaux dirigés et travaux pratiques

1. En Pascal, la valeur retournée par une fonction est celle « affectée » au nom de la fonction lors de la fin de celle-ci. Comparer ce choix avec celui de C qui propose une instruction spécifique (`return`) : du point de vue du programmeur, et du point de vue de l'écriture du compilateur.
2. Implanter les fonctions telles que décrites dans ce chapitre.
3. Proposer une syntaxe (ou reprendre celle de Pascal !) pour étendre les fonctions à différents types. Quels problèmes posent l'extension à des types construits (tableaux, structures...) ?

Quatrième partie

Édition de liens

Chapitre 14

Compilation séparée et édition de liens

Cette nouvelle version de PP, PP10, introduit les notions de *module* et de *compilation séparée* pour lesquelles la mise en place d'une *édition de liens* est nécessaire.

On se base ici sur la version 5 de PP (chapitre 9, les procédures simples, sans paramètres, sans déclarations locales, et non imbriquées).

14.1 L'exemple Unix

Sous Unix, le compilateur `cc` est composé de plusieurs phases :

`cpp` Le préprocesseur `cpp` est chargé de la gestion des fichiers inclus et des macros. `cpp` inclus « physiquement » les fichiers `#include` et remplace les macros `#define` par leur définition. Il passe à `cc` le source C ainsi produit.

`ccom` C'est le cœur du compilateur C. Il produit un fichier code objet (`.o`) à partir d'un fichier C.

`ld` L'éditeur de liens. Il assure la production d'un fichier exécutable à partir d'une liste de fichiers `.o` et des fichiers de bibliothèque (`.a`).

L'intérêt de cette approche est la possibilité de rassembler en un programme exécutable des morceaux (modules) de programmes compilés séparément et la possibilité de créer des exécutables à partir de modules compilés dont on n'a pas les sources ; c'est la notion de bibliothèque.

14.2 Grammaire et sémantique

Un programme PP10 est la réunion d'un ensemble de modules et d'un programme principal.

Un module ou un programme principal est contenu dans un fichier ; on distingue donc le fichier programme principal et les fichiers modules. Chacun des fichiers composant un programme est compilé séparément ; l'ensemble des fichiers objets produits lors de cette phase de compilation est passé à un éditeur de liens qui produit un programme P-Code complet.

14.2.1 Programme principal

La grammaire d'un fichier programme est une évolution de la grammaire de PP5.

```
PROGRAM  ::= program ID ; IMPORT EXPORT BLOCK .
IMPORT   ::= { import from ID_MODULE : ID { , ID } ; }
EXPORT   ::= export : ID { , ID } ; | ε
```

Les `ID` d'une déclaration `import` sont des identificateurs de procédures ou de variables définies dans un fichier module de nom `ID_MODULE`. Ces variables et procédures doivent avoir été exportées par le module `ID_MODULE` (cf. infra). Ces variables et procédures sont alors visibles (au niveau zéro) dans le programme principal : les variables peuvent être référencées en lecture et/ou en écriture ; les procédures peuvent être appelées.

Les `ID` d'une liste `export` sont des noms de procédures ou de variables globales définies au niveau zéro du programme principal. Ces procédures et ces variables pourront être accédées par des modules externes.

14.2.2 Modules

Un fichier module est défini comme un fichier programme principal ne comportant pas de partie `BLOCK` :

```
MODULE ::= module ID ; IMPORT EXPORT CONSTS VARS PROCS
IMPORT ::= { import from ID_MODULE : ID { , ID } ; }
EXPORT ::= export : ID { , ID } ; | ε
```

Les `ID` d'une déclaration `import` sont des identificateurs de procédures ou de variables définies dans un autre fichier module de nom `ID_MODULE` ou dans le fichier programme principal de nom `ID_MODULE`. Ces variables et procédures doivent avoir été déclarées dans une section `export`.

Les `ID` d'une liste `export` sont des noms de procédures ou de variables globales définies au niveau zéro du module. Ces procédures et ces variables pourront être accédées par des modules externes qui les importeront explicitement.

14.3 Forme du code généré

14.3.1 Instructions incomplètes

Lors de la compilation d'un module ou d'un programme principal, les adresses des variables et procédures externes ne sont pas connues.

L'idée est alors de générer des instructions incomplètes pour les appels de ces procédures (`CAL`) et les références à ces variables (`LDA`). Ces instructions incomplètes seront complétées par l'éditeur de liens. Pour ce faire, il est nécessaire d'informer l'éditeur de liens des instructions incomplètes et des noms des procédures/variables avec lesquelles il est nécessaire de compléter ces instructions. Une partie d'un fichier de code sera donc constituée d'une liste de triplet

module, *identificateur*, *instruction*

Un *module* est le nom d'un module qui doit exporter l'identificateur *identificateur*. L'adresse de cet identificateur permet de compléter l'instruction numéro *instruction* du code.

De manière symétrique, le code généré depuis un module ou un programme exportant des procédures/variables doit inclure une liste de couples

identificateur, *adresse*

donnant l'adresse de l'identificateur *identificateur* défini dans le module.

14.3.2 Adresses translatables

Un nouveau problème intervient à ce niveau : à quelle adresse mémoire faut-il allouer les variables globales définies dans un module ? Imaginons que l'on veuille allouer ces variables au dessus des variables globales du programme principal. Lors de la compilation d'un module, on ne peut déterminer l'adresse de ces variables car on ne connaît pas la taille des données globales du programme principal.

En fait, on va allouer ces données dans une zone de mémoire virtuelle dite translatable. L'ensemble des adresses dans cette mémoire virtuelle qui sont allouées et utilisées dans un module seront *translatées* lors de l'édition de liens vers des adresses dans la mémoire « réelle » de la machine P-Code.

Une information supplémentaire doit être présente dans le code généré par un programme principal ou un module : c'est la taille de la zone mémoire nécessaire aux variables globales du programme principal ou du module. à partir de ces tailles, l'éditeur de liens générera l'instruction `INT` nécessaire à la réservation de la mémoire pour toutes les variables globales.

14.3.3 Code translatable

Un problème similaire à celui de la mémoire se retrouve au niveau du code. Quel est le numéro d'une instruction donnée généré depuis un module ?

De manière symétrique, on introduit la notion de code translatable. Les numéros d'instruction référencés dans un module sont des numéros relatifs au début du code du module et seront translatés par l'éditeur de liens en fonction de la place de ce code dans le code final.

14.4 L'éditeur de liens

L'éditeur de liens est donc chargé de produire un code P-Code unique à partir de multiples codes. Il lui est nécessaire de

1. Construire la zone mémoire globale comme la réunion des zones mémoires globales des différents modules et du programme principal. Un déplacement dans cette zone est alors attaché à chaque module ; c'est la valeur qu'il sera nécessaire d'ajouter aux références des variables globales de ce module pour obtenir leur adresse « réelle ».
2. Construire un segment de code unique par réunion des segments de code des différents modules et du programme principal. Un déplacement dans cette zone est attaché à chaque module ; c'est la valeur de ce déplacement qu'il faut ajouter à chaque numéro d'instruction du module pour obtenir son adresse dans le segment de code produit.
3. Générer une instruction `INT` de réservation de la zone mémoire globale.
4. Compléter chacune des instructions incomplètes (numéro d'instruction ou adresse externe).

14.5 Travaux dirigés et travaux pratiques

1. Définir précisément le format des fichiers de code produits par le compilateur et qui seront acceptés en entrée de l'éditeur de liens.

2. Mettre en place les modifications nécessaires au compilateur pour générer cette information. Une structure de données interne est-elle nécessaire pour garder cette information lors de la compilation, ou est-il possible de produire directement l'information dans un fichier au fur et à mesure ?
3. Dans ce chapitre, nous n'avons considéré que les variables simples entières et les procédures simples. Le type des variables et procédures externes sont donc connues « par défaut ». Dans une version plus complète, il est nécessaire d'identifier les types des variables et des paramètres des procédures.

   Proposer une syntaxe pour cette déclaration. On pourra se rapprocher de la syntaxe des déclarations `forward` de Pascal.
4. Quelles erreurs peuvent être détectées au niveau de l'éditeur de liens ?

Cinquième partie

Annexes

Annexe A

Syntaxe du langage Pascal

Cette annexe donne la syntaxe complète du langage Pascal sous forme d'un fichier `yacc` (analyseur syntaxique) et d'un fichier `lex` (analyseur lexical).

A.1 Grammaire yacc

La grammaire de Pascal est donnée sous forme d'un fichier `yacc`. `yacc` est une commande UNIX produisant le source C d'un analyseur syntaxique à partir de la description d'une grammaire. Dans ses grandes lignes, la forme de la grammaire donnée en entrée ressemble à celle définie dans ce document. Le développement d'un compilateur à l'aide de `yacc` est réalisé par l'insertion de codes C appelés « actions » dans les règles de la grammaire. La grammaire présentée ici est dépourvue de telles actions.

```
%{
/*
 * grammar.y
 *
 * Pascal grammar in Yacc format, based originally on BNF given
 * in "Standard Pascal -- User Reference Manual", by Doug Cooper.
 * This in turn is the BNF given by the ANSI and ISO Pascal standards,
 * and so, is PUBLIC DOMAIN. The grammar is for ISO Level 0 Pascal.
 * The grammar has been massaged somewhat to make it LALR, and added
 * the following extensions.                                  at9.0pt
 *
 * constant expressions
 * otherwise statement in a case
 * productions to correctly match else's with if's
 * beginnings of a separate compilation facility
 */

%}

%token AND ARRAY ASSIGNMENT CASE CHARACTER_STRING
%token COLON COMMA CONST DIGSEQ DIV DO DOT DOTDOT
```



```
%token DOWNTO ELSE END EQUAL EXTERNAL FOR FORWARD
%token FUNCTION GE GOTO GT IDENTIFIER IF IN LABEL LBRAC
%token LE LPAREN LT MINUS MOD NIL NOT NOTEQUAL OF OR
%token OTHERWISE PACKED PBEGIN PFILE PLUS PROCEDURE
%token PROGRAM RBRAC REALNUMBER RECORD REPEAT RPAREN
%token SEMICOLON SET SLASH STAR STARSTAR THEN
%token TO TYPE UNTIL UPARROW VAR WHILE WITH

%%
file : program
        | module
        ;

program : program_heading semicolon block DOT
        ;

program_heading : PROGRAM identifier
        | PROGRAM identifier LPAREN identifier_list RPAREN
        ;

identifier_list : identifier_list comma identifier
        | identifier
        ;

block : label_declaration_part
        constant_definition_part
        type_definition_part
        variable_declaration_part
        procedure_and_function_declaration_part
        statement_part
        ;

module : constant_definition_part
        type_definition_part
        variable_declaration_part
        procedure_and_function_declaration_part
        ;

label_declaration_part : LABEL label_list semicolon
        |
        ;

label_list : label_list comma label
        | label
        ;

label : DIGSEQ
```

```
        ;

constant_definition_part : CONST constant_list
        |
        ;

constant_list : constant_list constant_definition
        | constant_definition
        ;

constant_definition : identifier EQUAL cexpression semicolon
        ;

/*constant : cexpression ;                /* good stuff! */

cexpression : csimple_expression
        | csimple_expression relop csimple_expression
        ;

csimple_expression : cterm
        | csimple_expression addop cterm
        ;

cterm : cfactor
        | cterm mulop cfactor
        ;

cfactor : sign cfactor
        | cexponentiation
        ;

cexponentiation : cprimary
        | cprimary STARSTAR cexponentiation
        ;

cprimary : identifier
        | LPAREN cexpression RPAREN
        | unsigned_constant
        | NOT cprimary
        ;

constant : non_string
        | sign non_string
        | CHARACTER_STRING
        ;

sign : PLUS
```



```
        | MINUS
        ;

non_string : DIGSEQ
        | identifier
        | REALNUMBER
        ;

type_definition_part : TYPE type_definition_list
        |
        ;

type_definition_list : type_definition_list type_definition
        | type_definition
        ;

type_definition : identifier EQUAL type_denoter semicolon
        ;

type_denoter : identifier
        | new_type
        ;

new_type : new_ordinal_type
        | new_structured_type
        | new_pointer_type
        ;

new_ordinal_type : enumerated_type
        | subrange_type
        ;

enumerated_type : LPAREN identifier_list RPAREN
        ;

subrange_type : constant DOTDOT constant
        ;

new_structured_type : structured_type
        | PACKED structured_type
        ;

structured_type : array_type
        | record_type
        | set_type
        | file_type
        ;
```

```
array_type : ARRAY LBRAC index_list RBRAC OF component_type
        ;

index_list : index_list comma index_type
        | index_type
        ;

index_type : ordinal_type ;

ordinal_type : new_ordinal_type
        | identifier
        ;

component_type : type_denoter ;

record_type : RECORD record_section_list END
        | RECORD record_section_list semicolon variant_part END
        | RECORD variant_part END
        ;

record_section_list : record_section_list semicolon record_section
        | record_section
        ;

record_section : identifier_list COLON type_denoter
        ;

variant_part : CASE variant_selector OF variant_list semicolon
        | CASE variant_selector OF variant_list
        |
        ;

variant_selector : tag_field COLON tag_type
        | tag_type
        ;

variant_list : variant_list semicolon variant
        | variant
        ;

variant : case_constant_list COLON LPAREN record_section_list RPAREN
        | case_constant_list COLON LPAREN record_section_list semicolon
                variant_part RPAREN
        | case_constant_list COLON LPAREN variant_part RPAREN
        ;
```



```
case_constant_list : case_constant_list comma case_constant
        | case_constant
        ;

case_constant : constant
        | constant DOTDOT constant
        ;

tag_field : identifier ;

tag_type : identifier ;

set_type : SET OF base_type
        ;

base_type : ordinal_type ;

file_type : PFILE OF component_type
        ;

new_pointer_type : UPARROW domain_type
        ;

domain_type : identifier ;

variable_declaration_part : VAR variable_declaration_list semicolon
        |
        ;

variable_declaration_list :
          variable_declaration_list semicolon variable_declaration
        | variable_declaration
        ;

variable_declaration : identifier_list COLON type_denoter
        ;

procedure_and_function_declaration_part :
                proc_or_func_declaration_list semicolon
        |
        ;

proc_or_func_declaration_list :
          proc_or_func_declaration_list semicolon proc_or_func_declaration
        | proc_or_func_declaration
        ;
```

```
proc_or_func_declaration : procedure_declaration
        | function_declaration
        ;

procedure_declaration : procedure_heading semicolon directive
        | procedure_heading semicolon procedure_block
        ;

procedure_heading : procedure_identification
        | procedure_identification formal_parameter_list
        ;

directive : FORWARD
        | EXTERNAL
        ;

formal_parameter_list : LPAREN formal_parameter_section_list RPAREN ;

formal_parameter_section_list : formal_parameter_section_list semicolon formal_parameter_section
        | formal_parameter_section
        ;

formal_parameter_section : value_parameter_specification
        | variable_parameter_specification
        | procedural_parameter_specification
        | functional_parameter_specification
        ;

value_parameter_specification : identifier_list COLON identifier
        ;

variable_parameter_specification : VAR identifier_list COLON identifier
        ;

procedural_parameter_specification : procedure_heading ;

functional_parameter_specification : function_heading ;

procedure_identification : PROCEDURE identifier ;

procedure_block : block ;

function_declaration : function_heading semicolon directive
        | function_identification semicolon function_block
        | function_heading semicolon function_block
        ;
```



```
function_heading : FUNCTION identifier COLON result_type
        | FUNCTION identifier formal_parameter_list COLON result_type
        ;

result_type : identifier ;

function_identification : FUNCTION identifier ;

function_block : block ;

statement_part : compound_statement ;

compound_statement : PBEGIN statement_sequence END ;

statement_sequence : statement_sequence semicolon statement
        | statement
        ;

statement : open_statement
        | closed_statement
        ;

open_statement : label COLON non_labeled_open_statement
        | non_labeled_open_statement
        ;

closed_statement : label COLON non_labeled_closed_statement
        | non_labeled_closed_statement
        ;

non_labeled_closed_statement : assignment_statement
        | procedure_statement
        | goto_statement
        | compound_statement
        | case_statement
        | repeat_statement
        | closed_with_statement
        | closed_if_statement
        | closed_while_statement
        | closed_for_statement
        |
        ;

non_labeled_open_statement : open_with_statement
        | open_if_statement
        | open_while_statement
        | open_for_statement
```

```
        ;

repeat_statement : REPEAT statement_sequence UNTIL boolean_expression
        ;

open_while_statement : WHILE boolean_expression DO open_statement
        ;

closed_while_statement : WHILE boolean_expression DO closed_statement
        ;

open_for_statement : FOR control_variable ASSIGNMENT initial_value direction
                        final_value DO open_statement
        ;

closed_for_statement : FOR control_variable ASSIGNMENT initial_value direction
                        final_value DO closed_statement
        ;

open_with_statement : WITH record_variable_list DO open_statement
        ;

closed_with_statement : WITH record_variable_list DO closed_statement
        ;

open_if_statement : IF boolean_expression THEN statement
        | IF boolean_expression THEN closed_statement ELSE open_statement
        ;

closed_if_statement : IF boolean_expression THEN closed_statement
                        ELSE closed_statement
        ;

assignment_statement : variable_access ASSIGNMENT expression
        ;

variable_access : identifier
        | indexed_variable
        | field_designator
        | variable_access UPARROW
        ;

indexed_variable : variable_access LBRAC index_expression_list RBRAC
        ;

index_expression_list : index_expression_list comma index_expression
        | index_expression
```



```
        ;

index_expression : expression ;

field_designator : variable_access DOT identifier
        ;

procedure_statement : identifier params
        | identifier
        ;

params : LPAREN actual_parameter_list RPAREN ;

actual_parameter_list : actual_parameter_list comma actual_parameter
        | actual_parameter
        ;

/*
 * this forces you to check all this to be sure that only write and
 * writeln use the 2nd and 3rd forms, you really can't do it easily in
 * the grammar, especially since write and writeln aren't reserved
 */
actual_parameter : expression
        | expression COLON expression
        | expression COLON expression COLON expression
        ;

goto_statement : GOTO label
        ;

case_statement : CASE case_index OF case_list_element_list END
        | CASE case_index OF case_list_element_list SEMICOLON END
        | CASE case_index OF case_list_element_list semicolon
                        otherwisepart statement END
        | CASE case_index OF case_list_element_list semicolon
                        otherwisepart statement SEMICOLON END
        ;

case_index : expression ;

case_list_element_list : case_list_element_list semicolon case_list_element
        | case_list_element
        ;

case_list_element : case_constant_list COLON statement
        ;
```

```
otherwisepart :   OTHERWISE
        | OTHERWISE COLON
        ;

control_variable : identifier ;

initial_value : expression ;

direction : TO
        | DOWNTO
        ;

final_value : expression ;

record_variable_list : record_variable_list comma variable_access
        | variable_access
        ;

boolean_expression : expression ;

expression : simple_expression
        | simple_expression relop simple_expression
        ;

simple_expression : term
        | simple_expression addop term
        ;

term : factor
        | term mulop factor
        ;

factor : sign factor
        | exponentiation
        ;

exponentiation : primary
        | primary STARSTAR exponentiation
        ;

primary : variable_access
        | unsigned_constant
        | function_designator
        | set_constructor
        | LPAREN expression RPAREN
        | NOT primary
        ;
```



```
unsigned_constant : unsigned_number
        | CHARACTER_STRING
        | NIL
        ;

unsigned_number : unsigned_integer | unsigned_real ;

unsigned_integer : DIGSEQ
        ;

unsigned_real : REALNUMBER
        ;

/* functions with no params will be handled by plain identifier */
function_designator : identifier params
        ;

set_constructor : LBRAC member_designator_list RBRAC
        | LBRAC RBRAC
        ;

member_designator_list : member_designator_list comma member_designator
        | member_designator
        ;

member_designator : member_designator DOTDOT expression
        | expression
        ;

addop: PLUS
        | MINUS
        | OR
        ;

mulop : STAR
        | SLASH
        | DIV
        | MOD
        | AND
        ;

relop : EQUAL
        | NOTEQUAL
        | LT
        | GT
        | LE
```

```
          | GE
          | IN
          ;

identifier : IDENTIFIER
          ;

semicolon : SEMICOLON
          ;

comma : COMMA
          ;

%%

#include <stdio.h>

extern int line_no;
extern char yytext[];

yyerror(s)
char *s;
{
          fprintf(stderr, "%s:  at or before '%s', line %d\n", s, yytext, line_no);
}
```


A.2 Syntaxe `lex`

La description lexicale de Pascal est ici donnée sous forme d'un fichier source `lex`. `lex` est un outil UNIX permettant de construire un analyseur lexical. Il est habituellement utilisé en complément à `yacc` auquel il retourne des tokens (instructions `return`).

```
%{
/*
 * scan.l
 *
 * lex input file for pascal scanner
 *
 * extensions: to ways to spell "external" and "->" ok for "^".
 */

#include <stdio.h>
#include "y.tab.h"
int line_no = 1;

%}

A         [aA]
B         [bB]
C         [cC]
D         [dD]
E         [eE]
F         [fF]
G         [gG]
H         [hH]
I         [iI]
J         [jJ]
K         [kK]
L         [lL]
M         [mM]
N         [nN]
O         [oO]
P         [pP]
Q         [qQ]
R         [rR]
S         [sS]
T         [tT]
U         [uU]
V         [vV]
W         [wW]
X         [xX]
Y         [yY]
Z         [zZ]
```

```
NQUOTE [^']

%%

{A}{N}{D}                       return(AND);
{A}{R}{R}{A}{Y}                 return(ARRAY);
{C}{A}{S}{E}                    return(CASE);
{C}{O}{N}{S}{T}                 return(CONST);
{D}{I}{V}                       return(DIV);
{D}{O}                          return(DO);
{D}{O}{W}{N}{T}{O}              return(DOWNTO);
{E}{L}{S}{E}                    return(ELSE);
{E}{N}{D}                       return(END);
{E}{X}{T}{E}{R}{N}     |
{E}{X}{T}{E}{R}{N}{A}{L}        return(EXTERNAL);
{F}{O}{R}                       return(FOR);
{F}{O}{R}{W}{A}{R}{D}           return(FORWARD);
{F}{U}{N}{C}{T}{I}{O}{N}        return(FUNCTION);
{G}{O}{T}{O}                    return(GOTO);
{I}{F}                          return(IF);
{I}{N}                          return(IN);
{L}{A}{B}{E}{L}                 return(LABEL);
{M}{O}{D}                       return(MOD);
{N}{I}{L}                       return(NIL);
{N}{O}{T}                       return(NOT);
{O}{F}                          return(OF);
{O}{R}                          return(OR);
{O}{T}{H}{E}{R}{W}{I}{S}{E}     return(OTHERWISE);
{P}{A}{C}{K}{E}{D}              return(PACKED);
{B}{E}{G}{I}{N}                 return(PBEGIN);
{F}{I}{L}{E}                    return(PFILE);
{P}{R}{O}{C}{E}{D}{U}{R}{E}     return(PROCEDURE);
{P}{R}{O}{G}{R}{A}{M}           return(PROGRAM);
{R}{E}{C}{O}{R}{D}              return(RECORD);
{R}{E}{P}{E}{A}{T}              return(REPEAT);
{S}{E}{T}                       return(SET);
{T}{H}{E}{N}                    return(THEN);
{T}{O}                          return(TO);
{T}{Y}{P}{E}                    return(TYPE);
{U}{N}{T}{I}{L}                 return(UNTIL);
{V}{A}{R}                       return(VAR);
{W}{H}{I}{L}{E}                 return(WHILE);
{W}{I}{T}{H}                    return(WITH);
[a-zA-Z]([a-zA-Z0-9])+          return(IDENTIFIER);

":="                            return(ASSIGNMENT);
'({NQUOTE}|'')+'                return(CHARACTER_STRING);
```



```
":"                          return(COLON);
","                          return(COMMA);
[0-9]+                       return(DIGSEQ);
"."                          return(DOT);
".."                         return(DOTDOT);
"="                          return(EQUAL);
">="                         return(GE);
">"                          return(GT);
"["                          return(LBRAC);
"<="                         return(LE);
"("                          return(LPAREN);
"<"                          return(LT);
"-"                          return(MINUS);
"<>"                         return(NOTEQUAL);
"+"                          return(PLUS);
"]"                          return(RBRAC);
[0-9]+"."[0-9]+              return(REALNUMBER);
")"                          return(RPAREN);
";"                          return(SEMICOLON);
"/"                          return(SLASH);
"*"                          return(STAR);
"**"                         return(STARSTAR);
"->"                     |
"^"                          return(UPARROW);

"(*"                     |
"{"                          { register int c;
                                 while ((c = input()))
                                 {
                                     if (c == '}')
                                         break;
                                     else if (c == '*')
                                     {
                                         if ((c = input()) == ')')
                                             break;
                                         else
                                             unput (c);
                                     }
                                     else if (c == '\n')
                                         line_no++;
                                     else if (c == 0)
                                         commenteof();
                                 }
                             }

[ \t\f]                      ;
```

```
\n                              line_no++;

.                               { fprintf (stderr,
                                "'%c' (0%o):  illegal charcter at line %d\n",
                                        yytext[0], yytext[0], line_no);
                                }

%%

commenteof()
{
        fprintf (stderr, "unexpected EOF inside comment at line %d\n",
                line_no);
        exit (1);
}

yywrap ()
{
        return (1);
}
```

Annexe B

Éléments de solutions des exercices

B.1 Procédures (chapitre 9)

Procédures PROCS de génération de code pour les procédures.

```
procedure PROCS ;
begin
   while TOKEN = PROC_TOKEN do begin
      NEXT_TOKEN ;
      TEST_ET_ENTRE (ID_TOKEN, PROCEDURE) ;
      TESTE (PT_VIRG_TOKEN) ;
      INSTS ;
      TESTE (PT_VIRG_TOKEN) ;
      GENERER1 (RET) ;
   end
end ;
```

La procédure TEST_ET_ENTRE est modifiée pour insérer le numéro de la prochaine instruction P-Code à générer dans le champ ADRESSE de la table des symboles.

```
procedure TESTE_ET_ENTRE (T : TOKENS ; C : CLASSES );
var PCODE_INST
begin
   ...
   CASE C in
      ...
      PROCEDURE : begin
         NEXT_INST (PCODE_INST) ;
         with TABLEINDEX [DERNIERSYM] do begin
            NOM := SYM ;
            CLASSE := C ;
            ADRESSE := PCODE_INST
         end
      end ;
      ...
end ;
```


B.2 Déclarations locales (chapitre 10)

Procédure PROCS d'analyse de déclarations de procédures avec allocation des variables locales.

```
procedure PROCS ;
var LIMIT, OLD_OFFSET : integer ;
begin
   while TOKEN = PROC_TOKEN do begin
      NEXT_TOKEN ;
      (* on entre le nom de la procedure en global *)
      TEST_ET_ENTRE (ID_TOKEN, PROCEDURE) ;
      TESTE (PT_VIRG_TOKEN) ;
      (* on memorise le dernier global *)
      LIMIT := DERNIERSYM ;
      (* on memorise l'offset *)
      OLD_OFFSET := OFFSET ;
      (* on met l'offset a zero pour les variables locales *)
      OFFSET := 0 ;
      if TOKEN = CONST_TOKEN then CONSTS ;
      if TOKEN = VAR_TOKEN then VARS ;
      (* on genere le INT pour les locaux *)
      GENERER2 (INT, OFFSET) ;
      INSTS ;
      TESTE (PT_VIRG_TOKEN) ;
      (* on oublie les declarations locales *)
      DERNIERSYM := LIMIT ;
      OFFSET := OLD_OFFSET ;
      GENERER1 (RET)
   end
end ;
```

Les instructions CAL et RET de l'interpréteur sont modifiées et l'instruction LDL est ajoutée:

```
with INST do
   case MNE of
      ...
      CAL : begin
         (* on empile l'adresse de retour *)
         SP := SP + 1 ;
         MEM [SP] := PC ;
         (* on empile la base de la zone de l'appelant *)
         SP := SP + 1 ;
         MEM [SP] := BASE ;
         (* mise a jour de BASE *)
         BASE := SP -2 ;
         PC := SUITE
      end ;
      RET : begin
```

```
                            |          |
                            |----------|
                            |    C     |
                            |----------|
                            |    ------|---------+
                            |----------|         |
                +-----------|-----     |         |
                |           |----------|         |
BASE  --------------------->|    AR    |         |
                |           |==========|         |
                |           |    B     |         |
                |           |----------|         |
                |           |    ------|---------+
                |           |----------|         |
                | +---------|-----     |         |
 chainage       | |         |----------|         |    chainage
 dynamique      +-|-------->|    AR    |         |    statique
                  |         |==========|         |
                  |         |    B     |         |
                  |         |----------|         |
                  |         |    ------|-----+   |
                  |         |----------|     |   |
                  | +-------|-----     |     |   |
                  | |       |----------|     |   |
                  +-|------>|    AR    |<----|---+
                    |       |==========|     |
                    |       |    A     |     |
                    +------>+----------+<----+
```

FIG. B.1 – *Liens de chaînage statique et dynamique dans la pile, positionnement de* `BASE`.

```
      (* on recupere l'adresse de retour *)
      PC := MEM [BASE] ;
      (* on depile toute la zone *)
      SP := BASE - 1 ;
      (* on recupere l'adresse de la zone de l'appelant *)
      BASE := MEM [BASE + 1]
    end ;
    LDL : begin
      (* on empile l'adresse relative au debut de la zone *)
      SP := SP + 1 ;
      MEM [SP] := BASE + 2 + SUITE
    end ;
end
```

B.3 Procédures imbriquées (chapitre 11)

Exercice 1

Le code de l'interpréteur pour l'instruction `LDL` est le suivant (cf. `BASE` sur la figure B.1)


```
LDL SUITE1 SUITE2
  i := SUITE2 ; Z := BASE ;
  while i > 0 do begin
    Z :== MEM [Z] ;
    i := i - 1
  end ;
  SP := SP + 1 ;
  MEM [SP] := Z + 3 + SUITE1 ;
```

Le `+ 3` correspond à `AR` et aux deux chaînages.

Bibliographie

[1] Alfred Aho, Ravi Sethi, et Jeffrey Ullman. *Compilateurs—Principes, Techniques et Outils*, InterÉditions, 1989 (version originale anglaise, Addison-Wesley, 1986).

[2] Jean-Marc Geib. *Génération de code, 1re partie*, LIFL, USTL, Oct. 1987.

[3] Per Brinch Hansen. *Brinch Hansen on Pascal Compilers*, Prentice-Hall, 1985

[4] Patrick D. Terry. *Programming Languages Translation, A pratical Approach*, Addison-Wesley, 1986.

Index