

Structure de la mémoire centrale
de la machine virtuelle

De plus, la mémoire de la machine virtuelle a une taille maximale fixe.

Differentes représentations de la mémoire sont possibles. On peut choisir, par exemple, de représenter la mémoire par un seul tableau:

```
type MOT_MEM = integer;           { mot mémoire }
const TAILLE_MAX_MEM = 10000;     { taille maximale de la mémoire centrale }
var MEM: array[0..TAILLE_MAX_MEM] of MOT_MEM; { mémoire centrale }
```

On peut aussi représenter les trois zones de la mémoire par des structures de données différentes.

Dans la suite, indépendamment de la représentation choisie, nous désignerons par:

- MEMVAR, la zone mémoire réservée aux variables globales,
- P_CODE, la zone mémoire réservée au code machine,
- PILEX, la zone mémoire réservée à la pile d'exécution et SOM_PILEX, le sommet de la pile d'exécution.

⇒ Instructions de la machine virtuelle

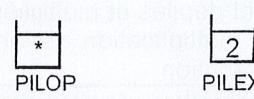
Toutes les opérations arithmétiques sont réalisées sur des valeurs dans la pile d'exécution PILEX. Le code de la machine virtuelle pour une expression arithmétique simule l'évaluation d'une représentation postfixée de l'expression en utilisant une pile. L'évaluation procède en traitant la représentation postfixée de gauche à droite, en empilant la valeur de chaque opérande dès sa rencontre. Quand un opérateur binaire est rencontré, son argument gauche est sous le sommet de la pile et son argument droit est au sommet de la pile. L'évaluation dépile les deux opérandes, applique l'opérateur aux deux opérandes et empile le résultat. La valeur au sommet de la pile à la fin est la valeur de l'expression arithmétique complète.

Exemple: Soit l'expression arithmétique $2 * 3 + 5$, équivalente à $2 * (3 + 5)$ puisque les opérateurs binaires du langage mini-Pascal ont la même priorité et sont associatifs à droite. La notation postfixée de cette expression est: $2\ 3\ 5\ +\ *$. Pour évaluer l'expression, on utilise une pile PILOP pour mémoriser les opérateurs, et les actions suivantes sont réalisées sur les piles PILEX et PILOP:

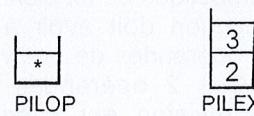
1. empiler 2 dans PILEX



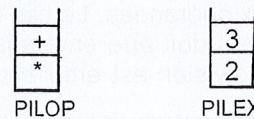
2. mémoriser l'opérateur de multiplication dans la pile des opérateurs PILOP



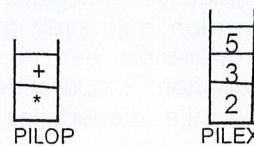
3. empiler 3 dans PILEX



4. mémoriser l'opérateur d'addition dans la pile des opérateurs PILOP



5. empiler 5 dans PILEX



6. dépiler l'opérateur au sommet de la pile PILOP et effectuer l'opération correspondante (+) sur les deux opérandes au sommet de la pile PILEX



7. dépiler l'opérateur au sommet de la pile PILOP et effectuer l'opération correspondante (*) sur les deux opérandes au sommet de la pile PILEX



Le processeur de la machine dispose d'un compteur ordinal **co** pointant sur la prochaine instruction à exécuter. Ainsi, lorsque **P_CODE[co] = ADDI**, c'est l'instruction arithmétique d'addition qu'il faut exécuter.

Les instructions de la machine virtuelle sont plutôt limitées et relèvent d'une des cinq classes suivantes: arithmétique entière, entrée/sortie, affectation, manipulation de pile et contrôle de flot:

- les instructions arithmétiques ADDI, SOUS, MULT, DIVI et MOIN;
- les instructions d'entrée/sortie LIRE, ECRL, ECRE et ECRC;
- l'instruction d'affectation AFFE;
- les instructions de manipulation de pile EMPI et CONT;
- l'instruction de contrôle de flot STOP.

Toutes ces instructions, à part ECRL, ECRC et STOP, manipulent le sommet de la pile. Seules les instructions ECRC et EMPI nécessitent des opérandes.

Instructions du P_CODE	Interprétation	
ADDI	Instruction arithmétique d'addition: La pile d'exécution doit avoir à son sommet les 2 opérandes de l'addition à effectuer. Ces 2 opérandes sont dépliés et additionnés. Le résultat de l'addition est empilé sur la pile d'exécution.	PILEX[SOM_PILEX-1] := PILEX[SOM_PILEX-1] + PILEX[SOM_PILEX]; SOM_PILEX := SOM_PILEX - 1; CO := CO + 1;
SOUS	Instruction arithmétique de soustraction: La pile d'exécution doit avoir à son sommet les 2 opérandes de la soustraction à effectuer. Ces 2 opérandes sont dépliés. Celui qui était au sommet est soustrait de celui qui était juste en dessous. Le résultat de la soustraction est empilé sur la pile d'exécution.	PILEX[SOM_PILEX-1] := PILEX[SOM_PILEX-1] - PILEX[SOM_PILEX]; SOM_PILEX := SOM_PILEX - 1; CO := CO + 1;
MULT	Instruction arithmétique de multiplication: La pile d'exécution doit avoir à son sommet les 2 opérandes de la multiplication à effectuer. Ces 2 opérandes sont dépliés et multipliés. Le résultat de la multiplication est empilé sur la pile d'exécution.	PILEX[SOM_PILEX-1] := PILEX[SOM_PILEX-1] * PILEX[SOM_PILEX]; SOM_PILEX := SOM_PILEX - 1; CO := CO + 1;
DIVI	Instruction arithmétique de division: La pile d'exécution doit avoir à son sommet les 2 opérandes de la division à effectuer. Ces 2 opérandes sont dépliés. Une division est effectuée entre ces deux opérandes. Le cas de la division par zéro doit être envisagé. Le résultat de la division est empilé sur la pile d'exécution	if PILEX[SOM_PILEX] = 0 then /* erreur d'exécution: division par zero */ else begin PILEX[SOM_PILEX-1] := PILEX[SOM_PILEX-1] / PILEX[SOM_PILEX]; SOM_PILEX := SOM_PILEX - 1; CO := CO + 1 end;
MOIN	Instruction arithmétique de négation: La pile d'exécution doit avoir à son sommet un opérande dont il faut calculer la négation. L'opérande au sommet de la pile d'exécution est changé de signe.	PILEX[SOM_PILEX] := - PILEX[SOM_PILEX]; CO := CO + 1;
AFFE	Instruction d'affectation: La pile d'exécution doit avoir à son sommet deux nombres: une valeur entière et l'adresse d'une cellule	MEMVAR[PILEX[SOM_PILEX-1]] := PILEX[SOM_PILEX]; SOM_PILEX := SOM_PILEX - 2; CO := CO + 1;

	mémoire. Ces deux nombres sont dépliés et la variable dopnt l'adresse est donnée par le second nombre est affectée avec la valeur donnée par le premier nombre.	
LIRE	Instruction de lecture: La pile d'exécution doit avoir à son sommet l'adresse d'une cellule mémoire. Un nombre entier est lu à partir du clavier et est placé à l'adresse indiquée par le sommet de la pile d'exécution. Cette adresse est ensuite dépliée.	readln(MEMVAR[PILEX[SOM_PILEX]]); SOM_PILEX := SOM_PILEX - 1; CO := CO + 1;
ECRL	Instruction d'écriture d'un saut de ligne: Aucune condition quant à l'état de la pile d'exécution n'est nécessaire. Un saut de ligne est affiché à l'écran.	writeln; CO := CO + 1;
ECRE	Instruction d'écriture d'un entier: La pile d'exécution doit avoir à son sommet un nombre entier. Ce nombre est déplié et affiché à l'écran.	write(PILEX[SOM_PILEX]); SOM_PILEX := SOM_PILEX - 1; CO := CO + 1;
ECRC chaine	Instruction d'écriture d'une chaîne: Aucune condition quant à l'état de la pile d'exécution n'est nécessaire. Tous les mots mémoire qui suivent l'instruction ECRC jusqu'au caractère FINC sont considérés comme formant une chaîne de caractères. Cette chaîne est affichée à l'écran. Une chaîne de caractères doit obligatoirement se terminer par le caractère FINC.	i := 1; ch := chr(P_CODE[CO+i]); while ch <> FINC do begin write(ch); i := i + 1; ch := chr(P_CODE[CO+i]) end; CO := CO + i + 1;
EMPI op	Instruction d'empilement sur la pile d'exécution: Aucune condition quant à l'état de la pile d'exécution n'est nécessaire. L'opérande op est placé au sommet de la pile.	SOM_PILEX := SOM_PILEX + 1; PILEX[SOM_PILEX] := P_CODE[CO+1]; CO := CO + 2;
CONT	Instruction d'accès au contenu d'une adresse: La pile d'exécution doit avoir à son sommet l'adresse d'une cellule mémoire. Le sommet de la pile est remplacé par l'élément stocké à l'adresse indiquée par ce sommet.	PILEX[SOM_PILEX] := MEMVAR[PILEX[SOM_PILEX]]; CO := CO + 1;
STOP	Instruction d'arrêt: Aucune condition quant à l'état de la pile d'exécution n'est nécessaire. Cette instruction provoque l'arrêt de l'exécution du programme.	

⇒ Exemple

Soit le programme mini-Pascal suivant:

```
(1) programme exemple;
(2) var nb;
(3) debut
(4)   ecrire('Entrez un nombre entier: ');
(5)   lire(nb);
(6)   nb := 2 * nb;
(7)   ecrire();
(8)   ecrire('Le double de ce nombre est: ', nb)
(9) fin.
```

Lors de la compilation de ce programme, en plus de l'analyse lexicale, syntaxique et sémantique du programme, le code machine suivant doit être généré:

ECRC 'E' 'n' 't' 'r' 'e' 'z' " " " u' 'n' " " " n' 'o' 'm' 'b' 'r' 'e' " " " e' 'n' 't' 'i' 'e' 'r' " " " FINC (4)

EMPI 0 (5)

LIRE

EMPI 0

EMPI '2
EMPI 0
CONT
MULT
AFFE

(6)

ECRL
ECRC 'L' 'e' ' ' 'd' 'o' 'u' 'b' 'l' 'e' ' ' 'd' 'e' ' ' 'c' 'e' ' ' 'n' 'o' 'm' 'b' 'r' 'e' ' ' 'e' 's' 't' ' ' ' FINC
EMPI 0
CONT
ECRE
STOP

(7)
(8)
(5)
(9)

Puisque les mots mémoire sont des entiers, et que le code machine est rangé en mémoire, il faut associer un entier à chaque instruction mémoire:

ADDI	SOUS	MULT	DIVI	MOIN	AFFE	LIRE	ECRL	ECRE	ECRC	FINC	EMPI	CONT	STOP
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Après la compilation de ce programme, l'état de la mémoire de la machine est le suivant:

- zone des variables globales: Une cellule mémoire a été réservée pour la seule variable globale nb, l'adresse de la variable nb est 0

(nb) 0 |

- zone du code machine: P_CODE[CO] = 9

ECRC	9	← CO
ord('E')	69	
ord('n')	110	
...	...	
ord(' ')	32	
FINC	10	
EMPI	11	
adresse de nb	0	
LIRE	6	
EMPI	11	
adresse de nb	0	
EMPI	11	
nombre entier 2	2	
EMPI	11	
adresse de nb	0	
CONT	12	
MULT	2	
AFFE	5	
ECRL	7	
ECRC	9	
ord('L')	76	
ord('e')	101	
...	...	
ord(' ')	32	
FINC	10	
EMPI	11	
adresse de nb	0	
CONT	12	
ECRE	8	
STOP	13	

⇒ Génération du code pour la machine virtuelle

La génération du code pour la machine virtuelle est introduite au sein du programme d'analyse syntaxique, dans les fonctions booléennes associées aux non terminaux de la grammaire G_0 .
 Aucun code n'est généré lors de la déclaration des constantes.

Aucun code n'est généré lors de la déclaration des variables. La gestion des adresses des variables globales a déjà été rajoutée lors de l'analyse sémantique.

Notation: $@x$ désigne l'adresse de la variable globale x .

■ Fonction associée au non terminal AFFECTATION de la grammaire G_0

Exemples:	$i := 3$	$\text{EMPI } @i \quad \leftarrow \text{code généré dans la fonction AFFECTATION}$
		$\text{EMPI } 3 \quad \leftarrow \text{code généré lors de l'appel de la fonction EXP par la fonction AFFECTATION}$
		$\text{AFFE} \quad \leftarrow \text{code généré dans la fonction AFFECTATION}$
	$nb := 2+3$	$\text{EMPI } @nb \quad \leftarrow \text{code généré dans la fonction AFFECTATION}$
		$\text{EMPI } 2$
		$\text{EMPI } 3 \quad \leftarrow \begin{cases} \text{code généré lors de l'appel de la fonction EXP par la fonction AFFECTATION} \\ \text{ADDI} \end{cases}$
		$\text{AFFE} \quad \leftarrow \text{code généré dans la fonction AFFECTATION}$

- Après la reconnaissance de l'identificateur, il faut générer le code qui empile l'adresse de l'identificateur sur la pile d'exécution:

```
PCODE[CO]:= EMPI;
PCODE[co+1]:= TABLE_IDENT[chercher(CHAINE)].adrV;
CO := CO+2;
```

- Le code correspondant à l'expression est généré lors de l'appel de la fonction EXP.

- A la fin de la reconnaissance de l'instruction d'affectation, il faut produire l'instruction AFFE:

```
PCODE[CO]:= AFFE;
CO := CO+1;
```

■ Fonction associée au non terminal LECTURE de la grammaire G_0

Exemples:	$LIRE(i);$	$\text{EMPI } @i \quad \leftarrow \begin{cases} \text{code généré dans la fonction LECTURE} \\ \text{LIRE} \end{cases}$
	$LIRE(a,b);$	$\text{EMPI } @a \quad \leftarrow \begin{cases} \text{code généré dans la fonction LECTURE} \\ \text{LIRE} \\ \text{EMPI } @b \\ \text{LIRE} \end{cases}$

Après la reconnaissance de chaque identificateur dans la fonction LECTURE, il faut empiler l'adresse de l'identificateur sur la pile d'exécution et produire l'instruction LIRE:

```
PCODE[CO]:= EMPI;
PCODE[co+1]:= TABLE_IDENT[chercher(CHAINE)].adrV;
PCODE[co+2]:= LIRE;
CO := CO+3;
```

■ Fonction associée au non terminal ECRITURE de la grammaire G_0

Exemples:	$ECRIRE();$	$\text{ECRL} \quad \leftarrow \text{code généré dans la fonction ECRITURE}$
	$ECRIRE('2*i = ', 2*i);$	$\text{ECRC } '2' '*' 'i' '=' '' FINC \quad \leftarrow \begin{cases} \text{code généré lors de l'appel de la fonction ECR_EXP par la fonction ECRITURE} \\ \text{EMPI } 2 \\ \text{EMPI } @i \\ \text{CONT} \\ \text{MULT} \end{cases}$
		$\leftarrow \begin{cases} \text{code généré lors de l'appel de la fonction EXP par la fonction ECR_EXP, elle-même appellée par la fonction ECRITURE} \\ \text{ECRE} \end{cases}$
		$\leftarrow \begin{cases} \text{code généré lors de l'appel de la fonction ECR_EXP par la fonction ECRITURE} \\ \text{ECRE} \end{cases}$

Si aucune « ECR_EXP » n'est présente dans l'instruction d'écriture, alors il faut produire l'instruction ECRL:

```
PCODE[CO]:= ECRL;
CO := CO+1;
```

■ Fonction associée au non terminal ECR_EXP de la grammaire G_0

- Si une expression est reconnue, le code correspondant à l'expression est généré lors de l'appel de la fonction EXP. Il suffit de produire, après l'appel de EXP, l'instruction ECRE:

```
PCODE[CO]:= ECRE;
```

CO := CO+1;

- Sinon, une chaîne est reconnue. A la fin de la reconnaissance de la chaîne, il faut produire l'instruction ECRC, ajouter à la suite dans le P-CODE, les codes ASCII de chacun des caractères de la chaîne reconnue, et produire l'instruction FINC:

```
PCODE[CO]:= ECRC;  
for i := 1 to length(CHAINE) do  
    PCODE[CO+i]:= ord(CHAINE[i]);  
PCODE[CO+length(CHAINE)+1]:= FINC;  
CO := CO+length(CHAINE)+2;
```

■ Fonction associée au non terminal EXP de la grammaire G_0

Exemples: 3 EMPI 3 ← code généré lors de l'appel de la fonction TERME

nb EMPI @nb } code généré lors de l'appel de la fonction TERME
CONT }

(2) EMPI 2 ← code généré lors de l'appel de la fonction TERME

-1 EMPI 1 } code généré lors de l'appel de la fonction TERME
MOIN }

2+3*5 EMPI 2 ← code généré lors de l'appel de la fonction TERME

EMPI 3 ← code généré lors de l'appel de la fonction TERME

EMPI 5 ← code généré lors de l'appel de la fonction TERME

MULT ← code généré lors de l'appel de la fonction EXP

ADDI ← code généré lors de l'appel de la fonction EXP

- Si l'expression est réduite à un terme, le code correspondant à l'expression est généré lors de l'appel de la fonction TERME.

- Sinon, l'expression E est de la forme « TERME OP_BIN EXP ». Soit T, O et E1 le terme, l'opérateur et l'expression qui forment cette expression E. Le code correspondant au terme T est généré lors de l'appel de la fonction TERME. Le code de l'opération correspondant à l'opérateur O est mémorisé dans la pile PILOP, lors de l'appel de la fonction OP_BIN. Le code correspondant à l'expression E1 est généré lors de l'appel de la fonction EXP. Après la reconnaissance de l'expression E1, il faut dépiler et produire l'instruction associée à l'opération correspondant à l'opérateur O:

```
PCODE[CO]:= PILOP[SOM_PILOP];  
SOM_PILOP := SOM_PILOP - 1;  
CO := CO+1;
```

SOM_PILOP désigne le sommet de la pile PILOP.

■ Fonction associée au non terminal OP_BIN de la grammaire G_0

Chaque fois qu'un opérateur binaire est reconnu, l'opération correspondante est empilée dans la pile PILOP et sera utilisée dans la fonction EXP, à la fin de la reconnaissance d'une expression de la forme « TERME OP_BIN EXP »:

```
SOM_PILOP := SOM_PILOP + 1;  
if CHAINE = '+' then  
    PILOP[SOM_PILOP] := ADDI  
else if CHAINE = '-' then  
    PILOP[SOM_PILOP] := MOIN  
else if CHAINE = '*' then  
    PILOP[SOM_PILOP] := MULT  
else  
    PILOP[SOM_PILOP] := DIVI;
```

■ Fonction associée au non terminal TERME de la grammaire G_0

- Si le terme est un entier, le code à produire est:

```
PCODE[CO]:= EMPI;  
PCODE[CO+1]:= NOMBRE;  
CO := CO+2;
```

- Sinon, si le terme est un identificateur, le code à produire est:

```
PCODE[CO]:= EMPI;  
PCODE[CO+1]:= TABLE_IDENT[chercher(CHAINE)].adrV;  
PCODE[CO+2]:= CONT;  
CO := CO+3;
```

- Sinon, si le terme est une expression entre parenthèses, le code est généré lors de l'appel de la fonction EXP.

- Sinon, le terme T est de la forme « - TERME ». Soit T1 le terme qui forme T. Le code du terme T1 est généré lors de l'appel de la fonction TERME. Après la reconnaissance du terme T1, il faut produire l'instruction MOIN:

```
PCODE[CO]:= MOIN;  
CO := CO+1;
```

■ Remarque

Les instructions de génération de code sont rajoutées dans le corps de l'analyseur. Ceci rend la lecture du programme difficile. Un autre désavantage est que le programme est difficile à modifier lorsque la machine cible

Change: l'analyseur (lexical, syntaxique et sémantique) doit être conservé, seule la partie génération de code doit être modifiée.

Une solution à ces deux problèmes est de regrouper les instructions de la génération de code dans des procédures qui sont alors appelées par l'analyseur. Une plus grande lisibilité sera obtenue si, de plus, le début du nom de ces procédures est toujours identique, par exemple: GENCODE_.

Exemple: procedure GENCODE_LECTURE(ch: string);
begin

```
PCODE[CO]:= EMPI;  
PCODE[CO+1]:= TABLE_IDENT[chercher(ch)].adrV;  
PCODE[CO]:= LIRE;  
CO := CO+3  
end
```

et les instructions de génération de code ajoutées dans la fonction LECTURE sont remplacées par l'appel: GENCODE_LECTURE(CHAINE);

■ Procédure CREER_FICHIER_CODE

La procédure CREER_FICHIER_CODE crée un fichier contenant le code généré. Le fichier doit avoir le même nom que le fichier contenant le programme source mini-Pascal, mais avec l'extension COD. Ce fichier pourra ensuite être édité par l'utilisateur.

Si PROG1.MP est le programme mini-Pascal de l'exemple de la page 13, le fichier PROG1.COD doit contenir le code généré sous la forme suivante:

```
1 mot(s) réservé(s) pour les variables globales  
ECRC 'Entrez un nombre entier: ' FINC  
EMPI 0  
LIRE  
EMPI 0  
EMPI 2  
EMPI 0  
CONT  
MULT  
AFFE  
ECRL  
ECRC 'Le double de ce nombre est: ' FINC  
ECRE 0  
STOP
```

⇒ Interprétation du code

Une fois le programme compilé, l'analyse lexicale, syntaxique, sémantique et la génération de code ont été effectuées. Si le programme est correct, il doit être exécuté. Pour exécuter le programme, il suffit d'interpréter le code machine généré. Le code machine contenu par la structure P_CODE est parcouru à partir du CO initial, jusqu'à l'instruction STOP et chaque instruction du P_CODE est interprétée.

■ Procédure INTERPRETER

```
procedure INTERPRETER;  
{ interprétation des instructions du P_CODE pour l'exécution des programmes mini-Pascal }  
begin  
  while P_CODE[CO] <> STOP do  
  begin  
    case P_CODE[CO] of:  
      ADDI: begin  
        PILEX[SOM_PILEX-1] := PILEX[SOM_PILEX-1] + PILEX[SOM_PILEX];  
        SOM_PILEX := SOM_PILEX - 1;  
        CO := CO + 1  
      end;  
      SOUS: ...  
      MULT: ...  
      DIVI: ...  
      MOIN: ...  
      AFFE: ...  
      LIRE: ...  
      ECRL: ...  
      ECRE: ...  
      ECRC: ...  
      EMPI: ...  
      CONT: ...  
    end  
  end
```

```
    end  
end;
```

■ Le programme principal

Le programme principal du compilateur et de l'interpréteur doit:

- appeler la procédure INITIALISER,
- appeler la procédure ANASYNT,
- appeler la procédure TERMINER,
- appeler la procédure CREER_FICHIER_CODE, (création d'un fichier contenant le code généré)
- appeler la procédure INTERPRETER. (interprétation)

A ce point, le programme est capable de lire un fichier source contenant un programme mini-Pascal, de vérifier que le programme est lexicalement, syntaxiquement et sémantiquement correct, conformément à la définition du langage mini-Pascal, et de produire le code machine correspondant.

- si une erreur a été détectée, un message d'erreur est affiché et le programme du compilateur-interpréteur s'interrompt;
- si aucune erreur n'a été détectée, le programme du compilateur-interpréteur a créé un fichier contenant le code machine généré et exécute le programme mini-Pascal du fichier source.