
MASTER 1re ANNÉE, COMPILATION

# Projet : Compilateur pseudo-pascal

UNIVERSITÉ CLAUDE BERNARD LYON 1, DÉPARTEMENT INFORMATIQUE

Révision : 1.0.5
En date du : 9 octobre 2014

*Auteur:*
**Erwan Guillou**

*Année:*
**2014-2015**

# *Table des matières*

# *Chapitre 1*

# *Le langage*

## Sommaire

## 1.1 Les identificateurs

Ils servent à donner un nom à un objet.

**Syntaxe**
On appelle lettre un caractère de 'a'..'z' ou 'A'..'Z' ou '_'.
On appelle digit un caractère de '0'..'9'.
Un identificateur Pascal est une suite de lettres ou de digit accolées, commençant par une lettre.

**Exemples**
x, y1, jour, mois, annee, NbCouleurs, longueur_ligne.

**Remarques**
- Il n'y a pas de différence entre minuscules et majuscules.
- On n'a pas le droit de mettre d'accents, ni de caractêres de ponctuation.
- Un identificateur doit être différent des mots clés (begin, function, real, . . .).

On se sert des identificateurs pour : le nom du programme, les noms de variables, les noms de constantes, les noms de types.

## 1.2 Les types prédéfinis

Un type décrit un ensemble de valeurs et un ensemble d'opérateurs sur ces valeurs.

### 1.2.1 Type entier : integer

Entier signé en complément à deux sur 16 bits.
Sur 16 bits, un entier à sa valeur dans $-32768 ... +32767$.

**Opérateurs sur les entiers**
- + x identité.
- - x signe opposé.
- x + y addition.
- x - y soustraction.
- x * y multiplication.
- x / y division, fournissant un résultat de type réel.
- x div y dividende de la division entière de x par y.
- x mod y reste de la division entière, avec y non nul.

**Remarques**
- Attention, les opérateurs /, div et mod, produisent une erreur à l'exécution si y est nul.
- Lorsqu'une valeur (ou un résultat intermédiaire) dépasse les bornes au cours de l'exécution, on a une erreur appelée débordement arithmétique.

### 1.2.2 Type réel : real

Leur domaine de définition dépend de la machine et du compilateur utilisés. On code un réel avec une certaine précision, et les opérations fournissent une valeur approchée du résultat dit < juste >.

**Exemples de real** 0.0 ; -21.4E3 (= $-21.4 * 10^3$ = -21400) ; 1.234E-2 (= $1.234 * 10^{-2}$)

**Opérateurs sur les réels :**
- + x identité.
- - x signe opposé.
- x + y addition.
- x - y soustraction.
- x * y multiplication.
- x / y division, fournissant un résultat de type réel.

**Remarques**
- Si l'un au moins des 2 arguments est réel, le résultat est réel pour : x - y, x + y, x * y.
- Résultat réel que l'argument soit entier ou réel : x / y (y doit être non nul)

### 1.2.3 Type caractère : char

Le jeux des caractères comportant les lettres, les digits, l'espace, les ponctuations, etc, est codé sur un octet non signé.

Le choix et l'ordre des 256 caractères possible dépend de la machine et de la langue. Sur PC, on utilise le code ASCII, où 'A' est codé par 65, 'B' par 66, 'a' par 97, ' ' par 32, '' par 123, etc.

Le code ascii est organisé comme suit : de 0 à 31, sont codés les caractères de contrôle (7 pour le signal sonore, 13 pour le saut de ligne, etc). De 32 à 127, sont codés les caractères et ponctuations standards et internationaux. Enfin de 128 à 255, sont codés les caractères accentués propres à la langue, et des caractères semi-graphiques.

**Remarques**
- Le caractère apostrophe se note "'".
- Une suite de caractères telle que 'Il y a' est une chaîne de caractères ; il s'agit d'un objet de type string, que l'on verra plus loin.

### 1.2.4 Type booléen : boolean

Utilisé pour les expressions logiques.
Deux valeurs : false (faux) et true (vrai).

**Opérateurs sur les booléens :**

- not (négation), and (et), or (ou).
- Opérateurs de comparaison (entre 2 entiers, 2 réels, 1 entier et 1 réel, 2 chars, 2 booléens) : <, >, <=, >=, = (égalité, à ne pas confondre avec l'attribution :=), <> (différent).
  Le resultat d'une comparaison est un booléen.
  On peut comparer 2 booléens entre eux, avec la relation d'ordre false < true.

**Remarques**

- En mémoire, les booléens sont codés sur 1 bit, avec 0 pour false et 1 pour true. De là les relations d'ordre.
- Les opérateurs booléens not, and, or s'apparentent approximativement à (1 - x), *, +.

# *Chapitre 2*

# *La machine cible*

## Sommaire

## 2.1 Description

La machine cible est de type 80x86 sur une architecture 32 bits. Le jeu d'instructions disponibles est similaire à celui d'Intel mais moins complexe.

L'objectif est de générer un exécutable pour cette machine. L'outil utilisé pour transformer le code assembleur généré en code machine sera Nasm [1].

## 2.2 Représentation des données en mémoire

| **Type** | Taille | Description |
|---|---|---|
| booléen | 8 bits | 0 équivaut à faux, le reste à vrai |
| char | 8 bits | code ascii du caractère |
| entier | 16 bits | entier signé |
| réel | 32 bits | nombre réel en virgule flottante |
| chaine | 8 + n*8bits | le premier octet contient la longueuer de la chaine<br>les octets suivants contiennent les codes ascii des caractères |

1. http ://nasm.us/

# *Chapitre 3*

# *Fonctionnement du compilateur*

## Sommaire

## 3.1 Arguments de la ligne de commande

Le compilateur que vous implémentez (et qui sera nommé **ppc**) devra accepter comme arguments :

— *-c fichier.pas* le nom du fichier source à compiler. Ce fichier sera situé dans le répertoire **pascal**.
— *-O* active les optimisations
— *-a* active la génération du code assembleur

## 3.2 Affichage des messages

Le compilateur devra s'arrêter à la première erreur rencontrée en fournissant un message d'erreur informatif de la forme :

$$Erreur\,fichier : ligne.colonne : fonction, message$$

Lors de la détection de certains problèmes ne générant pas d'erreur (comme la perte de précision lors de la conversion d'un réel vers un entier, une division par 0, ou encore une boucle infinie), un warning devra être généré sous la forme :

$$Warning\,fichier : ligne.colonne : fonction, message$$

## 3.3 Sorties obligatoires

Dans ce qui suis, le terme *p*refixe correspond au nom du fichier à compiler sans son extension. Le terme *X*XX décrit le nom d'un sous-programme.

Votre compilateur devra au minimum générer les fichiers suivants :

— sauvegarder la table des identificateurs, avant optimisation, dans un fichier nommé *prefixe.ti*
— sauvegarder les tables des symboles, avant optimisation, dans des fichiers nommés *prefixe.XXX.ts*
— sauvegarder le code 3 adresses non optimisé dans des fichiers nommés *prefixe.XXX.3ad*

Les fichiers intermédiaires (ti, ts, code 3 adresses) seront générés dans le répertoire **intermediaire**.

Vous devrez implémenter les optimisations vues en cours, qui ne seront utilisées que si l'option *-O* est utilisée. Si les optimisations sont activées, votre compilateur devra :

— sauvegarder le graphe de flot de chaque sous programme dans des fichiers nommés *prefixe.XXX.flot* dans le répertoire **intermediaire**.
— sauvegarder les données intermédiaires post optimisation (ti, ts, code 3 adresses) seront sauvegardées dans des fichiers nommées *prefixe.optim.ti*, *prefixe.XXX.optim.ts* et *prefixe.XXX.optim.3ad*.

## 3.4 Sorties optionnelles

Une partie optionnelle du projet concerne les sorties effectives pour la machine cible :
— sauvegarder le code assembleur pré-optimisation dans un fichier nommé *prefixe.asm* dans le répertoire **asm**.
— sauvegarder le code assembleur post-optimisation dans un fichier nommé *prefixe.optim.asm* dans le répertoire **asm**.

## 3.5 Exemple

En utilisant la ligne de commande suivante :

$$ppc\ -c\ pgcd.pas\ -O\ -a$$

Le code source sera lu depuis le fichier **pascal/pgcd.pas** et le compilateur génerera les fichiers (voir code source en fin de document) :
— tables des identificateurs : **intermediaire/pgcd.ti** et **intermediaire/pgcd.optim.ti**
— tables des symboles du programme principal : **intermediaire/pgcd.pgcd.ts** et **intermediaire/pgcd.pgcd.optim.ts**
— tables des symboles de la fonction de calcul : **intermediaire/pgcd.calculpgcd.ti** et **intermediaire/pgcd.calculpgcd.optim.ti**
— code 3 adresses du programme principal : **intermediaire/pgcd.pgcd.3ad** et **intermediaire/pgcd.pgcd.optim.3ad**
— code 3 adresses de la fonction de calcul : **intermediaire/pgcd.calculpgcd.3ad** et **intermediaire/pgcd.calculpgcd.optim.3ad**
— graphe de flot du programme principal : **intermediaire/pgcd.pgcd.flot**
— graphe de flot de la fonction de calcul : **intermediaire/pgcd.calculpgcd.flot**
— code assembleur : **asm/pgcd.asm** et **asm/pgcd.optim.asm**

# *Chapitre 4*

# *La grammaire effective du langage*

**Sommaire**

## La grammaire effective du langage

La grammaire utilisée est une version simplifiée de la grammaire réelle du langage pascal :

- Le formalisme utilisé interdit les déclarations imbriquées de procédure ou fonction.
- Les Expressions utilisées dans les déclarations de constantes doivent être calculables (et donc calculées) lors de la phase de compilation du code source.
- La déclaration de variables de types complexes doit passer par une première phase de déclaration du type complexe.
- D'un point de vue sémantiques, les fonctions et procédures doivent être déclarées dans l'ordre (ie de manière à ce qu'un appel soit résolvable lors de son analyse sémantique).
- Comme pour les variables, le type de retour d'un fonction (s'il est complexe) doit passer par une phase de déclaration de type.
- Dans le cas d'une procédure ou fonction n'acceptant aucun argument, le parenthésage est obligatoirement omis.

| **Symbole** | **Dérivation** |
|---|---|
| Program | program id ; Block . |
| Block | BlockDeclConst BlockDeclType BlockDeclVar BlockDeclFunc BlockCode |
| BlockSimple | BlockDeclConst BlockDeclVar BlockCode |
| BlockDeclConst | const ListDeclConst<br>*epsilon* |
| ListDeclConst | ListDeclConst DeclConst<br>DeclConst |
| DeclConst | id = Expression ;<br>id : BaseType = Expression ; |
| BlockDeclType | type ListDeclType<br>*epsilon* |
| ListDeclType | ListDeclType DeclType<br>DeclType |
| DeclType | id = Type ; |
| BlockDeclVar | var ListDeclVar<br>*epsilon* |
| ListDeclVar | ListDeclVar DeclVar<br>DeclVar |
| DeclVar | ListIdent : SimpleType ; |
| ListIdent | ListIdent , id<br>id |
| BlockDeclFunc | ListDeclFunc ;<br>*epsilon* |
| ListDeclFunc | ListDeclFunc ; DeclFunc<br>DeclFunc |
| DeclFunc | ProcDecl<br>FuncDecl |

*suite sur la prochaine page*

| Symbole | Dérivation |
|---|---|
| ProcDecl | ProcHeader ; BlockSimple |
| FuncDecl | FuncHeader ; BlockSimple |
| ProcHeader | procedure id FormalArgs |
| FuncHeader | function id FormalArgs : SimpleType |
| FormalArgs | ( ListFormalArgs )<br>*epsilon* |
| ListFormalArgs | ListFormalArgs ; FormalArg<br>FormalArg |
| FormalArg | ListIdent : SimpleType<br>var ListIdent : SimpleType |
| Type | UserType<br>SimpleType |
| UserType | EnumType<br>InterType<br>ArrayType<br>RecordType |
| SimpleType | id<br>BaseType |
| BaseType | integer<br>real<br>boolean<br>char<br>string |
| EnumType | ( ListEnumValue ) |
| ListEnumValue | ListEnumValue , id<br>id |
| InterType | NSInterBase .. NSInterBase |
| NSInterBase | id<br>TOKINTEGER<br>TOKCHAR |
| ArrayType | array [ ArrayIndex ] of SimpleType |
| ArrayIndex | id<br>InterType |
| RecordType | record RecordFields end |
| RecordFields | RecordFields ; RecordField<br>RecordField |
| RecordField | ListIdent : SimpleType |
| BlockCode | begin ListInstr end<br>begin ListInstr ; end<br>begin end |
| ListInstr | ListInstr SEPSCOL Instr |

*suite sur la prochaine page*

## La grammaire effective du langage

| Symbole | Dérivation |
|---|---|
| | Instr |
| Instr | while Expression do Instr<br>repeat ListInstr until Expression<br>for id := Expression to Expression do Instr<br>for id := Expression downto Expression do Instr<br>if Expression then Instr<br>if Expression then Instr else Instr<br>VarExpr := Expression<br>write ( ListeExpr )<br>writeln ( ListeExpr )<br>read ( VarExpr )<br>id ( ListeExpr )<br>id<br>BlockCode |
| Expression | MathExpr<br>CompExpr<br>BoolExpr<br>AtomExpr<br>VarExpr<br>id ( ListeExpr ) |
| MathExpr | Expression + Expression<br>Expression - Expression<br>Expression * Expression<br>Expression / Expression<br>Expression div Expression<br>Expression mod Expression<br>Expression ** Expression<br>- Expression<br>+ Expression |
| CompExpr | Expression = Expression<br>Expression <> Expression<br>Expression < Expression<br>Expression <= Expression<br>Expression > Expression<br>Expression >= Expression |
| BoolExpr | Expression and Expression<br>Expression or Expression<br>Expression xor Expression<br>not Expression |
| AtomExpr | ( Expression )<br>TOKINTEGER |

*suite sur la prochaine page*

| **Symbole** | **Dérivation** |
|---|---|
| | TOKREAL<br>TOKBOOLEAN<br>TOKCHAR<br>TOKSTRING |
| VarExpr | id<br>VarExpr [ ListIndices ]<br>VarExpr . id |
| ListeExpr | ListeExpr , Expression<br>Expression |
| ListIdent | ListIdent , id<br>id |

# *Chapitre 5*

# *Le code intermédiaire*

## Sommaire

Le code 3 adresses est le langage intermédiaire permettant de faire certaines optimisations indépendantes de la machine cible. Les opérandes d'une instruction sont définies soient par des valeurs, soit par des références aux identificateurs. Les instructions définies sont non typées (le typage est donné par la table des symboles).

## 5.1 Le jeu d'instructions

Le jeux d'instruction disponible est le suivant :

| **instruction** | **description** | **arguments** |
|---|---|---|
| nop | ne rien faire | |
| fin | termine l'exécution du programme | |
| appeler x | appeler un sous-programme | x est l'étiquette du sous-programme |
| reserver n | réserve la mémoire pour les données locales | n est le nombre d'octets à réserver pour les variables locales et les temporaires du sous-programme |
| retourner n | fin d'un sous-programme et retour à l'appelant | n correspond à la taille de l'espace mémoire alloué pour les arguments du sous-programme (y compris l'adresse du résultat dans le cas d'une fonction) |
| renvoyer x | initialise le résultat d'une fonction avec la valeur x<br>attention ceci ne termine pas l'exécution du sous-programme | x est soit un identificateur soit une constante |
| x := y + z<br>x := y - z<br>x := y * z<br>x := y / z<br>x := - y | réalise l'opération mathématique entre y et z<br>stocke le résultat dans x | x fait référence à un identificateur<br>y et z sont soit des identificateurs soit des constantes |
| x := y & z<br>x := y \| z<br>x := ! y | réalise l'opération booléenne entre y et z<br>stocke le résultat dans x | x fait référence à un identificateur<br>y et z sont soit des identificateurs soit des constantes |
| x := y < z | compare y et z<br>stocke le résultat dans x | x fait référence à un identificateur<br>y et z sont soit des identificateurs soit des constantes |

*suite sur la prochaine page*

| instruction | description | arguments |
|---|---|---|
| x := y > z<br>x := y <= z<br>x := y >= z<br>x := y == z<br>x := y != z | | |
| x := y | recopie la valeur de y dans x | x fait référence à un identificateur<br>y est soit un identificateur soit une constante |
| x := & y | stocke l'adresse de y dans x | x et y font référence à des identificateurs |
| x := * y | stocke dans x le contenu de la case mémoire pointée par y | x et y font référence à des identificateurs |
| * x := y | recopie la valeur de y dans la case mémoire pointée par x | x fait référence à un identificateur<br>y est soit un identificateur soit une constante |
| aller a x | la prochaine instruction à exécuter est celle définie par l'étiquette x | x est un identificateur d'étiquette |
| si y aller a x | la prochaine instruction à exécuter est celle définie par l'étiquette x si la valeur de y est vrai | x est un identificateur d'étiquette<br>y est un identificateur |
| empiler x | stocke la valeur de x en sommet de pile | x fait référence à un identificateur ou une constante |
| empiler & x | stocke l'adresse de x en sommet de pile | x fait référence à un identificateur |
| empiler * x | stocke le contenu de la cas mémoire pointée par x en sommet de pile | x fait référence à un identificateur |
| depiler x | dépile le sommet de pile et stocke la valeur dans x | x fait référence à un identificateur |
| depiler * x | dépile le sommet de pile et stocke la valeur dans la case mémoire pointée par x | x fait référence à un identificateur |
| ecrire x | écrit la valeur de x sur la sortie standard | x de type primitif |
| lire x | lit la valeur de x sur l'entrée standard | x de type primitif |
| sautdeligne | passe à la ligne sur la sortie standard | |

## 5.2 Nommage des temporaires et étiquettes

— les temporaires (stockage des calculs intermédiaires) seront nommés selon le schéma suivant :

*.__tempddd*

où ddd est une numérotation globale au programme (démarrant à 0).

— les étiquettes de branchement seront nommées selon la structure de contrôle utilisée, avec une numérotation liée à cette structure de contrôle (numérotation globale au programme, voir la partie schéma de traduction du cours).

$$.siXXX, .alorsXXX, .sinonXXX, .finsiXXX$$

$$.whileXXX, .bclwhileXXX, .finwhileXXX$$

$$.repeatXXX, .finrepeatXXX$$

$$.forXXX, .testforXXX, .finforXXX$$

— les étiquettes des sous-programmes seront nommées selon le schéma suivant :

$$xxxxx$$

où xxxxx est le nom du sous-programme.

Pour plus de clarté, les étiquettes seront placées sur des instructions vides (ie *nop*, voir ci-dessous).

# *Chapitre 6*

# *Schéma de traduction*

## Sommaire

## 6.1 Les expressions

### 6.1.1 Les opérateurs binaires

$$E \rightarrow E_1 \text{ op } E_3$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

| **Code 3 adresses** |
|---|
| $E_1$.code<br>$E_3$.code<br>$E$.place = $E_1$.place op $E_3$.place |

### 6.1.2 Les opérateurs unaires

$$E \rightarrow \text{ op } E_2$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

| **Code 3 adresses** |
|---|
| $E_2$.code<br>$E$.place = op $E_2$.place |

### 6.1.3 Les valeurs constantes

$$E \rightarrow \textit{valeur}|\text{id}$$

Seuls les attributs type et valeur de l'expression seront initialisés. Aucun code n'est généré.

### 6.1.4 Les identificateurs

$$E \rightarrow \text{ id}$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

| **Code 3 adresses** |
|---|
| si id fait référence à un argument passé par @<br>$E$.place = * id |

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

| Code 3 adresses |
|---|
| si id fait référence à une variable<br>  $E$.place@ = & id<br>si id fait référence à un argument passé par @<br>  $E$.place@ = id |

### 6.1.5 Les structures

$$E \rightarrow E_1 \text{ . id}$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

| Code 3 adresses |
|---|
| $E_1$.code@<br>temp = $E_1$.place@ + decalage ( id, $E_1$.type )<br>$E$.place = * temp |

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

| Code 3 adresses |
|---|
| $E_1$.code@<br>$E$.place@ = $E_1$.place@ + decalage ( id, $E_1$.type ) |

### 6.1.6 Les tableaux

$$E \rightarrow E_1[E_3]$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

| Code 3 adresses |
|---|
| $E_1$.code@<br>$E_3$.code<br>temp1 = $E_3$.place * taille ( $E_1$.type.typeelt )<br>temp2 = $E_1$.place@ + temp1<br>$E$.place = * temp1 |

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

| Code 3 adresses |
|---|
| $E_1$.code@<br>$E_3$.code<br>temp1 = $E_3$.place * taille ( $E_1$.type.typeelt )<br>$E$.place@ = $E_1$.place@ + temp1 |

### 6.1.7 Les appels de fonction

$$E \rightarrow \text{id} \mid \text{id}\ (E_1, E_2, \cdots, E_n)$$

Après vérification des cas d'erreur, le code 3 adresses généré pour calculer la valeur de E aura la forme suivante

| **Code 3 adresses** |
|---|
| empiler & $E$.place<br>pour chaque arg $E_i$ (i : n -> 1)<br>  passage par valeur<br>  $E_i$.code<br>  empiler $E_i$.place<br>  passage par adresse<br>  $E_i$.code@<br>  empiler $E_i$.place@<br>appeler id |

## 6.2 Les instructions

### 6.2.1 Les affectations

$$I \rightarrow E_1 := E_3$$

Dans le cas général :

| **Code 3 adresses** |
|---|
| $E_1$.code@<br>$E_3$.code<br>* $E_1$.place@ = $E_3$.place |

Cas particulier $E_1$ = id (id de fonction)

| **Code 3 adresses** |
|---|
| $E_3$.code<br>renvoyer $E_3$.place |

Cas particulier $E_1$ = id

| **Code 3 adresses** |
|---|
| $E_3$.code<br>id = $E_3$.place |

### 6.2.2 Les conditionnelles

$$I \rightarrow \text{if } E \text{ then } I_4$$

| **Code 3 adresses** |
|---|
| .siXXX : nop |
| $E$.code |
| si $E$.place aller a .alorsXXX |
| aller a .finsiXXX |
| .alorsXXX : nop |
| $I_4$.code |
| .finsiXXX : nop |

$$I \rightarrow \text{if } E_2 \text{ then } I_4 \text{ else } I_6$$

| **Code 3 adresses** |
|---|
| .siXXX : nop |
| $E$.code |
| si $E$.place aller a .alorsXXX |
| aller a .sinonXXX |
| .alorsXXX : nop |
| $I_4$.code |
| aller a .finsiXXX |
| .sinonXXX : nop |
| $I_6$.code |
| .finsiXXX : nop |

### 6.2.3 Les boucles

$$I \rightarrow \text{while } E \text{ do } I_4$$

| **Code 3 adresses** |
|---|
| .whileXXX : nop |
| $E$.code |
| si $E$.place aller a .bclwhileXXX |
| aller a .finwhileXXX |
| .bclwhileXXX : nop |
| $I_4$.code |
| aller a .whileXXX |
| .finwhileXXX : nop |

$$I \rightarrow \text{repeat } I_2 \text{ until } E$$

| **Code 3 adresses** |
|---|
| .repeatXXX : nop |
| $I_2$.code |
| $E$.code |
| si $E$.place aller a .finrepeatXXX |

*suite sur la prochaine page*

| **Code 3 adresses** |
|---|
| aller a .repeatXXX |
| .finrepeatXXX : nop |

$$I \rightarrow \text{for id} := E_4 \text{ to } E_6 \text{ do } I_8$$

| **Code 3 adresses** |
|---|
| .forXXX : nop |
| $E_4$.code |
| id = $E_4$.place |
| .testforXXX : nop |
| $E_6$.code |
| temp = id > $E_6$.place |
| si temp aller a .finforXXX |
| $I_8$.code |
| id = id + 1 |
| aller a .testforXXX |
| .finforXXX : nop |

$$I \rightarrow \text{for id} := E_4 \text{ downto } E_6 \text{ do } I_8$$

| **Code 3 adresses** |
|---|
| .forXXX : nop |
| $E_4$.code |
| id = $E_4$.place |
| .testforXXX : nop |
| $E_6$.code |
| temp = id < $E_6$.place |
| si temp aller a .finforXXX |
| $I_8$.code |
| id = id - 1 |
| aller a .testforXXX |
| .finforXXX : nop |

### 6.2.4 Les appels de procédure

$$I \rightarrow \text{ id} \mid \text{id } (E_1, E_2, \cdots, E_n)$$

| **Code 3 adresses** |
|---|
| pour chaque arg $E_i$ (i : n -> 1) |
| passage par valeur |
| $E_i$.code |
| empiler $E_i$.place |
| passage par adresse |

*suite sur la prochaine page*

| **Code 3 adresses** |
| --- |
| $E_i$.code@ |
| empiler $E_i$.place@ |
| appeler id |

# *Chapitre 7*

# *Implémentation*

## Sommaire

## 7.1 La table des identificateurs

C'est la table qui associe un numéro unique à chaque identificateur (que ce soit un identificateur du programme source ou un identificateur généré par le compilateur).

Le langage pascal ne différenciant pas les caractères miniscules et majuscules, les chaines définissant les noms des identificateurs seront converties en minuscules.

Le numéro unique sera représenté par un **entier non signé**. La numérotation commencera à **1** (la valeur 0 étant réservée).

Vous devez disposer, au minimum, des fonctions (ou méthodes selon que vous fassiez du C ou du C++) :

```
/*
  ajout d'un identificateur a la table
  renvoie le numero unique associe
  si l'identificateur est deja present, ne fait que renvoyer son numero
*/
unsigned int ajoutIdentificateur ( const char* );
/*
  recupere le nom associe a un numero unique
*/
const char* getNom ( const unsigned int );
/*
  affiche la table sur la sortie standard
*/
void afficherTableIdent ();
/*
  sauvegarde la table dans un fichier
  le nom du fichier est passe en argument
*/
void sauvegarderTableIdent ( const char* );
```

## 7.2 Les tables des symboles

C'est la table qui contient la description de la signification d'un identificateur dans un contexte donné. Les identificateurs sont décrits par leurs numéros uniques.

Les signification possibles sont : programme, fonction, procedure, type, constante, variable, argument, champs, valenum, temporaire, etiquette, chaine.

La signification

— champs fait référence aux champs d'une structure.
— valenum fait référence aux différentes valeurs d'une énumération.
— chaine fait référence aux chaines de caractères statiques définies dans le programme source.

Ainsi la table des symboles est une table associative faisant le lien entre un numéro unique et des données hétérogènes. Pour la mettre en oeuvre, vous avez différentes possibilités dont :

— utiliser l'héritage en C++, avec une classe mère **Symbole**. La table conserve des pointeurs sur cette classe mère mais qui sont initialisés en créant des instances des classes filles.
— utiliser des structures en C avec une structure **Symbole** définie par une union C.

Chaque table des symboles est associée à un contexte (défini par son nom) et connait le contexte parent (pointeur sur la table des symboles du contexte parent).

## 7.3 Le code 3 adresses

Les instructions 3 adresses sera décrite par les structures C suivantes (adaptable en classes si vous préférez faire du C++) :

```
typedef enum {
    OP_NONE,
    OP_CHAR,
    OP_INT,
    OP_BOOL,
    OP_FLOAT,
    OP_IDENT
} TypeOperande;
typedef struct {
    TypeOperande type;
    union {
        char valchar;
        byte valbool;
        short valint;
        float valfloat;
        unsigned int valident;
    } valeur;
} Operande;
typedef struct {
    unsigned int label;
    unsigned int code;
    Operande x, y, z;
} Instruction;
typedef std::vector < Instruction > BlocInstruction;
```