

# Projet compilation et système

## Partie 1 : compilation

Ce projet comporte deux parties. La première partie consiste à construire un compilateur d'un langage de programmation impératif (à la C) dans un langage intermédiaire. La deuxième partie aura pour but d'écrire le simulateur d'un petit système permettant d'interpréter le code généré par cet assembleur dans un contexte multitâches. Via une interface, l'utilisateur du système pourra ainsi exécuter plusieurs processus. Le sujet de cette deuxième partie sera distribué début février 2000. Ce document décrit la première partie à réaliser, le compilateur.

## 1 Travail demandé

La partie « compilation » du projet est à réaliser en C. Elle peut être effectuée par groupes de 6 personnes au maximum. Le travail est à remettre avant le 27 mars 2000. Il consiste en la réalisation d'un *compilateur* d'un langage source dans un langage intermédiaire, et d'un *interpréteur* du langage intermédiaire. Un des module de la partie « systèmes » sera précisément une extension de cet interpréteur. Il n'est pas demandé que l'interpréteur reconnaisse tous les modes d'adressage ni toutes les instructions proposées plus bas, mais il doit évidemment être compatible avec le code produit par le compilateur. Vous pourrez utiliser `lex` et `yacc`.

Des jeux d'essais représentatifs devront être joints à un rapport (remis au plus tard le 27/3/2000). Ce rapport doit contenir une présentation générale du compilateur et de l'interpréteur. Il doit exposer de façon synthétique les choix faits au cours de la réalisation en s'appuyant sur des exemples simples mais significatifs.

Il est conseillé de commencer par un compilateur traduisant un sous-langage du langage source. Néanmoins, il est aussi conseillé de réfléchir dès le départ à l'ensemble des extensions qui seront nécessaires.

## 2 Le langage à compiler

### Éléments lexicaux

Les commentaires sont compris entre `/*` et `*/` et ne s'imbriquent pas. Les blancs sont interdits au milieu d'un mot clé, ignorés ailleurs. Un *identificateur* est une suite de lettres, et une constante entière `nb_entier` est une suite de chiffres. Le lexème virgule `,` joue le rôle de séparateur d'identificateurs. Le point-virgule `;` joue le rôle de terminateur d'instruction. Les opérateurs relationnels sont `==`, `!=`, `<=`, `>=`, `<`, `>`. Les opérateurs logiques sont `||` (ou), `&&` (et), et `!` (non). L'affectation est notée `=`. Les opérateurs arithmétiques sont `+`, `-`, `*`, `%` et `/`. Le moins unaire est aussi noté `-`. Parenthèses `()`, crochets `[]`, accolades `{}` sont des lexèmes utilisés comme en C.

### Syntaxe

La syntaxe du langage est décrite par la grammaire suivante, où les terminaux sont indiqués en *fonte courier* et les non terminaux *<entre crochets>*.

```

<programme>      → <liste de déclarations> <liste de fonctions>
<liste de déclarations> → ε | <déclaration> ; <liste de déclarations>
<déclaration>    → int identificateur | static int identificateur
                  | int identificateur[nb_entier] | static int identificateur[nb_entier]
                  | <prototype>
<prototype>     → <type> identificateur( <suite de types> )
<type>          → void | int
<suite de types> → void | <liste de types>
<liste de types> → int | int, <liste de types>
<liste de fonctions> → <fonction> | <fonction> <liste de fonctions>
<fonction>      → <en-tête> { <corps> }
<en-tête>       → <type> identificateur( <suite de paramètres> )
<suite de paramètres> → void | <liste de paramètres>
<liste de paramètres> → <paramètre> | <paramètre> , <liste de paramètres>

```

<code>&lt;paramètre&gt;</code>	→ <code>int identificateur</code>   <code>int identificateur[nb_entier]</code>
<code>&lt;corps&gt;</code>	→ <code>&lt;liste de déclarations&gt; &lt;liste d'instructions&gt;</code>
<code>&lt;liste d'instructions&gt;</code>	→ <code>ε</code>   <code>&lt;instruction&gt; ; &lt;liste d'instructions&gt;</code>
<code>&lt;instruction&gt;</code>	→ <code>identificateur = &lt;expression&gt;</code>   <code>identificateur[ &lt;expression&gt; ] = &lt;expression&gt;</code>   <code>exit(nb_entier)   return &lt;expression&gt;   return</code>   <code>if( &lt;expression&gt; ) &lt;instruction&gt; else &lt;instruction&gt;</code>   <code>if( &lt;expression&gt; ) &lt;instruction&gt;</code>   <code>while( &lt;expression&gt; ) &lt;instruction&gt;</code>   <code>write_string(chaîne)   write_int( &lt;liste d'arguments&gt; )</code>   <code>read_int identificateur</code>   <code>{ &lt;liste d'instructions&gt; }</code>   <code>&lt;expression&gt;</code>
<code>&lt;expression&gt;</code>	→ <code>&lt;expression&gt; &lt;opérateur binaire&gt; &lt;expression&gt;</code>   <code>- &lt;expression&gt;   ! &lt;expression&gt;   ( &lt;expression&gt; )</code>   <code>identificateur   nb_entier</code>   <code>identificateur( &lt;suite d'arguments&gt; )</code>   <code>identificateur[ &lt;expression&gt; ]</code>
<code>&lt;suite d'arguments&gt;</code>	→ <code>ε</code>   <code>&lt;liste d'arguments&gt;</code>
<code>&lt;liste d'arguments&gt;</code>	→ <code>&lt;expression&gt;   &lt;expression&gt; , &lt;liste d'arguments&gt;</code>
<code>&lt;opérateur binaire&gt;</code>	→ <code>&gt;   &lt;   ==   &lt;=   &gt;=   !=   +   -   *   /   %        &amp;&amp;</code>

## Sémantique

Les seuls types de base sont le type `int` et le type chaîne de caractères. Ce dernier n'est utilisé que sous forme de constante, par la fonction `write_string`. Par exemple: `write_string("Hello world")`. La fonction `write_int` permet d'écrire sur la sortie des entiers: `write_int(1,2*3)` écrit les entiers 1 et 6 sur la sortie. Le seul type nouveau que l'on peut construire est le type tableau d'entiers. Les fonctions peuvent être appelées récursivement, les paramètres étant passés par valeur, sauf pour la fonction `read_int`, qui lit un entier au terminal et affecte la variable dont le nom lui est donné. On remarquera par ailleurs que les tableaux peuvent être passés en argument.

La sémantique des différents opérateurs, ainsi que leur associativité et leur priorité est la même qu'en C. Les variables sont allouées dynamiquement sauf si elles sont déclarées statiques. Comme en C, le point d'entrée du programme est la fonction `main`, de prototype `int main(void)`. Dans la deuxième partie, on ajoutera au langage source des « appels système ».

## 3 Le langage intermédiaire

Le langage intermédiaire est un pseudo-assembleur très simplifié. Il pourra utiliser les registres suivants :

- un compteur de programme contenant le numéro de l'instruction en cours,
- un registre contenant le sommet de la pile,
- un registre qui contient l'adresse de la base de l'enregistrement d'activation courant (*i.e.*, le début du cadre de la fonction courante sur la pile),

### Les instructions et les modes d'adressage

On pourra utiliser les modes d'adressage :

- Immédiat, noté `#a` : `a` est interprété littéralement. Exemple : `#2` représente la constante 2.
- Direct, noté `a` : `a` désigne l'adresse de l'opérande.
- Indirect, noté `@a` : `a` contient l'adresse de l'opérande.
- Indexé, noté `I(a)` c'est-à-dire relatif à un registre particulier, appelé registre d'index.

Le jeu d'instructions n'est donné qu'à titre indicatif. On pourra tout aussi bien utiliser, par exemple, des instructions à zéro adresse calculant directement sur la pile. Leur signification est donnée de façon informelle : on ne tient pas compte des modes d'adressage employés dans la colonne « Opération réalisée » ; `a`, `b` et `c` sont le plus souvent interprétés comme s'ils étaient donnés en mode direct. La signification réelle doit bien entendu tenir

compte des modes d'adressage. Certains modes sont évidemment interdits; par exemple, une destination n'est jamais donnée en adressage immédiat.

Nom	Arguments	Opération réalisée
read	a	Lit une valeur sur l'entrée standard et la charge dans a
write	a	Écrit a sur la sortie standard
write	s	Écrit la chaîne de caractères s sur la sortie standard
mov	a b	Transfère a dans b
push	a	Empile a
pop	a	Dépile a
label	a	Positionne l'étiquette a
add	a b c	c reçoit a+b
sub	a b c	c reçoit a-b
mult	a b c	c reçoit a*b
div	a b c	c reçoit le quotient de la division entière de a par b
and	a b c	c reçoit le « et » logique de a et b
or	a b c	c reçoit le « ou » logique de a et b
xor	a b c	c reçoit le « ou exclusif » de a et b
not	a b	b reçoit le « non » logique de a
neg	a b c	c reçoit 1 ou 0 suivant que a est inférieur ou non à b
zero	a b	b reçoit 1 ou 0 suivant que a est nul ou non
jmp	a	Branchement à l'adresse (étiquette) spécifiée par a
jpos	a b	Branchement à l'adresse spécifiée par a si b est positif
call	a	Appelle la fonction désignée par a
return		Retour d'appel
halt		Arrêt
nop		Ne fait rien