

# Projet de Compilation

## Maîtrise d'informatique 96–97

### Université Paris 7

## 1 Objet du projet. Modalités de réalisation

Le but de ce projet est d'écrire un générateur d'analyseur syntaxique basé sur l'analyse lexicale LL(1), puis d'utiliser cet outil pour compiler un langage. Il pourra être écrit en langage C, C++ ou CAML. Le choix du langage n'entrera pas en compte dans l'évaluation du projet. On pourra utiliser `lex` pour le travail des sections 2 et 3. Le projet est à réaliser par groupes de quatre personnes au maximum. Chaque membre d'un groupe doit réaliser une partie non triviale du travail demandé qu'il devra exposer lors de la soutenance. Par ailleurs, chacun est censé connaître les options choisies par le groupe ainsi que les difficultés rencontrées. Les sections suivantes décrivent le travail à réaliser. Le projet sera présenté dans un rapport synthétique illustré par des exemples. Ce rapport insistera sur les problèmes rencontrés et les choix effectués pour les résoudre.

## 2 Générateur d'analyseur syntaxique

On demande tout d'abord la réalisation d'un générateur d'analyseur syntaxique. Avant de passer à la section suivante, on testera le générateur sur de petits jeux d'essais.

Le générateur lit une spécification de grammaire dans le fichier qui lui est donné en argument et génère un analyseur LL(1) pour cette grammaire. La spécification de la forme du fichier décrivant la grammaire est à définir par chaque groupe. Il est possible par exemple de reprendre une spécification compatible avec celle de `yacc`. Ainsi, on pourra autoriser des blocs littéraux que le générateur se contentera de copier dans le code de l'analyseur. La forme de chaque règle de grammaire est imposée. Elle devra être du même type que celle employée par `yacc` :

```
non-terminal:  membre droit de la règle 1
              | membre droit de la règle 2
              ...
              ;
```

chaque règle étant alors considérée comme distincte. Un membre droit peut par ailleurs être vide.

Le générateur d'analyseur syntaxique doit fournir :

- La liste des ensembles **Premier**(X) et **Suivant**(X) pour chaque non-terminal X.
- La table d'analyse LL(1), consultable sous forme texte.
- Le code d'un analyseur syntaxique prenant en entrée une chaîne de terminaux, construisant l'arbre de dérivation associé et indiquant les éventuelles erreurs. Si la grammaire n'est pas LL(1), l'analyseur doit l'indiquer. Si cela a un sens, on choisira alors une façon de lever chacun des conflits.

## 3 Un compilateur utilisant le générateur d'analyseur syntaxique

Le but de cette partie est de construire un compilateur d'un petit langage de programmation évolué dans un langage intermédiaire. Le générateur d'analyseur syntaxique de la section 2 devra être utilisé pour construire le compilateur. Pour vérifier le fonctionnement du compilateur, on demande l'écriture d'un interpréteur de l'assembleur qui simulera l'exécution. À nouveau, cette partie devra fournir un jeu de tests convaincant.

### 3.1 Le langage intermédiaire

Le langage intermédiaire engendré par le compilateur sera interprété. Il correspond au code d'une machine virtuelle. La mémoire de cette machine pourra contenir une zone statique, une pile et une zone de texte contenant les instructions du programme à exécuter. La machine dispose aussi de registres. Typiquement, on pourra utiliser un compteur qui pointe sur l'instruction en cours, un registre de sommet de pile, et, si nécessaire (cf. Section 4), un registre de base contenant l'adresse de la base de l'enregistrement d'activation courant.

Les modes d'adressage ne sont pas imposés. On peut utiliser le mode immédiat, direct, indirect ou indexé et combiner raisonnablement ces modes. Tout mode généré par le compilateur devra évidemment être interprétable. Les instructions sont données dans le tableau qui suit. Ce sont des instructions à une, deux ou trois adresses. La signification est donnée de façon informelle et peut dans certains cas être comprise de plusieurs façons. Dans la colonne « Opération réalisée », on ne tient pas compte des modes d'adressage employés. 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. Il est possible de n'utiliser qu'un sous-ensemble des instructions proposées. Les accents ne sont mis ici que pour la lisibilité.

<i>Nom</i>	<i>Arguments</i>	<i>Opération réalisée</i>
Lire		Lit une valeur sur l'entrée standard et la charge dans A
Écrire	A	Écrit A sur la sortie standard
Transférer	A B	Transfère A dans B
Empiler	A	Empile A
Dépiler	A	Dépile A
Étiquette	A	Positionne l'étiquette A
Incrémenter	A	Incrémente A
Décrémenter	A	Décrémente A
Additionner	A B C	C reçoit A+B
Soustraire	A B C	C reçoit A-B
Multiplier	A B C	C reçoit A*B
Diviser	A B C	C reçoit le quotient de la division entière de A par B
PrendreLeReste	A B C	C reçoit le reste de la division entière de A par B
ChangerDeSigne	A B	B reçoit l'opposé de A
Et	A B C	C reçoit le «et» logique, bit à bit, de A et B
Ou	A B C	C reçoit le «ou» logique, bit à bit, de A et B
Non	A B	B reçoit le «non» logique, bit à bit, de A
TestSiInférieur	A B	C reçoit 1 ou 0 suivant que A est inférieur ou non à B
TestSiNul	A B	B reçoit 1 ou 0 suivant que A est nul ou non
Branchement	A	Branchement inconditionnel à l'adresse spécifiée par A
BranchementSiVrai	A B	Branchement à l'adresse (ou à l'étiquette) spécifiée par A si le booléen désigné par B est vrai
Appeler	A	Appelle la fonction désignée par A
Retourner		Retour d'appel
Arrêter		Arrêt
PasDOpération		Ne fait rien

### 3.2 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 () et crochets [] 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 courrier** et les non terminaux *<entre crochets>*.

<i>&lt;programme&gt;</i>	→	<b>début</b> <i>&lt;liste de déclarations&gt;</i> <i>&lt;liste de fonctions&gt;</i> <i>&lt;liste d'instructions&gt;</i> <b>fin</b>
<i>&lt;liste de déclarations&gt;</i>	→	$\varepsilon$   <i>&lt;déclaration&gt;</i> ; <i>&lt;liste de déclarations&gt;</i>
<i>&lt;déclaration&gt;</i>	→	<b>entier</b> <i>identificateur</i>   <b>entier</b> <b>statique</b> <i>identificateur</i>   <b>tableau</b> <b>entier</b> <i>identificateur</i> [ <b>nb_entier</b> ]   <b>tableau</b> <b>statique</b> <b>entier</b> <i>identificateur</i> [ <b>nb_entier</b> ]
<i>&lt;liste de fonctions&gt;</i>	→	$\varepsilon$   <i>&lt;fonction&gt;</i> ; <i>&lt;liste de fonctions&gt;</i>
<i>&lt;fonction&gt;</i>	→	<i>&lt;en-tête&gt;</i> <i>&lt;corps&gt;</i>
<i>&lt;en-tête&gt;</i>	→	<b>fonction</b> <i>&lt;type&gt;</i> <i>identificateur</i> ( <i>&lt;suite de paramètres&gt;</i> )
<i>&lt;type&gt;</i>	→	<b>vide</b>   <b>entier</b>
<i>&lt;suite de paramètres&gt;</i>	→	<b>vide</b>   <i>&lt;liste de paramètres&gt;</i>
<i>&lt;liste de paramètres&gt;</i>	→	<i>&lt;paramètre&gt;</i>   <i>&lt;paramètre&gt;</i> , <i>&lt;liste de paramètres&gt;</i>
<i>&lt;paramètre&gt;</i>	→	<b>entier</b> <i>identificateur</i>   <b>tableau</b> <b>entier</b> <i>identificateur</i> [ <b>nb_entier</b> ]
<i>&lt;corps&gt;</i>	→	<b>début</b> <i>&lt;liste de déclarations&gt;</i> <i>&lt;liste d'instructions&gt;</i> <b>fin</b>
<i>&lt;liste d'instructions&gt;</i>	→	$\varepsilon$   <i>&lt;instruction&gt;</i> ; <i>&lt;liste d'instructions&gt;</i>
<i>&lt;instruction&gt;</i>	→	<b>début</b> <i>&lt;liste d'instructions&gt;</i> <b>fin</b>   <i>identificateur</i> = <i>&lt;expression&gt;</i>   <i>identificateur</i> [ <i>&lt;expression simple&gt;</i> ] = <i>&lt;expression&gt;</i>   <b>retourner</b> <i>&lt;expression&gt;</i>   <b>arrêt</b>   <b>retour</b>   <b>si</b> <i>&lt;expression&gt;</i> <b>alors</b> <i>&lt;instruction&gt;</i> <b>sinon</b> <i>&lt;instruction&gt;</i>   <b>si</b> <i>&lt;expression&gt;</i> <b>alors</b> <i>&lt;instruction&gt;</i>   <b>tant que</b> <i>&lt;expression&gt;</i> <b>faire</b> <i>&lt;instruction&gt;</i>   <b>écrire</b> <i>&lt;liste d'arguments&gt;</i>   <b>lire</b> <i>identificateur</i>   <i>&lt;expression&gt;</i>
<i>&lt;expression&gt;</i>	→	<i>&lt;expression simple&gt;</i> <i>&lt;comparaison&gt;</i> <i>&lt;expression simple&gt;</i>   <i>&lt;expression simple&gt;</i>
<i>&lt;expression simple&gt;</i>	→	<i>&lt;expression simple&gt;</i> + <i>&lt;terme&gt;</i>   <i>&lt;expression simple&gt;</i> - <i>&lt;terme&gt;</i>   <i>&lt;expression simple&gt;</i>    <i>&lt;terme&gt;</i>   <i>&lt;terme&gt;</i>   - <i>&lt;terme&gt;</i>
<i>&lt;terme&gt;</i>	→	<i>&lt;terme&gt;</i> * <i>&lt;facteur&gt;</i>   <i>&lt;terme&gt;</i> / <i>&lt;facteur&gt;</i>   <i>&lt;terme&gt;</i> % <i>&lt;facteur&gt;</i>   <i>&lt;terme&gt;</i> && <i>&lt;facteur&gt;</i>   <i>&lt;facteur&gt;</i>
<i>&lt;facteur&gt;</i>	→	<i>identificateur</i>   <b>nb_entier</b>   <i>identificateur</i> ( <i>&lt;suite d'arguments&gt;</i> )   ( <i>&lt;expression&gt;</i> )   <i>identificateur</i> [ <i>&lt;expression simple&gt;</i> ]   ! <i>&lt;facteur&gt;</i>
<i>&lt;suite d'arguments&gt;</i>	→	$\varepsilon$   <i>&lt;liste d'arguments&gt;</i>
<i>&lt;liste d'arguments&gt;</i>	→	<i>&lt;expression&gt;</i>   <i>&lt;expression&gt;</i> , <i>&lt;liste d'arguments&gt;</i>
<i>&lt;comparaison&gt;</i>	→	<   >   ==   <=   >=   !=

Il est permis de modifier la grammaire mais pas de changer le langage engendré.

## Sémantique

Le seul type de base est le type entier et le seul type nouveau que l'on peut construire est le type tableau d'entiers. Il n'y a pas de procédures, mais il y a des fonctions, qui peuvent être appelées récursivement, les paramètres étant passés par valeur. La sémantique des opérateurs et leur priorité est la même qu'en C. Les variables sont allouées dynamiquement sauf si elles sont déclarées statiques.

## 4 Extensions

Une fois le travail précédent réalisé, on ajoutera une ou plusieurs extensions au langage de la section 3.2. Ces extensions ne sont pas imposées, chaque groupe est libre d'imaginer ce qu'il juge intéressant. On donne ici quelques possibilités.

- Reprise de l'analyseur syntaxique sur erreurs.
- Passage de paramètres par «variables». Manipulation d'adresses, gestion d'un tas.
- Imbrication de fonctions à la PASCAL. Gestion de portée statique/dynamique des identificateurs.
- Ajout de nouveaux types de données. Contrôle de types. Exemple : ajout d'un type *graphique* permettant de manipuler des images. On pourra dans ce cas envisager des fonctions fournies par le langage de la section 3.2 permettant de définir des constantes de ce type de donnée (comme segment de droite ou cercle), puis d'afficher des variables de ce type. On pourra également enrichir le langage intermédiaire d'instructions de base, comme **Segment** ou **Cercle**.