Thème 1
Analyse Lexicale, Analyse Syntaxique

1.1 Un peu de cours

1.1.1 Analyse Lexicale avec flex

Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal peut être par exemple un nombre, un signe '+', un identificateur, etc...). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique (voir ci-dessous). Le but de l'analyseur lexical est donc de "consommer" des symboles et de les fournir à l'analyseur syntaxique. Un fichier de description pour Lex est formé de trois parties, selon le schéma suivant :

```
declarations
%%
définition des symboles
%%
code additionnel
```

dans lequel aucune partie n'est obligatoire. Cependant, le premier %% l'est, afin d'indiquer la séparation entre les déclarations et les productions.

Les déclarations Cette partie d'un fichier Lex peut contenir :

— Du code écrit dans le langage cible, encadré par %{ et %}, qui se retrouvera au début du fichier synthétisé par Lex. C'est ici que l'on va spécifier les fichiers à inclure. Lex recopie tel quel tout ce qui est écrit entre ces deux signes, qui devront toujours être placés en début de ligne.

— Des expressions régulières (dont on donne une partie de la syntaxe abondante dans la Figure 1.1) définissant des notions non terminales, telles que lettre, chiffre et nombre. Ces spécifications sont de la forme :

```
notion      expression_reguliere
```

Exemples :

```
blancs              [\t\n ]+
lettre              [A-Za-z]
chiffre10           [0-9]
chiffre16           [0-9A-Fa-f]
entier10            {chiffre10}+
```

Les définitions des symboles Cette partie sert à indiquer à Lex ce qu'il devra faire lorsqu'il rencontrera tel ou tel symbole. Celle-ci peut contenir :

— Des spécifications écrites dans le langage cible, encadrées par %{ et %} (toujours placés en début de ligne), qui seront placées au début de la fonction yylex(), la fonction chargée de consommer les terminaux, et qui renvoie un entier.

— Des productions de la forme :

```
expression_reguliere        action
```

Si l'action est absente, Lex recopiera les caractères tels quels sur la sortie standard. Si l'action est présente, elle sera écrite en code du langage cible. Exemple, les deux règles suivantes :

```
[ \t]+$             ;
[ \t]               printf(" ");
```

Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf le retour à la ligne
$[xyz]$	Soit x, soit y, soit z
$[\wedge bz]$	Tous les caracteres, SAUF b et z
$[a-z]$	N'importe quel caractere entre a et z
$[\wedge a-z]$	Tous les caracteres, SAUF ceux compris entre a et z
$R*$	Zero R ou plus, ou R est n'importe quelle expression reguliere
$R+$	Un R ou plus
$R?$	Zero ou un R (c'est-a-dire un R optionnel)
$R\{2,5\}$	Entre deux et cinq R
$R\{2,\}$	Deux R ou plus
$R\{2\}$	Exactement deux R
$"[xyz\backslash"foo"$	La chaine '$[xyz\backslash"foo$'
$\{NOTION\}$	L'expansion de la notion NOTION definie plus haut
$\backslash X$	Si X est un 'a', 'b', 'f', 'n', 'r', 't'.
$\backslash 0$	Caractere ASCII 0
$\backslash 123$	Caractere ASCII dont le numero est 123 EN OCTAL
$\backslash x2A$	Caractere ASCII en hexadecimal
RS	R suivi de S
$R\&S$	R ou S
R/S	R, seulement s'il est suivi par S
^{R}	R, mais seulement en debut de ligne
$R\$$	R, mais seulement en fin de ligne
$<<EOF>>$	Fin de fichier

FIGURE 1.1 – Expressions régulières en Flex, syntaxe et sémantique

permettent de supprimer tous les espaces inutiles dans un fichier. Si l'action comporte plus d'une seule instruction ou ne peut tenir sur une seule ligne, elle devra être parenthésée par `{` et `}`. Il faut de plus savoir que les commentaires tels que `/* ... */` ne peuvent être présents dans la deuxième partie d'un fichier Lex que s'il sont placés dans les actions parenthésées. Dans le cas contraire, ceux-ci seraient considérés par Lex comme des expressions régulières ou des actions, ce qui donnerait lieu à des messages d'erreur, ou, au mieux, à un comportement inattendu.

Afin de pouvoir utiliser dans les actions la valeur de l'expression reconnue, Lex fournit une variable nommée `yytext` qui désigne dans les actions les caractères acceptés par l'expression régulière à gauche de la règle. Il s'agit d'un tableau de caractères de longueur `yyleng` (donc défini comme `char yytext[yyleng]`).

Le code additionnel Cette section du fichier flex contient les implémentations C des fonctions nécessaires. Si aucune fonction `main` n'est écrite (et que le fichier n'est pas lié à un fichier bison qui y fait référence), le code suivant sera automatiquement généré lors de la compilation :

```
int main() {
  yylex();
}
```

La compilation La génération de l'exécutable se fera à l'aide des commandes suivantes :

```
flex -o lexer.c lexer.l
gcc -o lexer.o -c lexer.c
gcc -o main lexer.o -lfl
```

1.1.2 Analyse syntaxique avec bison

L'outil bison permet de générer des analyseurs syntaxiques. Un fichier d'entrée décrit la grammaire à analyser ainsi que les attributs et actions sémantiques associés. Un fichier `.c` est généré à partir de cette description, celui-ci contient la mise en oeuvre de l'automate à pile pour une analyse ascendante. L'appel de l'analyseur se fait par le biais de la fonction `int yyparse ()` créée par bison. Les fichiers de description de grammaire pour bison ont un formalisme similaire à ceux de Flex :

```
Définitions
%%
```

```
Règles de production
%%
Code C/C++
```

Les définitions Cette section permet de décrire certaines parties de la grammaire (ensemble des symboles terminaux, axiomes, attributs sémantiques...). Il est possible d'y inclure du code C en l'encadrant avec `%{` et `%}`. Le mot-clé `%start` permet de définir l'axiome de la grammaire. Le mot-clé `%token` permet de définir les éléments du vocabulaire terminal. Les mot-clés `%left`,`%right` et `%nonassoc` permettent de définir la priorité (ordre de spécification des éléments) et l'associativité des opérateurs.

Les règles de production Les règles de production sont données sous la forme

```
symbole : règle [action];
```

L'action est ici encore optionnelle. Dans le cas où un symbole accèpte plusieurs dérivations possibles, on utilisera le caractère '|' pour différencier les règles. Exemples :

```
%%
A : a b c ... z
;
%%

%%
A : a b c
  | d e f
;
%%
```

EXEMPLE 1. Soit la grammaire des expressions arithmétiques (addition et multiplication) entières :

```
E -> E + E | E * E | cste
```

Elle sera décrite par le fichier Bison suivant :

```
%start E
%token +
%token *
%token cste
%%
E : E + E
  | E * E
  | cste
;
%%
int main ( int argc, char** argv ) {
  yyparse ();
}
```

Le code C La dernière section du fichier de description contient du code C (fonctions, variables globales...). Si aucune fonction `main` n'est écrite, le code suivant sera automatiquement généré lors de la compilation :

```
main() {
  yyparse();
}
```

1.1.3 Faisons communiquer Flex et Bison

L'analyseur syntaxique fait appel à l'analyseur lexical pour connaître le prochain symbole dans la chaîne à analyser. Pour Bison l'appel à l'analyseur lexical se fait par le biais de la fonction `int yylex ()`. Dans la majeure partie des cas, cet analyseur lexical sera généré à l'aide de Flex. La Figure 1.2 illustre la communication et la chaîne de compilation associée. La génération de l'exécutable se fera à l'aide des commandes suivantes :

```
flex -o lexer.c lexer.l
bison -d -o parser.c parser.y
gcc -o parser.o -c parser.c
gcc -o lexer.o -c lexer.c
gcc -o main parser.o lexer.o -lfl
```

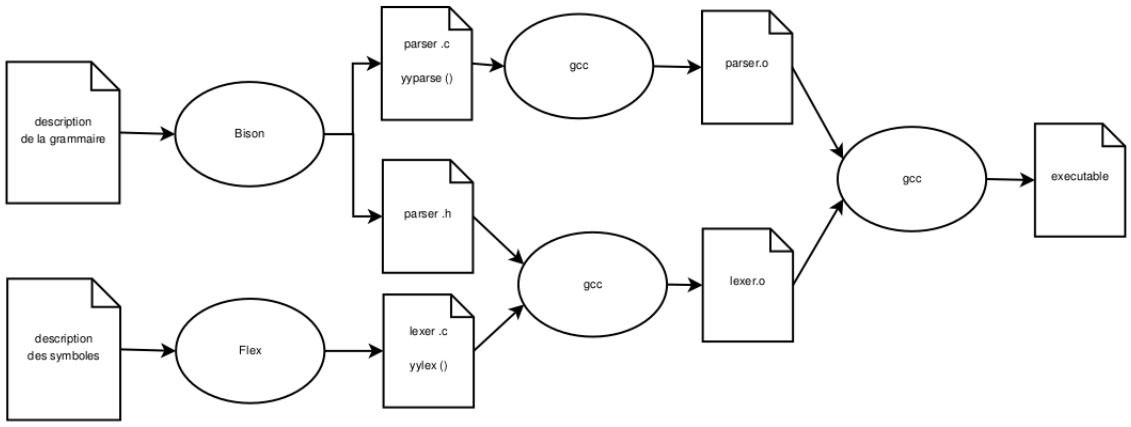

FIGURE 1.2 – Chaîne de compilation Flex/Bison

L'option `-d` de bison permet de dire à ce dernier de générer un fichier `.h` (ici `parser.h`) contenant la définition des constantes associées aux différents symboles terminaux de la grammaire. Ce fichier est à inclure (`#include<parser.h>`) dans le fichier `lexer.l` pour que l'analyseur lexical puisse renvoyer la valeur de ces constantes en résultat. Dans les actions de l'analyseur lexical, on aura un return «symbole» où «symbole» est l'une des valeurs définies dans les `%token`.

EXEMPLE 2 (Le classique $a^n b^n$, en flex+bison). Le fichier Flex retourne des tokens :

```
%{
#include "parser.h"
int yyerror (char*); %}
%%
a       return TOKEN_A;
b       return TOKEN_B;
(.|\n)  yyerror ( "symbole non reconnu" );
%%
int yyerror ( char* m ) {
  printf ( "%s\n", m );
  return 1;
}
```

Ces tokens sont utilisés dans le fichier Bison (remarquez la déclaration) :

```
%token TOKEN_A
%token TOKEN_B
%start E
%%
E : TOKEN_A E TOKEN_B
  | TOKEN_A TOKEN_B;
%%
int main ( int argc, char** argv ) {
  yyparse ();
}
```

1.2 Exercices

EXERCICE 1.1 ▸ **Reconnaissance de langages**
On utilisera dans cet exercice la syntaxe flex pour définir des macros Flex.

1. Écrire une définition des identificateurs (lettres, chiffres et _, ne commençant ni par un chiffre ni par_).
2. Écrire une définition d'un réel (parties entière et fractionnaire obligatoires, signe optionnel, sans exposant)
3. Écrire une définition d'un réel (partie entière obligatoire, partie flottante optionnelle, signe optionnel, avec/sans exposant) chiffre.

EXERCICE 1.2 ▸ **Mystère**
On fournit le fichier `flex` suivant :

```
%%
[a-z]   printf ( "%c", yytext [ 0 ] - 'a' + 'A' );
\n      printf ( "%s", yytext );
.       printf ( "%s", yytext );
%%
int main ( int argc, char** argv )
{
   yylex ();
}
```

Qu'elle est la fonctionnalité de ce fichier `flex` après compilation ?

EXERCICE 1.3 ▸ **Ordre d'évaluation des règles**

On fournit les fichiers suivants :

Flex :

```
%%
"("  return PO;
")"  return PF;
"["  return CO;
"]"  return CF;
\n
.
%%
```

Bison

```
F : L           printf("F->L\n");
L : LI          printf("L->LI\n") ;
|               printf("L->eps\n");
LI : LI I       printf("LI->LI I\n");
| I             printf("LI->I\n");
I : PO L PF     printf("I->(L)\n");
| CO L CF       printf("I->[L]\n");
%%
```

1. Que fait le fichier `flex` ?
2. Donner l'affichage obtenu lors de l'analyse de la chaîne `(3+a)*b[8-c]` en précisant l'arbre de dérivation.

EXERCICE 1.4 ▸ $a^n b^n$ **avec des compteurs**
Ecrire un couple de fichiers flex/bison qui permettent de reconnaître la grammaire $a^+ b^+$. Utiliser des variables compteurs dans le fichier Bison pour reconnaître uniquement $a^n b^n$.