Université d'Orléans **Maîtrise d'Informatique**

Feuille de Travaux Dirigés 0

Prise en main de **Lex** et **Yacc**

---

1 Petit manuel de lex et yacc

Copiez chez vous les fichiers `lexgram.l` et `gram.y` se trouvant dans le répertoire

```
/home/info/ens/todinca/Compilation/lex-yacc/V0
```

Pour compiler, il suffit de faire :

```
lex lexgram.l
yacc gram.y
gcc y.tab.c
```

ce qui aura comme effet de créer un fichier exécutable `a.out`.

1.1 Structure des fichiers lex

Lex est un outil standard d'analyse lexicale. Un fichier lex est de la forme :

```
déclarations
%%
productions
%%
code additionnel
```

- La partie `déclarations` contient des expressions régulières définissant des notions non terminales, telles que lettre, chiffre et nombre. Ces spécifications sont de la forme : `notion expression_régulière` (cf. annexe A). Elle peut contenir aussi du code en langage cible (en occurrence le C) encadré entre %{ et %}, par exemple :

  ```
  %{
  #include <stdio.h>
  %}
  ```
- Les `productions` ont la forme : `expression_régulière action`. Lorsque `action` est absente, **Lex** recopiera les caractères tels quels sur la sortie standard. Si `action` est présente, elle devra être écrite en langage cible. Typiquement, une action retourne comme résultat un symbole terminal de la grammaire (nombre, opérateur, mot clef). Remarquez l'existence des variables `yytext`, qui contient le texte correspondant à l'unité lexicale, et `yylval`, qui permet d'associer une valeur (un attribut) à cette unité lexicale (voir aussi la description de **yacc**).
- On peut enfin utiliser la dernière partie pour écrire du code en langage cible.

1.2 Structure des fichiers yacc

Les fichiers yacc ont une structure similaire aux fichiers lex :

```
déclarations
%%
productions
%%
code additionnel
```

- La partie `déclarations` est formée de :
  - Spécifications écrites dans le langage cible, placées entre %{ et %}, ces deux symboles étant obligatoirement en début de ligne.
  - Les déclarations des terminaux pouvant être rencontrés, à l'aide du mot-clé %`token`
  - Le type de donnée du terminal courant, avec le mot-clé %`union`
  - Des informations donnant la priorité et l'associativité des opérateurs.
  - L'axiome de la grammaire, avec le mot-clé %`start` (si celui-ci n'est pas précisé, l'axiome de la grammaire est la première production de la deuxième partie).
- La zone des `productions`, qui ne peut pas être vide, contient des productions de la grammaire du langage choisi et éventuellement des déclarations et/ou définitions encadrées par %{ et %}.
  Observez l'utilisation des attributs de la grammaire à l'aide des opérateurs \$\$ et $\$n$.
- Le `code additionnel` devra obligatoirement comporter une déclaration du `main()` (qui devra appeler la fonction `yyparse()`), et de la fonction `yyerror(char *message)`, appelée lorsqu'une erreur de syntaxe est trouvée.

2 Votre travail

2.1 Expressions arithmétiques

Enrichissez la grammaire pour que l'on puisse utiliser les expressions arithmétiques habituelles (opérateurs +,-,*,/ et parenthèses). Que pensez vous de l'associativité de l'opérateur "puissance" ?

2.2 Variables

L'objectif est de compiler un langage qui accepte en plus
- des déclarations de variables, de la forme : `decl variable1 ;`
- des affectations : `variable1 = 2+3 ; variable1 = variable1*5 ;`
- des instructions d'affichage : `print variable1+3 ;`

2.2.1 Un peu de syntaxe, très peu se sémantique

Définissez un jeton `VARIABLE` et une production (au niveau de l'analyse lexicale) qui identifiera les variables. Cette production devra associer à `yylval` le nom de la variable, donc `yylval` prendra tantôt des valeurs entières (lorsqu'il s'agit d'un nombre), tantôt des valeurs chaîne de caractères (lorsqu'il s'agit d'une variable).

La modification du type `YYSTYPE` de `yylval` se fait en rajoutant les lignes suivantes dans les déclarations du fichier yacc :

```
/* YYSTYPE, le type de yylval */
%union
{
      int nombre;
      char variable[LG_MAX_ID];
}
```

Pour indiquer qu'au symboles terminaux de type NOMBRE correspond l'attribut `yylval.nombre` et aux symboles VARIABLE correspond l'attribut `yylval.variable`, la déclaration des deux jetons prendra la forme :

```
%token <nombre> NOMBRE
%token <variable> VARIABLE
```

L'attribut des non-terminaux `expr` sera aussi `yylval.nombre` :

```
%type <nombre> expr
```

Pour que l'évaluation des expressions continue de fonctionner, faites comme si chaque variable valait 0.

2.2.2 Table des symboles

Implémentez une table de symboles simple (un tableau dont chaque élément est formé d'un nom de variable de d'une valeur). Vous pouvez définir les fonctions suivantes :

```
/* prototypes */
typedef int POS;
typedef int err;

void yyerror(char*); /* traitement des ereurs */

err init_TDS();
err creer_id(char*);
POS chercher_id(char* nom); /* renvoie la position de l'ID
      dans la TDS ou -1 si l'ID n'existe pas*/
int valeur_id(char*); /* renvoie la valeur de l'ID;
      provoque une erreur si l'ID n'existe pas */
err maj_id(char*, int); /* mise a jour de la valeur de l'ID,
      provoque une erreur si l'ID n'existe pas */
void affiche_TDS(void);
```

On décide qu'une variable déclarée est initialisée à 0. Vous pouvez faire fonctionner votre calculette !

A Expressions regulières en lex

Symbole	Signification
x	le caractère 'x'
.	n'importe quel caractère sauf \n
[xyz]	soit x, soit y, soit z
[^bz]	tous les caractères, sauf b et z
[a-z]	n'importe quel caractère entre a et z
[^a-z]	tous les caractères, sauf ceux compris entre a et z
R*	zéro R ou plus, où R est n'importe quelle expression régulière
R+	un R ou plus
R ?	zéro ou un R (c'est-à-dire un R optionnel)
R{2,5}	entre 2 et 5 R
R{2,}	2 R ou plus
R{2}	exactement 2 R
"[xyz\"foo"	la chaîne "[xyz"foo"
{NOTION}	l'expansion de la notion NOTION définie plus haut
\X	si X est un "a", "b", "f", "n", "r", "t", ou "v", représente l'interprétation ANSI-C de \X.
\0	le caractère ASCII 0
\123	le caractère dont le code ASCII est 123, en octal
\x2A	le caractère dont le code ASCII est 2A, en hexadécimal
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 début de ligne
R$	R, mais seulement en fin de ligne
«EOF»	fin de fichier

Ainsi, la définition

```
lettre          [a-zA-Z]
chiffre         [0-9]
identificateur  {lettre}(_|{lettre}|{chiffre})*
```

reconnaîtra comme identificateur les mots "intEGER", "une_variable", "a1", mais pas "_ident" ni "1variable".