MODULE : Techniques de Compilation

DEPARTEMENT : GLSI - ISI

NIVEAU : 1ère Ingénieur

TP1 - Analyse lexicale (Outil Flex)

Rappel

La tâche principale d'un analyseur lexical est de lire un texte source (suite de caractères) et de produire comme résultat une suite d'unités lexicales. La reconnaissance des unités lexicales est basée sur la notion d'expressions régulières. Théoriquement, la construction d'un analyseur lexical consiste à :

- Définir les unités lexicales.
- Modéliser chaque unité lexicale par une expression régulière.
- Représenter chaque expression régulière par un diagramme de transition (automate).
- Construire le diagramme global.
- Implémenter à la main le diagramme obtenu.

Généralement, l'implémentation à la main d'un diagramme avec un grand nombre d'états est une tâche qui n'est pas assez facile. En outre, si on a besoin d'ajouter ou modifier une unité lexicale, il faut parcourir tout le programme pour effectuer les modifications nécessaires. Plusieurs outils ont été bâtis pour simplifier cette tâche. (Flex par exemple)

Flex :

L'outil Flex (version GNU de LEX) est un générateur d'analyseurs lexicaux. Il accepte en entrée des unités lexicales sous formes d'expressions régulières et produit un programme écrit en langage C qui, une fois compilé, reconnaît ces unités lexicales. L'exécutable obtenu lit le texte d'entrée caractère par caractère jusqu'à l'identification de plus long préfixe du texte source qui concorde avec l'une des expressions régulières. Un fichier de spécifications Flex se compose de quatre parties :

%{
Les déclarations en c
%}
Déclaration des définitions régulières
%%
Règles de traduction
%%
Bloc principal et fonctions auxiliaires en C

Définitions régulières :

Une définition régulière permet d'associer un nom à une expression régulière et de se référer par la suite dans la section de règles à ce nom, plutôt qu'à l'expression régulière.

Règles de traduction :

*exp*1 **{** *action*1**}**
exp2 **{** *action*2**}**
… …
expn **{** *actionn***}**

Chaque *expi* est une expression régulière qui modélise une unité lexicale. Chaque *actioni* est une suite d'instructions en C.

Exercice 1 – premiers pas en Flex fait

Partie 1

1) Ecrire le programme Flex suivant permettant de dire si une chaîne en entrée est un nombre binaire ou non. Le fichier doit être enregistré avec l'extension **.l** (exemple : *binaire.l)*

```
%%
(0|1)+   printf  ("c'est un nombre binaire");
. *   printf ("ce n'est pas un nombre binaire");
%%
int  yywrap(){return 1;}
main ()
{
yylex ();
}
```

2)Créer un nouveau répertoire et y placer le fichier binaire.l.

3)Lancer l'invite de commande propre a Flex Windows et compiler votre fichier « binaire.l » à l'aide de la commande**: flex binaire.l.** S'il y aura pas de problème, un fichier lex.yy sera créé.

Ensuite on va compiler le fichier lex.yy.c et générer un fichier C .exe à l'aide de la commande : **gcc lex.yy.c -o prog**

Enfin, vous pouver appeler le programme prog.exe à travers la commande **prog.exe**

4) Créer un fichier texte dans votre répertoire de travail et y éditer quelques lignes.

Rajouter le code suivant dans le main() de votre programme.

```
int main(int argc, char *argv[])
{
        ….
        ++argv, --argc;
        if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
        else
        yyin = stdin;
        yylex() ;
}
```

Modifier votre programme pour rediriger la sortie standard vers un fichiers **resultat.txt.**

Partie 2

1. Ecrire et compiler le fichier de spécifications suivant :

```
pairpair (aa|bb)*((ab|ba) (aa|bb)*(ab|ba) (aa|bb)*)*
%%
{pairpair} printf ("[%s]: nombre pair de a et de b\n", yytext);
a*b* printf ("[%s]: des a d'abord et des b ensuite\n", yytext);
.
%%
int yywrap() {return 1;}
main()
{
yylex();
}
```

2. Tester les entrées babbaaab abbb aabb baabbbb bbaabbba baabbbbab aaabbbba.

3. Même question en permutant les deux lignes :

```
a*b* printf("[%s]: des a d'abord et des b ensuite \n",yytext);
(pairpair) printf ("[%s]: nombre pair de a et de b \n",yytext);
```

4. Y'a t-il une différence ? Laquelle ? Pourquoi ?
5. On considère l'unité lexicale **id** définie comme suit : un identificateur est une séquence de lettres et de chiffres. Le premier caractère doit être une lettre. Ecrire à l'aide de Flex un analyseur lexical qui permet de reconnaître à partir d'une chaîne d'entrée l'unité lexicale **id.**

6. Modifier l'exercice précédent pour que l'analyseur lexical reconnaisse les deux unités lexicales **id** et **nb** sachant que **nb** est une unité lexicale qui désigne les entiers naturels.

Exercice 2 fait

Ecrire un analyseur lexical qui permettra de :

- Compter le nombre de mots.
- Compter le nombre de ligne.
- Compter le nombre de caractères.
- Retourner la somme des nombres lus.

L'exécution doit être effectuée sur un fichier texte (en entrée) et le résultat est redirigé vers un fichier **resultat.txt.**

Exercice 3

On considère un répertoire téléphonique où chaque ligne est écrite selon le format suivant :
Channouf belgacem @ 04 25 32 15 88
Ben salah jalel @85 75 89 56 56
Soufi mohamed amine@ 45 78 12 37 57
Touons omar chatty @78 65 42 15 19
Brun leon paul@54 89 67 32 45

Donner et exprimer en Flex les définitions régulières pour décrire nom, prénom et téléphone.

a) On s'est restreint aux noms et prénoms exclusivement composés de caractères alphanumériques.
b) Le nom doit commencer par une lettre majuscule.
c) On pourra avoir plusieurs prénoms (écrits qu'en minuscules) séparés par un nombre quelconque de blancs.
d) Le numéro de téléphone est composé de 5 paires de digits séparées par un blanc.

Exercice 4 done

Ecrire un analyseur lexical permettant d'analyser la requêtes SQL suivante :

```
CREaTE TaBLE utilisateur
(
iduser int PRIMARY KEY,
nom varchar(100),
prenom varchar(100),
date_naissance DATE,
ville varchar(255),
code_postal varchar(5)
)
```

Le résultat d'analyse demandée est :

```
CREaTE          CREATE
TaBLE           TABLE
utilisateur             IDENTIFIER
(               (
iduser          IDENTIFIER
int             DATATYPE
PRIMARY KEY             PRIMARY_KEY
,               COMMA
nom             IDENTIFIER
varchar         DATATYPE
(               (
100             NUMBER
)               )
,               COMMA
prenom          IDENTIFIER
varchar         DATATYPE
(               (
100             NUMBER
)               )
,               COMMA
date            DATATYPE
_naissance              IDENTIFIER
DATE            IDENTIFIER
,               COMMA
ville           IDENTIFIER
varchar         DATATYPE
(               (
255             NUMBER
)               )
,               COMMA
code            IDENTIFIER
_postal         IDENTIFIER
varchar         DATATYPE
(               (
5               NUMBER
)               )
)               )
```

Exercice 5

Ecrire à l'aide de Flex un analyseur lexical qui reconnaît les unités lexicales suivantes:

ENTIER : une constante entière
REEL : une constante réelle
IDENT : un identificateur type C
OP-ARTHM : + , - , * , /
OP-REL : < , > , <= , >= , == , !=
AFFECT: =
MOTCLE: if, else, while
CHAINE : une chaîne de caractères entre guillemets
COMMENTAIRE : une ligne commençant par #

a. Votre analyseur devra ignorer les espaces, tabulations, passages à la ligne et les lignes vides.
b. Afficher pour chaque lexème, son type (ENTIER, REEL,…etc.) et sa valeur.
c. Traiter quelques cas d'erreurs en affichant le numéro de la ligne, le type de l'erreur et la chaîne qui pose le problème.

Par exemple, si le fichier en entrée est le suivant :

```
" fichier exemple "

if a >b a=a+2
"chaîne deux " # commmentaire#
3.5  5.3E2   +4.2
-.2e2    2.1et
truc1  x1  1x
"chaîne incomplète
-t
#fin
```

L'analyseur doit indiquer :

CHAINE : " fichier exemple "
MOTCLE: if
IDENT : a
OP-REL : >
IDENT : b
IDENT : a
AFFECT : =
IDENT : a
OP- ARTHM: +
ENTIER : 2
CHAINE : "chaîne deux "
COMMENT : # commmentaire

REEL : 3.500000
……
REEL : -20.00000
…
Détection des erreurs lexicales avec le numéro de ligne
"Chaîne incomplète ** Erreur : ligne 6 **
fin de chaîne attendue
2.1e exposant attendu
1x identificateur mal formé
-x nombre attendu
#fin #attendu

Annexe et référence

Référence gnu : *http://www.gnu.org/software/bison/manual/*

Variables flex:

yyin fichier de lecture (par défaut: stdin)
yyout fichier d'écriture (par défaut: stdout)
char yytext [] : tableau de caractères qui contient le lexème accepté.
int yyleng : la longueur du lexème accepté.

Fonctions :

int yylex ( ) : fonction qui lance l'analyseur.
Int yywrap ( ) : fonction qui est toujours appelée à la fin du texte d'entrée. Elle retourne 0 si l'analyse doit se poursuivre et 1 sinon.

Expressions régulières abrégées

EXPRESSION	CHAÎNES RECONNUES	EXEMPLE
c	le caractère unique non-opérateur c	a
$\backslash c$	le caractère c littéralement	*
"s"	la chaîne s littéralement	"**"
.	tout autre caractère qu'une fin de ligne	a.*b
^	le début d'une ligne	^abc
$	la fin d'une ligne	abc$
$[s]$	un caractère quelconque de la chaîne s	[abc]
$[\hat{}s]$	un caractère quelconque ne faisant pas partie de la chaîne s	[^abc]
$r*$	zéro, une ou plusieurs chaînes reconnues par r	a*
$r+$	une ou plusieurs chaînes reconnues par r	a+
$r?$	zéro ou une fois r	a?
$r\{m,n\}$	entre m et n occurrences de r	a{1,5}
r_1r_2	r_1 suivi par r_2	ab
$r_1 \mid r_2$	r_1 ou r_2	a|b
(r)	mêmes chaînes que r	(a|b)
r_1/r_2	r_1 lorsqu'il est suivi par r_2	abc/123

TP2 - Analyse syntaxique & analyse sémantique (Outil Bison)

Le travail à rendre fait partie d'une évaluation TP.

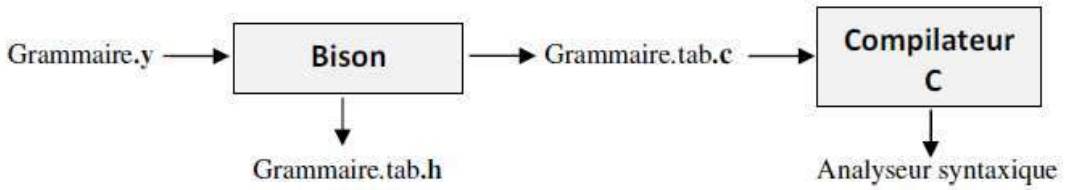
Dans la partie III (Mini-projet), vous avez à choisir un sujet à développer et à rendre

1. ***un rapport détaillant les spécifications du langage utilisé,***
2. ***les fichiers sources (.l et .y).***

Partie I - Rappel (Bison)

Le rôle de l'analyseur syntaxique est de vérifier la syntaxe du programme source. Il reçoit une suite d'unités lexicales fournie par l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

Bison (version GNU de YACC) est un générateur d'analyseurs syntaxiques. Il accepte en entrée la description d'un langage sous forme d'une grammaire et produit un programme écrit en C qui, une fois compilé, reconnaît les mots (programmes) appartenant au langage engendré par la grammaire en entrée.



Un fichier de spécifications Bison se compose de quatre parties :

%{
Les déclarations en c
%}
Déclaration des ***Unités lexicales*** *utilisées*
Déclaration de priorités et de types
%%
Règles de production avec éventuellement des actions sémantiques
%%
Bloc principal et fonctions auxiliaires en C

Règles de production :

Non-terminal **:** $prod_1$
| $prod_2$
| $prod_3$
….
| $prod_n$
;

Les symboles terminaux sont :

- Des unités lexicales (qu'on doit impérativement déclarer dans la partie 2)
  Syntaxe : % *token nom-unité-lexicale*
  Exemple : % *token* NB
  % *token* ID
- Des caractères entre quotes : '+', 'a'…
- Des chaînes de caractères entre guillemets : "while"

Les symboles non-terminaux représentent toute suite de lettres majuscules et/ou minuscules.

Remarque: La partie *Bloc principal et fonctions auxiliaires* doit contenir une fonction **yylex()** effectuant l'analyse lexicale du programme source. On peut soit écrire directement cette fonction soit utiliser la fonction produite par Flex.

Partie II – Installation de l'environnement et Tests

1. Commencer par l'installation l'outil Bison.
2. Ouvrir un nouveau fichier texte et taper le code ci-dessus. Le fichier doit être enregistré avec l'extension **.y** (par exemple *grammaire.y*).
   2. Placer le fichier obtenu dans '*C:\Program Files\GnuWin32\bin* '
   3. A partir de l'invite de commande, lancer la commande :
      C:\Program Files\GnuWin32\bin> ***bison grammaire.y***
   4. En cas de réussite, le fichier *grammaire.tab.c* est généré dans le même répertoire.
   5. Compiler le fichier *grammaire.tab.c* pour générer l'exécutable.
   6. Quels sont les mots acceptés ?

```
#include <stdio.h>
int yylex(void);
int yyerror (char*);
%}
%%
mot : S '$' {printf("mot correct"); getchar();}
;
S : 'a'S'a'
| 'b'S'b'
| 'c'
;
%%
int yylex()
{
char c=getchar();
if (c=='a'||c=='b'||c=='c'||c=='$') return(c);
else printf("erreur lexicale");
}
int yyerror(char *s){
printf("%s \n",s);
return 0;
}
int main(){
yyparse();
printf("\n");
return 0;
}
```

Coordination entre Flex et Bison

On peut utiliser la fonction *yylex()* générée par Flex pour effectuer l'analyse lexicale.

Remarque: Un attribut est associé à chaque symbole de la grammaire (terminal ou non). L'attribut d'une unité lexicale est la valeur contenue dans la variable globale prédéfinie ***yylval***. Cette variable est l'outil de base de communication entre Flex et Bison au cours de l'analyse. Il faut donc penser à affecter correctement ***yylval*** lors de l'analyse lexicale. La définition de ***yylval*** est partagée dans le fichier ".h" issu de "bison -d".

Exemple:

```
%%
[0-9]+ {yylval=atoi(yytext); return NB ;}
%%
```

- Par défaut ***yylval*** est de type entier. On peut changer son type par la déclaration dans la partie 2 d'une union. Exemple :

```
% union {
int entier ;
double reel ;
char * chaine ;
}yylval ;
```

Dans ce cas, on peut stoker dans ***yylval*** des entiers, des réels ou bien des chaines de caractères. Il faut typer les unités lexicales et les symboles non-terminaux dont on utilise l'attribut, soit l'exemple:

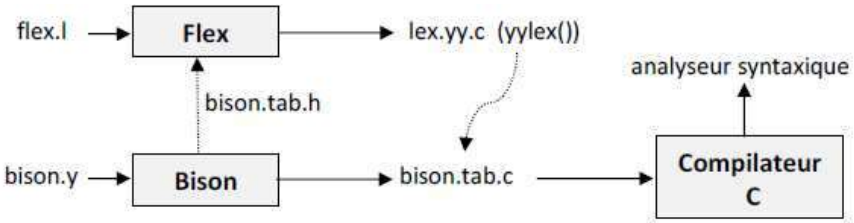
```
% token <entier> NB
% token <chaine> ID
% type <entier> S
% type <chaine> Expr
```

- Pour les symboles non-terminaux, $$ désigne la valeur de l'attribut associé au non-terminal de la partie gauche. $$_i$$ désigne la valeur associée à l'ième symbole non-terminal de la partie droite, par exemple :

```
Expr : Expr '+' Expr {temp=$1 + $3 ;} '*' Expr {$$=temp + $6 ;} ;
```

- Les attributs peuvent être utilisés dans les actions sémantiques.

La génération d'analyseur syntaxique à l'aide de Flex et Bison est illustrée par la figure ci-dessous :



Exercice 1

fait

On veut écrire à l'aide de Flex et Bison un analyseur syntaxique qui permet de reconnaître les instructions de la forme suivante :

somme 1, 2, 3. produit 2,5. $

Il s'agit d'une liste d'entiers séparés par des virgules, se terminant par un point et précédées soit par le mot somme soit par le mot produit.

1. Ecrire tout d'abord le fichier de spécifications Bison suivant :

```
%{
#include<stdio.h>
int yylex(void);
int yyerror(char *s);
%}

%token FIN;
%token SOM;
%token PROD;
%token NB;

%%
liste:FIN {printf("correct");}
      |SOM listesom'.'liste
      |PROD listeprod'.'liste;
listesom: NB|listesom','NB;
listeprod: NB|listeprod','NB;
%%

#include "lex.yy.c"
int yyerror(char *s)
{
   printf ("%s", s);
   return (0);
}
int main()
{
  yyparse();
  getchar();
}
```

2. Compiler le fichier en utilisant l'option –d, par exemple bison -d nom.y
3. En cas de réussite, les fichiers *nom.tab.h* et *nom.tab.c* sont générés dans le même répertoire.
4. Inclure le fichier *nom.tab.h* dans le fichier de spécifications Flex (ce fichier contient la description d'unités lexicales utilisées).
5. Ouvrir un nouveau fichier texte et taper le code de spécifications Flex suivant :

```
%{
#include<stdio.h>
#include<math.h>
#include"nom.tab.h"
%}
%%
[0-9]+ {yylval=atoi(yytext); return NB;}
produit {return PROD;}
somme {return SOM;}
[,|.] {return yytext[0];}
[$] {return FIN;}
[ \t\n] {}
. {printf("Erreur");}
%%
int yywrap()
{return 1;}
```

6. Compiler ce fichier avec Flex pour générer le fichier lex.yy.c.
7. Inclure le fichier lex.yy.c dans le fichier de spécifications Bison.
8. Compiler maintenant le fichier nom.tab.c pour obtenir l'exécutable.
9. Tester le fichier nom.tab.exe obtenu pour vérifier qu'il fonctionne correctement.
10. Modifier l'exercice précédent pour que l'exécutable affiche la somme (respectivement le produit) des entiers formant chaque suite.

Exemple : Si l'entrée est : *somme 2,5. Produit 3,6.$*
Le résultat sera : *Somme = 7*
Produit=18

11. Modifier cet exemple pour construire un analyseur syntaxique permettant de faire la soustraction et la division.

Exercice 2

fait

Compléter le travail du TP1 (outil Flex) afin de réaliser un analyseur syntaxique permettant d'analyser et de vérifier la syntaxe de requête SQL de la forme:

- CREATE TABLE utilisateur(nom varchar(100),prenom varchar(100),ville varchar(255))

Variables et Fonctions à utiliser :

1. **Variables :**
   YYLVAL : variable prédéfinie qui contient la valeur de l'unité lexicale reconnue.
   YYACCEPT: instruction qui permet de stopper l'analyseur syntaxique. Dans ce cas, yyparse retourne la valeur 0 indiquant le succès.
   YYABORT: instruction qui permet également de stopper l'analyseur. yyparse retourne alors 1, ce qui peut être utilisé pour signaler l'échec de l'analyseur.
   % START non-terminal : c'est une action pour dire que le non-terminal est l'axiome.

2. **Fonctions :**
   int yyparse () : fonction principale qui lance l'analyseur syntaxique.
   int yyerror (char *s) : fonction appelée chaque fois que l'analyseur est confronté à une erreur.

Référence gnu : *http://www.gnu.org/software/bison/manual/*

Partie III – Mini-projet

Sujet 1 :

A) En utilisant les outils Flex/Bison, développer un interpréteur d'expressions mathématiques permettant de calculer la valeur d'une expression formées de nombres réels et des opérateurs usuels +, -, * et /. Vous avez à :

1. Donner la spécification de l'analyseur lexical correspondant (fichier Flex).
2. Développer le fichier de spécification Bison définissant l'analyseur syntaxique.
3. Enrichir l'analyseur syntaxique en rajoutant les actions sémantiques permettant d'évaluer une expression donnée en entrée.

B) A ce niveau, on souhaite concevoir et réaliser un interpréteur pour un langage de programmation simple acceptant des variable et des structures simples et itératives telles que :

- des instructions d'affectation,
- des structures conditionnelles de la forme *si-alors-finsi* et *si-alors-sinon-finsi*,
- des structures itératives de type boucle Pour, Répeter, Tantque.

soit l'exemple d'entrée:

```
Si x=y Alors z := 12+24 Finsi
Si x=y Alors z := 3 Sinon x :=2 Finsi
```

1. Enrichir l'interpréteur développé tout en précisant les spécifications nécessaires au niveau lexical (fichier FLex) et au niveau syntaxique (fichier Bison).
2. Vous avez à assurer quelques fonctions de contrôle (aspect sémantiques):
   - évaluer les expressions,
   - détection de l'erreur 'manque de finsi' :

   `Si a=a Alors b := 12+24` //afficher à l'utilisateur: *erreur syntaxique, manque du finsi*
   - détection de boucle infinie,…

Sujet 2 :

On souhaite développer avec les outils Flex/Bison un interpréteur acceptant les requêtes de manipulation et d'accès à une base de données. L'interpréteur doit accepter les requêtes de type :

- Requête create/delete/updtae
- Requête : select

  Exemples :
  ```
  SELECT * FROM fournisseur
  SELECT prenom, nom FROM client WHERE numClt=2
  ```

1. Ecrire la spécification de l'analyseur lexical correspondant
2. Donner la spécification Bison pour l'analyse syntaxique.
3. Rajouter des actions sémantiques pour :
   - calculer le nombre de champs à sélectionner dans la requête.
   -détecter et signaler à l'utilisateur différent types d'erreurs.