**Les expressions régulières**

- Dans une expression régulièretous les caractères sont significatifs, y compris les espaces.

. - Les caractères suivants sont réservés : ils doivent être précédés par \ ou notés entre " " si l'on veut leur valeur littérale.

|  |  |  |
| --- | --- | --- |
| **caractères** | **significations** | **exemples** |
| + | Répéter 1 ou plusieurs fois | x+ 🡪 x...x |
| **\*** | Répéter 0 ou plusieurs fois | t\* 🡪 vide ou t...t |
| ? | 0 ou 1 fois | a? 🡪vide ou a |
| | | union | ab|bc🡪 ab ou bc |
| ( ) | factorisation | (a|b)c 🡪ac ou bc |
| " | valeur littérale des caractères | . "+?"+ 🡪 +?...+? |
| \ | valeur littérale de caractère qui le précède même entre "" | \++🡪+…+  "\"+ 🡪 erreur car la première " ne se ferme pas. |
| . | N’importe quel caractère sauf la fin de ligne (\n). | .|\n 🡪n’importe quel caractère |
| [...] | ensemble de caractères. | [aeiou]🡪 la voyale :**a** ou **e** ou **i** ou **o** ou **u.** |
| - | Utilisé dans [] signifie un intervalle d'ensemble | [0-9] 🡪 0,1,2,3,4,5,6,7,8,9 |
| ^ | Utilisé dans [] signifie le complément d'ensemble | [^0-9] 🡪tout caractère sauf chiffre |
| **Attention :**   1. [...] = ensemble de caractères, pas d'expressions régulières, ce qui veut dire:   [(0|1)+] 🡪 un des caractères (, ), 0, 1, |, +, pas une suite de 0 ou 1.   1. Mais les caractères ^, \, - restent des caractères spéciaux.   [^\]] 🡪tout caractère sauf ] | | |
| {} | Répétition bornée | r{1,5}🡪r ou rr ou rrr ou rrrr ou rrrrr  r{2,}🡪rr ou rrr ou ….  r{3}🡪 rrr |
| ^ | Utilisé hors [], signifie condition de début de ligne. | ^r 🡪 r si seulement r est en début de ligne. |
| $ | Condition de fin de ligne. | r$--> r si seulement r est en fin de ligne. |

**Exercice :**

Donner une expression régulière qui définit les entités suivantes :

1- Une suite de caractères alphanumériques qui commence par un caractère alphabétique.

2- Une suite de caractères alphanumériques qui commence par un caractère alphabétique et qui peut contenir un ‘\_’ au milieu ou à la fin.

3- Les constantes numérique signées par le (-/+).

4- N’importe quel caractère (la fin de ligne (\n) est inclue).

5- Tous les caractère à part les espace, les tabulations et les fins de lignes.

6- Un commentaire de forme /\*ceci est un commentaire\*/

Attention :

/\*\*\*\*\*\*\*\*\*/ est un commentaire.

Alors que /\*\*\*\*\*\*/\*\*\*/ n’est pas un commentaire.

**Solution**

1. [a-zA-Z]([a-zA-Z]|[0-9])\*
2. [a-zA-Z] ]([a-zA-Z]|[0-9]|\_)\*
3. [+-]([1-9][0-9]\*|0)
4. .|\n
5. [^ \n\t]
6. "/\*" ([^\*]\* ("\*"+ [^\*/])\*)\* "\*"+ "/"
7. **Vue générale de processus de compilation :**

Lorsqu’on compile un programme soit :

- Notre programme est juste, donc le compilateur rendre le programme compilé et pris pour être exécuté.

- Notre programme contient quelques erreurs et donc, le compilateur va signaler ces erreurs.

**Compilateur**

Programme source écrit dans un langage X.

Erreurs de compilation

Programme compilé et pris pour être exécuté

Ou

Plus en détail, un compilateur analyse notre programme source en passant par plusieurs étapes :

Erreurs lexicales

Erreurs syntaxiques

Erreurs sémantiques

Erreurs

Analyse lexicale

Analyse syntaxique

Analyse sémantique

Génération et Optimisation de code

Entités lexicales

Arbre syntaxique

Table de symboles

On s’intéresse dans ce TP, aux trois étapes premières qui sont : l’analyse lexicale, syntaxique et sémantique.

Exemple d’erreurs dans  un programme C:

Main ()

{Int y,x=2 ;

Char c ;

If (x==2) y=2 ; /\* instruction juste lexicalement, syntaxiquement et sémantiquement.

If (x==2) 1y=2 ; /\* l’analyse lexicale va détecter une erreur lexicale à cause que ***1y*** est une entité non reconnue

par aucune expression régulière de langage. \*/

if (x==2 y=2 ; /\* il n’y a d’erreur lexicale car toutes les entités de l’instruction correspond à une expression

régulière de langage. Mais il y a une erreur syntaxique car la syntaxe d’une instruction **if** est

donnée par la grammaire: intructionIf🡪 If (expression) instruction ;

et dans l’exemple il manque la parenthèse fermante \*/

if (x==2) y=c ; /\* il n’y a pas d’erreur lexical : toutes les entité est identifiée par des expressions régulières, de

même pas d’erreurs syntaxiques : la syntaxe de l’instruction correspond à la syntaxe de la

grammaire, mais il y a une erreur sémantique car en affecte un caractère ( la variable c) à un

entier (la variable y). l’analyse sémantique utilise pour détecter une telle erreur la table des

symboles\*/

}

1. **Analyse lexicale**

L’analyseur lexical constitue la première étape d’un compilateur. Ses tâches principales sont :

1. Lire les caractères d’entrée et de produire comme résultat une suite d’entités lexicales que l’analyseur syntaxique aura à traiter.
2. L’élimination de caractères superflus comme les commentaires, les tabulations, fin de lignes, …)
3. Gérer les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient.

**Exemple :**

* Code source avant compilation :

If (x==2) y=z ; /\* affectation\*/

* Après l’analyse lexicale :

MotCleIf Sep Idf Egal Const Sep Idf Aff Idf sep

Pour ce faire, l’analyseur lexical a besoin de définitions associées à toutes les entités lexicales de langage à compiler. Il est basé sur l’algorithme simple suivant :

Lire (ChaineEntrée) ;

Switch (ChaineEntrée)

Case (ExpReg1) : écrire (« Entité1 ») ;

Case (ExpReg2) : écrire (« Entité2 ») ;

….

Case (ExpRegN) : écrire (« EntitéN ») ;

Default : écrire (« Erreur lexicale ») ; /\* la chaine d’entrée ne correspond à

aucune entité de langage. \*/

L’implémentation d’un tel algorithme nécessitera l’écriture de centaines/milliers de lignes de code. Heureusement des outils logiciels, tel que F(lex), existent pour nous faciliter la tâche.

**2.1 Flex :**

flex est générateur de code d’analyseur lexical. Il accepte en entrée des spécifications d’entités lexicales sous forme de définitions régulières (les programme flex portent l’extention « .l ») et produit le code source de l’analyseur lexical associé écrit en langage C (par défaut le programme généré par flex porte le nom : lex.yy.c. On peut changer ce nom ultérieurement).

**programme.l lex.yy.c**

Code source de l’analyseur lexicale associé aux expressions régulières définies dans programme.l

Spécification d’expressions régulières

(+ d’autres choses)

flex

* Afin d’utiliser le code généré par flex il faut le compiler, le programme exécutable obtenu est appelé par défaut *a.out* :

Compilateur C

Lex.yy .c

a.out

* « a.out » représente donc l’analyseur lexical sous sa forme exécutable.

Programme analysé

+

erreurs lexicales

a.out

Programme a analysé

* 1. **Format d’un fichier Flex** :

Le format d'un fichier Flex est composé de trois parties séparées par '%%'.

Les définitions

%%

Les règles

%%

Les fonctions

La partie fonction peut être omise. Le plus petit programme flex est le programme « %% », qui ne fait aucun analyse lexicale.

1. **La partie Définition :**

La section de définitions est divisée en deux sous partie :

* Une sous partie qui sert à nommer des expressions régulières pour une utilisation future dans la section règle.

* Une sous partie qui sert à définir des définitions en langage C. cette partie est notée entre %{ et %}. Tous ce qui est écrit dans cette partie sera inséré tel quel dans le fichier lex.yy.c.

Exemple

%{

Int nb\_ligne=0 ;

%}

lettre [a-zA-Z]

%%

***Attention***

***Le nom de l’expression régulière ainsi que les deux %{ et %} doivent être écrit au début ligne, c'est-à-dire à la colonne 0 de la ligne associée. Sinon une erreur va se générer.***

1. **La partie règle :**

* Une règle est de la forme :

Expression\_régulière action ;

ou

{Nom\_expression\_régulière} action ;

* Les actions sont des instructions C. S'il y a plusieurs actions, elles doivent être entre { et }, et dans ce cas elles peuvent tenir sur plusieurs lignes.
* Ne pas indiquer d'action revient à définir une action vide.

Une fois une chaîne reconnue, l'action correspondante est effectuée. Si une chaîne n'est pas reconnue, elle est écrite sur la sortie standard associée à l’analyseur lexical.

La chaîne reconnue est rangée dans un tableau de caractères nommé yytext et sa longueur est rangée dans la variable yyleng (de type int).

Exemple :

%{

int nb\_ligne=0;

%}

lettre [a-zA-Z]

chiffre [0-9]

IDF {lettre}({lettre}|{chiffre})\*

%%

{IDF} printf("identificateur ") ;

[+\*/=-] printf ("opérateur ") ;

[ \t] ;

\n {nb\_ligne++; printf ("\n") ;}

**.** printf("erreur lexicale à la ligne %d \n",nb\_ligne) ;

1. **Les fonctions**

Elles peuvent contenir des fonctions C écrites par l’utilisateur, qui peuvent bien entendu être appelées dans les actions. Ainsi on peut dans cette section redéfinir des fonctions prédéfinies de flex (voir la section 2.4).

**2.3 Commande pour compiler et exécuter un programme flex**

Soit le programme flex progm.l

* **Compilation  de fichier flex:** flex progm.l

Résultat : création d’un fichier C dont le nom lex.yy.c

* **Compilation de fichier C:** gcc lex.yy.c –lfl –o NomExe.exe

Tel que lfl est une bibliothèque externe qu’on doit ajouter.

Résultat : un fichier exécutable qui porte le nom NomExe.exe, ou le nom a.out si on omit l’option –o NomExe.exe.

* **Exécuter l’analyseur sur un programme d’entrée :**

1. **Sous Windows :** NomExe.exe <PgmEntre >PgmSort
2. **Sous Linux :** ./ NomExe.exe <PgmEntre >PgmSort

**2.4 Variables et fonctions prédéfinies:**

Flex utilise un certain nombre de fonctions prédéfinies, dans ce que suit, je cite les fonctions les plus utilisées :

1. **char yytext[]:** tableau de caractères qui contient la chaîne d'entrée en cours d’analyse.

Exemple d’utilisation :

Idf : var

Idf : x

var

x

IDF [a-zA-Z]([a-zA-Z]|[0-9])\*

%%

{IDF} {printf ("idf :  ") ; printf (" %s ",yytext) ;}

PgmCompilé

PgmSource

1. **int yyleng**: retourne la longueur de la chaîne d’entrée en cours d’analyse.

Exemple d’utilisation :

IDF [a-zA-Z]([a-zA-Z]|[0-9])\*

%%

{IDF} {if (yyleng <=7) printf (" idf ") ;

else printf ("erreur lexicale : idf trop longue") ;

}

Idf

Erreur lexicale :idf trop longue

PgmSource

var

vartoplogue

PgmCompilé

1. **ECHO :** recopie le flot d’entrée sur le flot de sortie. Elleest équivalente à printf ("%s", yytext).

Idf : var

Idf : x

PgmSource

var

x

IDF [a-zA-Z]([a-zA-Z]|[0-9])\*

%%

{IDF} {printf (" idf :  ") ; ECHO ;}

PgmCompilé

1. **yyterminate():** fonction qui stoppe l'analyseur lexical.

Idf : var

Idf : x

var

x

stop

y

IDF [a-zA-Z]([a-zA-Z]|[0-9])\*

%%

stop yyterminate() ;

{IDF} {printf (" idf :  ") ;

PgmCompilé

PgmSource

***Remarque****: remarquez que la chaine* ***stop*** *vient avant* ***{IDF}*** *sinon, flex va considérer* ***stop*** *comme* ***idf****.*

1. **int yylex()**: c’est la fonction qui lance l'analyseur lexical.
2. **int main ():**la fonction main() par défaut contient juste un appel à yylex(). L'utilisateur peut la redéfinir dans la section du code additionnel.

%{

int nb\_ligne=0;

%}

%%

\n nb\_ligne++;

%%

int main()

{

yylex();

printf("nombre de ligne %d",nb\_ligne);

}

Var

X

Y

z

Nombre de ligne 4

PgmCompilé

PgmSource

**Exercice 1**

Ecrire un analyseur lexical flex capable de reconnaitre les entités suivantes :

* Les nombres (séquences de chiffres).
* Les mots clés : begin, end.
* Les identificateurs (séquences de chiffres et de lettres commençant par une lettre, et qui ne sont pas des mots clés).
* Les opérateurs :+, -, \*, /

Les espaces, tabulations et retours à la ligne sont des séparateurs. Toute autre séquence de caractères qui ne forme pas une entité est considérée comme une erreur.

L’analyseur lexical doit afficher à l’écran la liste des entités reconnues comme dans l’exemple :

Nombre 12

Identificateur abc

Erreur à ligne 3.

**Exercice 2**

Soit la portion de programme suivante :

INT adr1, adr2 ;

REAL x\_1, y ;

Adr1=Vale ; y=Valr ;

* adr1, adr2, x\_1, y, sont des identificateurs qui sont représentés par une suite de caractères alphanumériques commençant par une lettre, dont la longueur ne dépasse pas 7 caractères.
* Vale est une constante entière dont la longueur ne dépasse pas 5 chiffres et dont la valeur ne dépasse pas 32768.
* Valr est une constante réelle dont la longueur ne dépasse pas 9 chiffres, à part le point décimal.

1. Donner les entités lexicales de la portion de programme.
2. Ecrire l’analyseur lexical correspondant.

**Solution exercice 1**

%{

int nb\_ligne=0;

%}

chiffre [0-9]

lettre [a-zA-Z]

idf {lettre}({lettre}|{chiffre})\*

nombre {chiffre}+

opr [-+\*/]

sep [ \t\n]

mot\_cle begin|end

%%

stop yyterminate();

{mot\_cle} {printf("mot\_clé: "); printf("%s ",yytext);}

{opr} {printf("operateur: "); printf("%s ",yytext);}

\n {nb\_ligne++; printf("\n");}

{sep}

{nombre} {printf("chiffre: "); printf("%s ",yytext);}

{idf} {printf("IDF: "); printf("%s ",yytext);}

. {printf("erreur lexical à la ligne %d ",nb\_ligne); printf("%s ",yytext);}

%%

int main()

{

printf("pour arrêter le programme écrire 'sotp'");

yylex();

}

**Solution d’exercice n°2**

%{

int nb\_ligne=0;

%}

chiffre0 [0-9]

chiffre1[1-9]

lettre [a-zA-Z]

idf {lettre}({lettre}|{chiffre0}|\_)\*

sep [;,=]

mot\_cle INT|REAL

constEn {chiffre1}{chiffre0}\*|0

constRe ({chiffre1}{chiffre0}\*|0)"."{chiffre0}\*{chiffre1}

espace [ \t]

%%

{mot\_cle} {printf("mot\_clé ");}

\n {nb\_ligne++; printf("\n");}

{sep} {printf("separateur ");}

{idf} { if (yyleng<=7) printf("IDF ");

else printf("erreur lexical: IDF trop long ");

}

{constEn} {if ((yyleng<=5)&&(atoi(yytext)<=32768)) printf("contEN ");

else printf("erreur lexicale: contEn erronée "); }

{constRe} {if(yyleng<=10) printf("contRe ");

else printf("erreur lexical: constRe erronée ");

}

{espace} ;

. {printf("erreur lexical à la ligne %d ",nb\_ligne);}

1. **Analyse syntaxique**

L’analyse syntaxique vérifie que les unités lexicales (le résultat de l’analyse lexicale) sont dans le bon ordre défini par le langage.

**Exemple :** dans le langage C l’instruction ***if*** doit se présenter sous la forme :

*If ( expression ) instruction ;*

Si l’analyseur syntaxique reçoit la suite d’unités lexicales

MCIF IDF OPR ENTIER

il doit signaler que qu’il y a une erreur syntaxique car il n’y a pas de ***(*** juste après le ***MCIF***

La vérification de la syntaxe de programme source nécessite l’implémentation d’une des méthodes d’analyse syntaxique descendantes ou ascendantes (vues en cours) : LL(k), LR(k), SLR(k), LALR(k), ..etc. Ceci nécessitera l’écriture de milliers de ligne de code. Heureusement un outil tel que Bison existe pour nous faciliter la tâche.

**3.1 Bison**

Bison est un générateur de code d’analyseur syntaxique. Il accepte comme entrée la sortie de l’analyse lexicale (les entités lexicales), et la grammaire de langage à analyser (les fichiers bison portent l’extension « .y »). Il produit à la sortie le code source de l’analyseur syntaxique associé écrit en langage C ( par défaut le programme généré par Bison porte le nom NomProgramme.tab.c).

NomProgramme.y NomProgramme.tab.c

Entités lexicales

+ grammire

+ d’autres choses

Bison

Code source de l’analyseur syntaxique associé à la grammaire définie dans NomProgramme.y

L’algorithme d’analyse syntaxique de Bison est basé sur la méthode LALR(1).

**3.2 Format d’un fichier Bison**

Le format d’un fichier Bison est similaire à celui de Flex. Il est composé de trois parties séparées par '%%'.

Les définitions

%%

Les règles

%%

Les fonctions

1. **La partie Définition :**

La partie de définitions est divisée en deux sous partie :

**a.1)** **Définition C**

Cette sous partie sert à introduire des définitions en langage C. cette partie est notée entre %{ et %}. Tous ce qui est écrit dans cette sous partie sera inséré tel qu’il dans le fichier NomProgramme.tab.c.

**a.2)** **Définition de symboles de la grammaire**

Cette sous partie sert à définir les symboles de la grammaire : les terminaux (les entités lexicales), l’axiome ainsi que les priorités de quelques terminaux en cas de grammaire ambigüe.

* **L’axiome** est précéder par %start. Si on ne définie pas d’axiome, Bison va considérer, par défaut, comme axiome, la partie gauche de la première règle de la grammaire.
* **Les symboles terminaux** (entités lexicaux) doivent être :

- Dans le cas d’une grammaire non ambigüe : précédés par :%token. Les caractères non alphabétiques doivent apparaître entre **' '**.

- Dans le cas d’une grammaire ambigüe : on doit imposer des priorités et des associativités sur les terminaux pour enlever l’ambigüité.

\* Les terminaux associatifs à gauche : précédés par : %left

\* Les terminaux associatifs à droite : précédés par : %right

\* La priorité est en fonction de l'ordre de déclaration, les moins prioritaires en

premier.

**Exemple :** pour la règle de grammaire

<inst\_if> 🡪 MC\_IF (<expression>) <instruction> MC\_ELSE <instruction>

Les symboles terminaux sont :mc\_if, mc\_else, (, ) . Dans Bison on les déclare par :

**%token mc\_if mc\_else '(' ')'**

**%start inst\_if**

Les symboles non terminaux sont les caractères ou les chaines de caractères non déclarées comme unités lexicales. Les espaces (blanc, fin de ligne, tabulation) ne sont pas significatifs.

**b- Les règles :**

- La partie règle décrit la grammaire du langage ainsi qu’elle insère des actions sémantiques. La syntaxe de la grammaire est la suivante :

<symbole NonTerminal> : <règle de dérivation 1> {action 1 en langage C }

| <règle de dérivation 2> {action 2 en langage C }

| ...

|<règle de dérivation N> {action N en langage C}

;

* On regroupe les règles de dérivation partageant la même partie gauche à l’aide du caractère |
* Les actions sémantiques sont des instructions en langage C insérées dans les règles des dérivations. Elles sont exécutées chaque fois qu’il y a dérivation par la règle de dérivation associée.

**Exemple**: analyse syntaxique d’une instruction if simple

%token mc\_if mc\_else '(' ')' v f

%start inst\_if

%%

instIf : Mcif '(' cond ')' inst Mcelse inst {printf("syntaxe correct");}

;

cond: v | f

;

inst :

;

7

**c- Les fonctions**

Elle peut contenir des fonctions C écrites par l’utilisateur. Ainsi on peut dans cette section redéfinir des fonctions prédéfinies de Bison (voir la section **3.5**). Pour que l’analyseur syntaxique fonctionne, la partie fonction doit contenir au minimum la fonction main.

main ()

{ yyparse(); }

**3.3. Passerelle entre Flex et Bison** :

La passerelle entre Flex et Bison se réalise en plusieurs étapes :

**a) Inclusion de l’entête généré par Bison au début du fichier Flex**

On doit compiler le fichier bison NomProgramme.y avec l’option –d comme suit :

**>Bison –d NomProgramme.y**

Cette compilation rendre comme résultat deux fichiers : NomProgramme.tab.c et NomProgramme.tab.h.

On doit inclure ce dernier fichier au début du fichier Flex.

**Exemple :**

%{

# include ”NomProgramme.tab.h ”

%}

.

**b) Retourne des entités lexicales**

Dans le code de l’analyseur lexical, à la rencontre d’une entité lexical, l’action associée va être cette fois un : *return entité ;* (au lieu de *printf(‘’entité’’) ;* ). Cette instruction va renvoyer à l’analyseur syntaxique l’unité lexicale *entité*.

**Exemple :**

%{

#include "Nomprogramme.tab.h"

%}

%%

if return(Mcif);

else return(Mcelse);

[)(] return (yytext[0]);

[ ] ;

v return(v);

f return(f);

stop yyterminate();

. printf("erreur lexicale: entité non reconnue");

**c) Les entités lexicales et les symboles terminaux :**

Les entités lexicales de flex sont les terminaux de la grammaire de bison.

%token Mcif Mcelse '(' ')' v f

%%

instIf : Mcif '(' cond ')' inst Mcelse inst {printf("syntaxe correcte"); YYACCEPT;}

;

cond: v | f

;

inst :

;

%%

main ()

{

printf("Taper stop pour arreter \n");

yyparse();

}

* 1. **Commande pour compiler et exécuter un programme bison/flex**

Analyseur Exécutable

Lex.yy.c

Fichier.l

Description lexicale

Analyseur lexical

Identificateurs des terminaux

NomFichier.y

NomFichier.tab.c

Description

syntaxique

Analyseur syntaxique

NomFichier.tab.h

En récapitulant les étapes de réaliser un analyseur syntaxique d’un langage donné avec l’outil Bison :

1. Il faut décrire la syntaxe de langage dans un format propre à bison dans un fichier qui porte l’extension « .y » ( exp : *NomFichier.y* ).
2. Il faut compiler le fichier *NomFichier.y* avec le compilateur Bison et ceci avec la commande : **>*bison –d NomFichier.y***

Cette compilation aura comme résultats la génération de deux fichiers, à savoir : *NomFichier.tab.c* et *Nom.tab.h.*

Le fichier *NomFichier.tab.c*: est l’analyseur syntaxique de notre langage.

Le fichier *NomFichier.tab.h*: est un fichier qui contient la liste de tous les terminaux de notre langage.

1. Il faut compiler le fichier *NomFichier.tab.c* afin d’obtenir un exécutable. Pour ce faire il faut ajouter l’analyseur lexical, de notre langage, obtenu à l’aide de flex (exp : *lex.yy.c*) à la ligne de commande :

***>gcc NomFichier.tab.c lex.yy.c –lfl –ly –o NomFichierExecutable.exe***

1. On lance l’exécution de notre analyseur par :

sous Windows : **> *NomFichierExecutable.exe***

sous Linux **: > *./ NomFichierExecutable.exe***

**Exercice (maison)**

Ecrire l’analyseur lexical/syntaxique à base de flex/bison de l’instruction while

<InstWhile> 🡪 while (<Cond>) do <Inst> end

<Cond> 🡪 idf<OprLog>idf

<Inst> 🡪 idf :=idf ;

<OprLog> 🡪 = | < | >

**Solution**

1. partie flex

%token Mcwhile Mcdo Mcend '(' ')' '=' '>' '<' ' ;' AFF v f idf

%%

instWhile : Mcwhile '(' cond ')' Mcdo inst ' ;' Mcend {printf("ok"); YYACCEPT;}

;

cond: idf OprLog idf | v | f

;

OprLog : '>'|'<'|'='

;

inst : idf AFF idf |

;

%%

main ()

{

printf("Taper stop pour arreter \n");

yyparse();

}

%{

#include "exp2.tab.h"

%}

idf [a-zA-Z]([a-zA-Z]|[0-9])\*

%%

while return(Mcwhile);

do return(Mcdo);

end return(Mcend);

[()=<>] return yytext[0];

[ \t\n] ;

v return(v);

f return(f);

stop yyterminate();

{idf} return(idf);

":=" return(AFF);

**.** printf("erreur lexicale: entité non reconnue");

2) Partie bison

* 1. **Fonctions prédéfinies**
* **YYACCEPT :** instruction qui permet de stopper l’analyseur syntaxique en cas de sucées.
* **main () :** elle doit appeler la fonction yyparse (). L’utilisateur doit écrire son propre main dans la partie du bloc principal.
* **yyparse () :** c’est la fonction principale de l’analyseur syntaxique. On doit faire appelle à cette fonction dans la fonction main().
* **int yyerror ( char\* msg )** : lorsque une erreur syntaxique est rencontrée, *yyparse* fait appelle à cette function. On peut la redéfinir pour donner plus de détails dans le message d’erreur. Par défaut elle est définie comme suit:

int yyerror ( char\* msg ) {

printf ( ” Error : %s encountered \n ” , msg );

}

* 1. **Les attributs**

Pour faire référence au symbole terminal ou non terminal dans les actions sémantiques, Bison utilise la notion d’attributs. Deux symboles d’attribues sont utilisées $$ et $i.

* Le symbole $$ référence la valeur associée au non terminal de la partie gauche d’une règle de grammaire.
* Le symbole $i référence la valeur associée au i ème symbole terminal **ou** non terminal **ou** action sémantique de la partie droite d’une règle de grammaire.

**Exemple 1**

E : E ’+’ E {$$ = $1 + $3; }’-‘E{$$ = $$- $6; }

Exemple 2

InstDiv : idf = idf/0 {if ($5==0) printf ( "erreur sémantique : division par Zéro"); }

**Mais Mais Mais !!!**

Comment Bison peut traiter les valeurs des entités lexicales, sachant que l’analyse syntaxique travaille sur la sortie de l’analyse lexicale où les chaines d’entrées de programme source sont déjà codées ?

Idf=const / const ;

X=4/0 ;

source

Analyse lexicale

La solution est que flex peut (mais il n’est pas obligé) envoyer les valeurs des entités lexicales avant de les codés.

* 1. **Echange de valeurs d’entité lexicales entre flex et bison**

Au-delà de la codification de l’entité lexicale que flex envoie à bison par l’instruction

*return EntitéLexicale.* Flex peut envoyer ainsi la valeur associée à chaque entité et ceci en utilisant la variable *yylval*.

Avant d’utiliser *yylval* on doit la déclarer comme variable de type *YYSTYPE*. Ce dernier est un type prédéfini de flex, il est par défaut équivalant au type C *int*, mais on peut changer sa définition par défaut.

**Exemple d’utilisation :**

1. **Partie flex**

%{

#include "exp3.tab.h"

extern YYSTYPE yylval;

%}

cst [0-9]

%%

{cst} { yylval=atoi(yytext);

return cst;

}

[/] return yytext[0];

. printf("erreur lexicale");

1. **Partie bison**

%token cst '/'

%%

somme : cst '/' cst { if ($3==0) printf(" erreur : division par zéro");

else printf("la divion est %d", $1/$3); YYACCEPT;}

;

%%

main ()

{

yyparse();

}

Nous pouvons changer le type de YYSTYPE pour qu’il contient d’autres types ou même une combinaison de types.

1. **YYSTYPE comme type simple**

On change le type de YYSTYPE dans la partie déclaration C de fichier Bison.

**Exemple**

1. **partie Bison**

%{

# define YYSTYPE char

%}

%token idf '+'

%%

somme : idf '+' idf {printf("réduction de %c +%c en somme",$1,$3); YYACCEPT;}

;

%%

main ()

{

yyparse();

}

1. **Partie flex**

%{

#include "exp4.tab.h"

extern YYSTYPE yylval;

%}

idf [a-z]

%%

{idf} { yylval=yytext[0];

return idf;

}

[+] return yytext[0];

. printf("erreur lexicale");

1. **YYSTYPE comme union de plusieurs types**

On change le type de YYSTYPE dans la partie définition de fichier Bison pour qu’il soit une union de plusieurs types en utilisant le mots clé bison « %union »

Exemple

1. **Partie Bison**

%union {

int entier;

char caractere;

}

%token <caractere> idf <entier> cst '+'

%%

somme : idf '+' cst {printf("réduction de %c +%d en somme",$1,$3); YYACCEPT;}

;

%%

main ()

{

yyparse();

}

1. **Partie flex**

%{

#include "exp5.tab.h"

extern YYSTYPE yylval;

%}

cst [0-9]

idf [a-z]

%%

{idf} { yylval.caractere=yytext[0];

return idf;

}

{cst} { yylval.entier=atoi(yytext);

return cst;

}

[+] return yytext[0];

. printf("erreur lexicale");