Master 1 informatique : TP de compilation

Analyse lexico-syntaxique avec LEX et YACC

1. Lex : générateur d'analyseur lexical (+variante Flex)
2. Yacc : générateur d'analyseur syntaxique
3. Lex avec Yacc

ENT : cours en ligne (Clarotice)

→ Analyse lexicale et syntaxique avec Lex et Yacc

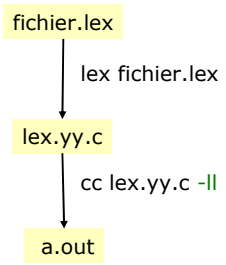
- Formation Lex & Yacc
- Exercices de TP
- (Solutions)
- (Sujet d'examen)

Lex

- Utilitaire Unix (Linux: Flex)
- Flux d'entrée → suite de motifs

1. Définir les motifs
   ← méta-langage (E.R.)
2. Préciser les actions associées
   ← code C/C++ (E.R. aug.)
3. Générer le code source
   → en C/C++
4. Compiler
   ← librairie Lex

   utiliser –lfl avec Flex !



Le fichier-lex

4 parties (3 facultatives):

1. Code utilisateur préalable
2. Définitions opératoires
   - définitions opératoires
   - expressions prédéfinies
3. Règles de production
4. Code utilisateur final

```
{%
int v=0;
%}

%s config
voyelle [aeiou]

%%
{voyelle}   v++
%%

main()
{yylex(); printf("voyelles:%d",v);}
```

tab

seul sur ligne !

%% obligatoire avec Flex

attention! espaces et tabulations

Les E.R. vues par Lex

Méta-caractères:

{ }	1. utilisation d'expressions 2. nombre d'occurrences
%	initiateur (p. 3 et 4)
< >	configurations (p. 3)
" "	délimiteur de texte brut
( )	délimiteur de motif
\	\n, \s, \t, …
[ ]	délimiteur ensembliste
+ * ?	# occurrences
-	
/	suivi de
|	ou
^	début de ligne
$	fin de ligne
.	tout caractère sauf \n

$
^
-

négation (après [)
intervalle (entre 2 caractères)

→ erlex.htm

LEX : ordre des motifs ?

```
%%
a       {simple++;}
aa      {double++;}
aaa     {triple++;}
```

aaaaaaaaaaa

→ triple=0, double=0, simple=11

```
%%
aaa     {triple++;}
aa      {double++;}
a       {simple++;}
```

aaaaaaaaaaa

→ triple=3, double=1, simple=0

```
%%
aaa     {triple++; REJECT;}
aa      {double++; REJECT;}
a       {simple++;}
```

aaaaaaaaaaa

→ *?! triple=9, double=10, simple=11*

N.B. Les solutions de TD ont été compilées avec Lex ...

FLEX : longueurs des motifs

```
%%
a       {simple++;}
aa      {double++;}
aaa     {triple++;}
%%
```

aaaaaaaaaaa

→ triple=3, double=1, simple=0

```
%%
aaa     {triple++;}
aa      {double++;}
a       {simple++;}
%%
```

aaaaaaaaaaa

→ triple=3, double=1, simple=0

```
%%
aaa     {triple++; REJECT;}
aa      {double++; REJECT;}
a       {simple++;}
%%
```

aaaaaaaaaaa

→ *?! triple=9, double=10, simple=11*

... Adaptez les solutions de TD à Flex !

Variables et fonctions lex

yytext	tableau contenant le motif reconnu
yyleng	taille de yytext
yylval	"last value" de type YYTYPE (cf Yacc)

ECHO	affiche yytext
BEGIN(conf)	place l'analyseur dans la configuration indiquée
REJECT	replace le motif reconnu dans le flot d'entrée
yymore()	conserve dans yytext le motif (le motif suivant sera rajouté en suffixe - au lieu d'écraser yytext)
yyless(n)	comme yymore() mais supprime d'abord les n premiers caractères du motif

Configurations d'analyse

```
%%
<INITIAL,condition1>^C$         {BEGIN (condition2);}
<INITIAL,condition2>^V$         {BEGIN (condition1);}
<condition1>voyelle             {printf("-");}
<condition2>consonne            {printf("+");}
<condition1,condition2>^I$      {BEGIN(INITIAL);}
```

Au début de chaque règle de production

Valeur par défaut: INITIAL

Compilation

Sous Unix (Lex) :

Librairie Lex : –ll

Sous Linux (Flex) :

librairie Flex : -lfl

code C ↔ compilation C
code C++ ↔ compilation C++

Exécution

Origine et destination du flux d'analyse:

par défaut ← stdin (clavier) et → stdout (écran)

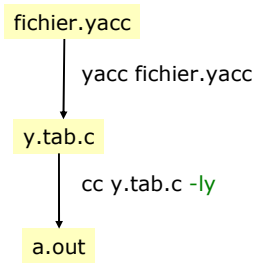
+ Redirigeable (Unix: <, >)

+ Reprogrammable (Lex: yyin)

Yacc

- Utilitaire Unix (GNU: Bison)
- analyse syntaxique (grammaire algébrique)

1. Définir les règles de production
   ← méta-langage
2. Préciser les actions associées
   ← code C/C++ (gram. aug.)
3. Générer le code source
   → en C/C++
4. Compiler
   ← librairie Yacc



Le fichier Yacc

4 parties (3 facultatives):

1. Code utilisateur préalable

```
{%

%}
```

2. Définitions opératoires
   - définitions opératoires
   - définitions de lexèmes
   - table de précédence

```
%token plus
[aeiou] return voyelle
%left plus
```

3. Règles de production

```
%%
S : voyelle S voyelle
   |
   ;
```

4. Code utilisateur final

```
%%
int yylex() {…}
```

attention! espaces et tabulations

Les règles de production

```
S : L S          $$=$1+$2;
  | L            $$=$1;
  |              $$=0;
  ;
L : consonne     $$=0;
  | voyelle      $$=1;
  ;
```

Grammaire correspondante: S→LS|L|ε, L→c|v

Convention: NON-TERMINAL / terminal (AT&T: convention inverse)

Variables $ pour transmettre des valeurs dans l'arbre d'analyse


```
S : nombre plus S     $$=$1+$3;
  | nombre mult S     $$=$1*$3;
  | nombre            $$=yylval
  ;
```

yylval: valeur attribuée au token nombre (par défaut type int)

type de yylval redéfinissable dans le code préalable (cf ymf.htm)

symbole de départ: le premier trouvé

ou choisi dans la partie 2: %start *symbole*

Table de précédence (des opérateurs)

%left
%right
%nonassoc

→ CM de compilation et de théorie des langages

Variables et fonctions prédéfinies

error : token qui récupère une impasse dans la table d'analyse

YYABORT : équivalent à return 1
YYACCEPT : équivalent à return 0
YYERROR : équivalent à return 1

main() {yyparse();}
code prédéfini, ajouté dans la partie 4 par –ly
return met fin à l'analyseur yyparse

main(){while yyparse()=0}
code à placer dans la partie 4
pour relancer l'analyseur après un return nul

Interfacer Lex et Yacc

Pour utiliser Yacc sans Lex, on peut:

1. remplacer les tokens par des caractères
   (pexp: 'a', 'e', 'i', 'o', 'u' pour voyelles)
2. définir la fonction yylex
   (pexp: yylex(){char c=getchar(); return c;})

Sinon il faut "interfacer" les deux applications…

…avec des tokens

fichier-lex :

```
%{
#include y.tab.h
%}
%%
[aeiou]+            {return v;}
[b-df-hj-np-tv-z]   {return c;}
.|\n                {return autre;}
...
```

fichier-yacc :

```
...
%token v
%token c
%token autre
...
%%
S : S v       $$=$1+1;
  | S c       $$=$1;
  | S autre   $$=$1;
  ;
...
```


transmission des valeurs

fichier-lex :

```
%{
extern int yylval;
…
%}
…
[0-9]+ {yylval=atoi(yytext); return n;}
[+]    {return p;}
...
```

fichier-yacc :

```
...
%token n
...
%%
S : S p n       {$$=$1+$3;}
  | n           {$$=yylval;}
  ;
...
```

typage des valeurs (#define YYSTYPE) → inter_ly.htm

Interfacer et compiler

Dépendances:

- yylex() défini avant yyparse()
- yyparse() défini avant main()
- -ly -ll (main de Yacc prioritaire)
- tokens définis dans fichier-yacc et utilisés dans fichier-lex.

Procédé de base:

- Construire fichier-lex et fichier-yacc
- yacc -d fichier.yacc (créer y.tab.h)
- Ajouter #define y.tab.h dans fichier-lex
- lex fichier.lex
- Compiler dans le bon ordre : gcc lex.yy.c y.tab.c -ly -ll