Université de Bordeaux. Collège Sciences et Techniques.
Licence d'Informatique, Semestre 6, Session de printemps, 2017/2018.
UE 4TIN603U : Compilation. Sujet de : G. Sénizergues.

Compilation

Devoir surveillé du 25/04/18

Durée : 1h 30
Tous documents autorisés

Exercice 1 (6 pts) Analyse lexicale

On souhaite reconnaître les mots sur l'alphabet $\{a, b, c\}$ qui ont au moins un facteur dans l'ensemble $F = \{abbb, abc, acc, abccc\}$. Par exemple, sur l'entrée :
`abbbaaaccabccaaaabccc`
le programme devra retourner : `mot sur a,b,c et un facteur est present`
Le fichier `exo1.1.0.l` donné en annexe donne ce résultat.
1- On souhaite que l'analyseur teste aussi que le mot est écrit sur l'alphabet $\{a, b, c\}$, i.e. qu'il ne contient pas de caractère autre que a, b, c. Par exemple, sur l'entrée :
`abbbaaaccdabccaaaabccc`
le programme devra retourner : `lettre autre que a,b,c`
1.1 Le fichier `exo1.1.0.l` donnera-t-il ce résultat ?
1.2 Ecrire un fichier flex `exo1.1.l` qui fournisse, sur toute entrée, le résultat correct.
2- On souhaite donner la liste des couples (position, numero du facteur) dans le mot d'entrée (on appelle "position" du facteur son numéro de colonne et on suppose que le mot est écrit sur une seule ligne). Par exemple, sur l'entrée :
`abbbaaaccabccaaaabccc`
le programme devra retourner : `(0,0)(6,2)(9,1)(16,3)`
Ecrire un fichier flex `exo1.2.l` qui produise un analyseur qui ait ce comportement.
3- On souhaite maintenant lire des mots écrits sur *plusieurs lignes* ; on appelle "position" du facteur son numéro de colonne dans la ligne courante. Par exemple sur l'entrée :
`abbbaaaccabccaaaabccc`
`aaaabcabbb`
le programme devra retourner :`(0,0)(6,2)(9,1)(16,3)(3,1)(6,0)`
Ecrire un fichier flex `exo1.3.l`, qui produise un analyseur qui ait ce comportement.
4- On considère l'ensemble de mots $G = \{abbb, ab, acc, abccc\}$. Reprendre la question 2 avec ce nouvel ensemble de mots G.
Par exemple sur l'entrée :
`abbbbacccabccc`
le programme devra retourner : `(0,1)(0,0)(5,2)(9,1)(9,3)`
Ecrire un fichier lex `exo1.4.l` qui produise un analyseur qui ait ce comportement.
***5**- On considère l'ensemble de mots $H = \{abbb, ab, bbc, cab\}$
Reprendre la question 2 avec ce nouvel ensemble de mots H.
Par exemple sur l'entrée :
`abbbcacacabbc`
le programme devra retourner : `(0,0)(0,1)(2,2)(8,3)(9,1)(10,2)`
Rappel : l'instruction flex `yyless(n)` fait reprendre l'analyse lexicale à la position n du mot

courant (la position 0 est celle de la lettre la plus à gauche du mot courant).

Exercice 2 (7 pts) **Analyse syntaxique et attribut numérique**

Les fichiers `arit.l`, `arit.y` donnés en annexe engendrent, après traitement par bison, flex et gcc, un anayseur des expressions arithmétiques. La table `arit.output` fournie par bison ne signale *aucun conflit*.

0- Que représente le fichier `arit.output` par rapport à la grammaire contenue dans `arit.y` ? Que peut-on conclure de l'absence de conflit ?

On cherche, dans cette exercice, à analyser et interpréter des expressions *booléennes*. Les expressions booléennes sont les mots engendrés par la grammaire suivante (écrite dans le format de bison) :

`expr: expr OR expr | expr AND expr | NEG expr | BOOL |'(' expr ')'`

Les non-terminaux sont `expr term factor` et les unités lexicales `BOOL AND OR NEG` sont définies par le fichier `bool.l` donné en annexe.

1- Donner une grammaire LALR(1) pour les expressions booléennes (*sans* utiliser de règles de précédences).

Aide : on pourra s'inspirer de la grammaire donnée dans `arit.y`.

2- Écrire un fichier `evalbool.y` qui permet d'*évaluer* des expressions booléennes.

Aide : on peut munir la grammaire de la question 1 d'un *attribut synthétisé* de type entier.

3- Le professeur Cosinus (qui est parfois distrait) utilise le fichier flex `bool.l` et le fichier bison `evalbug.y` suivant :

```
-----------------------------------------------------------------
                            /* evalbug.y */
%{
#include  <stdio.h>
#include  <ctype.h>
extern int yylex();
int yyerror(char *s);
%}

%start start
%token BOOL AND OR NEG

%%
start: expr  {printf("valeur: %d",$1);}

expr:   expr OR expr  {$$= $1 || $3;}
        | expr AND expr {$$= $1 && $3;}
        | NEG expr      {$$= 1-$1;}
        | BOOL          {$$= $1;}
        |'(' expr ')'   {$$= $1;}
;
%%
#include "lex.yy.c"

int yyerror(char *s){
  fprintf( stderr, "*** ERROR: %s\n", s );
  return 0;
}

int main(int argn, char **argv){
    yyparse();
}
-----------------------------------------------------------------
```

Sur l'entrée `f and t` (resp. `f or t`) son programme imprime `0` (resp. `1`). Cosinus est ainsi (provisoirement) confiant.

3.1 Quelles sorties produira son programme sur chacune des entrées suivantes ?

`f and t or t,  neg t or t ,  neg t and f or f`

3.2 La confiance de Cosinus dans son programme est-elle justifiée ?
4- Corriger le fichier `evalbug.y` de Cosinus, *sans changer* ni les règles de grammaire, ni les règles de calcul d'attributs, mais en introduisant des directives `%left %right %nonassoc`, de facon à produire des évaluations correctes.
La convention à respecter est que : pour tous $x, y, z \in \{t, f\}$,
l'expression x `or` y `and` z doit être interprétée comme x `or` (y `and` z) ;
l'expression x `and` y `or` z doit être interprétée comme (x `and` y)`or` z ;
l'expression `neg` x `and` y doit être interprétée comme (`neg` x) `and` y ;
l'expression x `and neg` y doit être interprétée comme x `and` (`neg` y).

Exercice 3 (7 pts) **Analyse syntaxique et attributs non-numériques**
On considère la grammaire G donnée (au format BISON) dans le fichier `exo3.1.y` de l'annexe. On produit une commande `exo3.1` en appliquant `flex` à `arit.l`, `bison` à `exo3.1.y` et `gcc` au fichier `exo3.1.tab.c`.
Par exemple, sur l'entrée `1 + 2` , la commande `exo3.1` affiche `r5 r4 r2 r5 r4 r1`.
1- Qu'affichera la commande `exo3.1` sur chacune des entrées ? :
`0 * 1 + 1,  - 1 + 1,  - 1 * 2 + 3`
2- Que calcule la commande `exo3.1` en général ? une dérivation gauche ? droite ? le miroir d'une dérivation gauche ? le miroir d'une dérivation droite ? autre chose ?
Exemple : une dérivation *gauche* du mot `NUMBER * NUMBER + NUMBER` est `r1 r2 r3 r4 r5 r5 r4 r5`
3- Munir la grammaire G de règles de calcul d'un attribut synthétisé de facon à calculer un *arbre de dérivation* et à imprimer le parcours *préfixe* de cet arbre de dérivation. On écrira ces calculs d'attributs, au format de bison, dans un fichier `exo3.3.y` qui pourra inclure le module `tree.h` de l'annexe et utiliser les fonctions déclarées dans `tree.h`. [on supposera déjà disponible le fichier `tree.c` correspondant] [1].
Exemple : à partir du mot `NUMBER * NUMBER + NUMBER`, l'arbre de dérivation est dessiné sur la figure **??** et votre commande imprimera
`expr,expr,term,term,factor,NU,Mu,factor,NU,Pl,term,factor,NU`

4- Comment peut-on, à partir de la grammaire attribuée de la question 3, imprimer une *dérivation gauche* du mot analysé ? Par exemple, sur l'entrée `0*1+1` il faudrait afficher
`r1 r2 r3 r4 r5 r5 r4 r5`. Adapter le fichier `exo3.3.y` en un fichier `exo3.4.y` qui réalise cet affichage.

ANNEXE

```
-----------------------------------------------------------------
-----------------------------------------------------------------
                          /* exo1.1.0.l */
%{
 int res=0;
%}

%%
abbb    {res=1;}
```

1. écrire le fichier `tree.c` demande des compétences en *programmation* que nous supposons acquises par le candidat ; la correction des projets a confirmé cette hypothèse

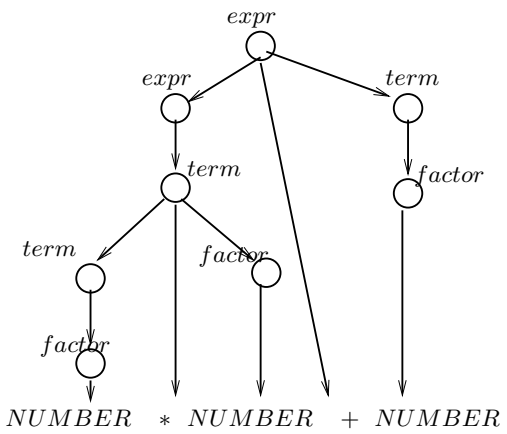

FIGURE 1 – arbre de dérivation

```
abc     {res=1;}
acc     {res=1;}
abccc   {res=1;}
.       {;}
\n      {;}
%%

int main()
{ yylex();
 if (res==1)
 printf("un facteur parmi abbb, abc, acc, abccc est present\n");
 else
 printf("aucun facteur detecte \n");
}
-----------------------------------------------------------------------
-----------------------------------------------------------------------
                              /* arit.l*/
%{
%}
sep               [ \t\n]+

%option nounput noinput
%%

[0-9]+ {return NUMBER;}
"+"    {return PLUS;}
"*"    {return MUL;}
"-"    {return MINUS;}
{sep}  {;}
.      {return *yytext;}

%%

int yywrap(){
   return 1;
}
-----------------------------------------------------------------------
                              /* arit.y*/
%{
#include  <stdio.h>
#include  <ctype.h>
extern int yylex();
int yyerror(char *s);
```

```
%}

%start start
%token NUMBER PLUS MUL MINUS

%%

start: expr {printf("expression correcte");}

expr: expr PLUS term
| term
;

term: term MUL factor
| factor
;

factor: NUMBER
| MINUS NUMBER
|'(' expr ')'
;

%%
#include "lex.yy.c"

int yyerror(char *s){
  fprintf( stderr, "*** ERROR: %s\n", s );
  return 0;
}

int main(int argn, char **argv){
  if (yyparse()==0)
    printf(" expression correcte");
}
-------------------------------------------------------------------
-------------------------------------------------------------------
                              /* bool.l*/
%{
%}
sep               [ \t\n]+

%option nounput noinput
%%

"t"    {yylval=1;return BOOL;}
"f"    {yylval=0;return BOOL;}
"and"  {return AND;}
"or"   {return OR;}
"neg"  {return NEG;}
{sep}  {;}
.      {return *yytext;}

%%
int yywrap(){
   return 1;
}
-------------------------------------------------------------------------
-------------------------------------------------------------------------
                            /* exo3.1.y */
%{
#include <assert.h>
#include  <stdio.h>
#include <stdlib.h>
#include  <ctype.h>
#include  <string.h>
extern int yylex();
```

```
int yyerror(char *s);
/* la regle courante */
 char *regle;
/* derivation ? or not derivation ? */
 char *deriv;
 #define MAXDERIV 200
 %}

%start expr
%token NUMBER PLUS MUL MINUS

%%

expr: expr PLUS term  {regle=strdup("r1 ");
                       strcat(deriv,regle);}
| term                {regle=strdup("r2 ");
                       strcat(deriv,regle);}
;

term: term MUL factor {regle=strdup("r3 ");
                       strcat(deriv,regle);}
| factor              {regle=strdup("r4 ");
                       strcat(deriv,regle);}
;

factor: NUMBER         {regle=strdup("r5 ");
                        strcat(deriv,regle);}
| MINUS NUMBER         {regle=strdup("r6 ");
                        strcat(deriv,regle);}
|'(' expr ')'          {regle=strdup("r7 ");
                        strcat(deriv,regle);}
;

%%
#include "lex.yy.c"

int yyerror(char *s){
  fprintf( stderr, "*** ERROR: %s\n", s );
  return 0;
}


int main(int argn, char **argv){
  deriv=malloc(MAXDERIV);
  yyparse();
  printf("suite des regles:\n %s",deriv);
--------------------------------------------------------------------------
--------------------------------------------------------------------------
                           /* tree.h */
#ifndef TREE_H
#define TREE_H
/*--------------------------constantes-types--------------------------*/
#define ARMAX 3 /* arite max des arbres de derivation                  */
#define MAXL 10 /* longueur max d'une etiquette                        */
struct Tree{
  struct Tree *fils[ARMAX]; /* tableau des fils                       */
  int regle;/* numero de regle de grammaire etiq(noeud) -->etiq(fils)  */
  char *etiq; /* etiquette: non-terminal                              */
  };
/*---------------------------------fonctions--------------------------*/
extern struct Tree *createTree();/* retourne un struct Tree */
extern char *Idalloc();/* retourne une chaine de longueur MAXL*/
extern void affecTree(struct Tree *tree,struct Tree *f1,struct Tree *f2,struct Tree *f3, int rg,char *et);
/* affecte les champs de tree par les arguments */
extern void prefixTree(struct Tree *tree); /* ecrit tree en notation prefixe */
#endif
```