Université de Bordeaux. Collège Sciences et Techniques.
Licence d'Informatique, module Compilation, 2016/2017.

Compilation

Examen du 25/04/17 : une solution

Barème :
Ex1(sur 15) : Q1/1,Q2/2,Q3/1,Q4/1,Q5/1,Q6/2,Q7/1,Q8/2,Q9/2,Q10/1,Q11/2
Ex2(sur 14) : Q1/0,5,Q2/0,5,Q3/1,Q4/1,Q5/1,Q6/1,Q7/2,Q8/2,Q9/2,Q10/3.

Exercice 1 (15 pts) **Grammaires et attributs**

1- Le mot $NopNopN$ admet deux dérivations gauches distinctes dans G :

$$\begin{aligned} expr &\rightarrow expr\ \ op\ \ expr \rightarrow N\ \ op\ \ expr \rightarrow N\ \ op\ \ expr\ \ op\ \ expr \rightarrow N\ \ op\ \ N\ \ op\ \ expr \\ &\rightarrow N\ \ op\ \ N\ \ op\ \ N \end{aligned}$$

$$\begin{aligned} expr &\rightarrow expr\ \ op\ \ expr \rightarrow expr\ \ op\ \ expr\ \ op\ \ expr \rightarrow N\ \ op\ \ expr\ \ op\ \ expr \rightarrow N\ \ op\ \ N\ \ op\ \ expr \\ &\rightarrow N\ \ op\ \ N\ \ op\ \ N \end{aligned}$$

Elle est donc ambiguë.
Comme toute grammaire LALR(1) est non-ambiguë, on en déduit que G n'est pas LALR(1).
2- Les trois grammaires engendrent le *même langage* que G.
La grammaire G_2 admet un analyseur descendant, de gauche à droite, déterministe :
si l'on doit dériver un mot terminal commençant par $N\ \ op$, alors on choisit d' utiliser la règle $expr \rightarrow N\ \ op\ \ expr$,
si l'on doit dériver un mot terminal commençant par $N\$$, alors on choisit d'utiliser la règle $expr \rightarrow N$,
La grammaire G_2 est donc non-ambiguë. La grammaire G_1 est miroir de la grammaire G_2. Comme G_2 est non-ambiguë, G_1 est aussi non-ambiguë.
Le mot $N\ \ op\ \ N$ admet deux dérivations gauches distinctes dans G_3 :

$$expr \rightarrow N\ \ op\ \ expr \rightarrow N\ \ op\ \ N$$

$$expr \rightarrow expr\ \ op\ \ N \rightarrow N\ \ op\ \ N$$

Elle est donc ambiguë.
3- Quel arbre de dérivation cet analyseur construit-il à partir du mot `N + N + N` ?
Après avoir lu le préfixe `N + N`, l'analyseur, se trouve dans l'état 5. avec dans la pile `expr + expr` et une vue en avant qui vaut `+` . Cet état présente un conflit "décalage/réduction", et l'analyseur choisit le décalage (il va dans l'état 4). La suite de son calcul consiste alors à : réduire le dernier `expr + N` en `expr + expr` puis en `expr`
L'arbre de dérivation associé est celui de la figure 1.
4- Evaluons les expressions :

$$e_1 = (2{*}{*}2){*}{*}3 \mapsto 4{*}{*}3 \mapsto 64$$

$$e_2 = 2{*}{*}(2{*}{*}3) \mapsto 2{*}{*}8 \mapsto 256$$

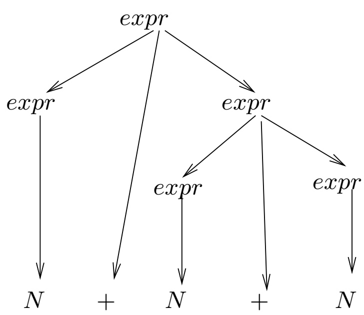

FIGURE 1 – arbre de dérivation de $N + N + N$

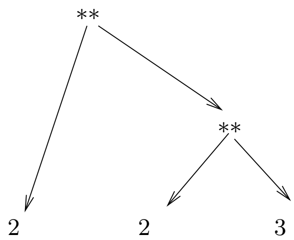

FIGURE 2 – arbre de syntaxe abstraite

Elles n'ont pas la même valeur. L'opération $**$ sur l'ensemble des entiers $\mathbb{N}$ n'est donc pas associative.
5- L'arbre de syntaxe abstraite associé à 2$**$2$**$3 doit comporter un sous-arbre dont les feuilles forment le facteur 2$**$3. On choisit donc l'arbre de la figure 2
6-
<u>Grammaire G</u> :
Elle n'est pas LALR(1) : donc la table d'analyse générée par BISON présentera des conflits. Cependant, en "imposant à l'analyseur" un choix de résolution qui correspond aux arbres de la forme vue à la question 5, on pourra obtenir une interprétation correcte. En fait, le choix de résolution *par défaut* de BISON consiste à privilégier "décaller" sur "réduire", ce qui produit les arbres corrects. Donc G peut être utilisée.
<u>Grammaire G_1</u> :
Elle est non-ambiguë. Mais l'arbre de dérivation (unique) de 2$**$2$**$3 *n'a pas* la forme recquise (pas de sous-arbre dont les feuilles forment le facteur 2$**$3). Elle n'est donc *pas* adaptée à l'écriture d'un interpréteur.
<u>Grammaire G_2</u> :
Elle est non-ambiguë et l'arbre de dérivation (unique) de 2$**$2$**$3 *a* la forme recquise. (En fait elle est aussi LALR(1)). Elle est adaptée à l'écriture d'un interpréteur.
<u>Grammaire G_3</u> :
Elle est ambiguë. Néanmoins le choix de résolution *par défaut* de BISON produit les arbres corrects : l'analyseur n'utilise jamais la règle 2 pour réduire. Il produit donc les mêmes arbres que l'analyseur de la grammaire G. Donc G_3 peut être utilisée (quoiqu'elle possède une règle inutile).

Le classement de la plus adaptée à la moins adaptée est finalement :

$$G_2 \;>\; G \;>\; G_3 \;>\; G_1.$$

7- Choisissons G_2 :

```
...
%start expr
%token op NUMBER

%%

expr:  NUMBER op expr {$$ = pow($1,$3); val=$$;}
    |  NUMBER         {$$ = $1;}
    ;

%%
...
```

8- Le professeur Cosinus a choisi la grammaire G_1. On peut néanmoins interpréter les expressions comme suit :
- on calcule l'arbre de syntaxe abstraite qui est un peigne gauche
- on évalue l'expression par un calcul descendant (de la racine de l'arbre vers ses feuilles).
N.B. Ce calcul pourrait être décrit formellement par un attribut *hérité* ; mais BISON ne génère pas automatiquement l'évaluation des attributs hérités [1].

```
...
%start expr
%union{NOE NO;}
%type <NO> expr
%token <NO> op NUMBER

%%

expr: expr op NUMBER {$$=Nalloc();
                      $$->ETIQ=malloc(2);
                      strcpy($$->ETIQ,"**");
                      $$->FG=$1;
                      $$->FD=$3;
                      syntree=$$;
                      }
    | NUMBER         {$$ = $1;
                      }
    ;

%%

#include "lex.yy.c"

/* retourne l'attribut de l'expression t(qui est un peigne gauche)
si t est de profondeur >= 2, att(t,c):= op(fd(t),c));
si t est de profondeur =1, att(t,c):= op(t,c);
*/
int att(NOE n,int attoncld)
{ if(n != NULL)
     {if ((n->FG) && (n->FD))
         {assert (strcmp(n->ETIQ,"**")==0);
          int vd=atoi(n->FD->ETIQ);
          return(pow(vd,attoncld));
         }
```

1. Le logiciel SYNTAX (conçu par INRIA, voir `https://fr.wikipedia.org/wiki/SYNTAX`), permet de le faire, mais il semble ne plus être maintenu depuis quelques années.

```
      else
         {int v=atoi(n->ETIQ);
          return(pow(v,attoncld));
         };
      }
   else
      return(0);
}

/* retourne la valeur de l'expression */
int eval(NOE n)
{NOE ncour=n;
 int atcour=1;
 while (ncour)
   {atcour=att(ncour,atcour);
     ncour=ncour->FG;
   }
 return(atcour);
}

int main(int argn, char **argv){
  yyparse();
  printf("valeur de l'expression: %d", eval(syntree));
}
```

9- La grammaire H comporte les règles

$$expr \to expr \;\; op1 \;\; expr$$

et

$$expr \to N.$$

On a vu à la question 1 que ces deux règles suffisent pour construire deux dérivations gauches différentes du mot $N \;\; op1 \;\; N \;\; op1 \;\; N$. La grammaire H est donc *ambiguë*. Par conséquent elle n'est *pas LALR(1)*.
10- L'expression 2+3+4 a plusieurs arbres de dérivation dans H : on l'a vu à la question 9. Ce phénomène n'empêche pas d'atteindre le but d'évaluer les H-expressions : les deux arbres produits conduisent aux évaluations de (2+3)+ 4 et de 2+(3+ 4), qui fournissent le même résultat (car l'opération d'addition des entiers est associative).
11- On peut utiliser une grammaire LALR(1), calquée sur la grammaire de l'exercice 3.6 du TP3 [2]

```
%{
#include  <stdio.h>
...
 int val=0;        /* valeur */
%}
%start expr
%token op1 op2 NUMBER

%%

expr:  term op1 expr {$$ = $1+$3; val=$$;}
    |  term          {$$=$1;}
    ;
term:  NUMBER op2 term {$$ = pow($1,$3);}
    |  NUMBER          {$$ = $1;}
    ;

%%
```

2. ceux qui avaient traité cette grammaire avec BISON en TP étaient *sûrs* qu'elle est LALR(1)

On peut aussi utiliser une grammaire ambiguë (non LALR(1), ni LR(1), ni même LR(k) pour aucun entier k !), que BISON traduit néanmoins en un analyseur à pile déterministe.

```
%{
#include  <stdio.h>
...
 int val=0;        /* valeur */
%}

%start expr
%token op1 op2 NUMBER
%left op1
%right op2
%%

expr:  expr op1 expr {$$ = $1+$3; val=$$;}
    |  expr op2 expr {$$ = pow($1,$3); val=$$;}
    |  NUMBER         {$$ = $1;}
    ;

%%

#include "lex.yy.c"


int main(int argn, char **argv){
  yyparse();
  printf("%d \n",val);
}
```

Exercice 2 (14 pts) **Interprétation des tableaux**

Un exemple

1- Soit $w \in \{0,1\}^*$ un mot de longueur ℓ. Le mot $\varphi(w)$ est obtenu en remplaçant chaque lettre (de longueur 1) par un mot de longueur 2, il est donc de longueur $2 \cdot \ell$.
2- On vérifie que,

$$t_1 := 01, \;\; t_2 := \varphi(01) = 0110, \;\; t_3 := \varphi(0110) = 01101001.$$

Et

$$t_4 = \varphi(01101001) = 0110100110010110.$$

3- La suite ℓ_n vérifie :

$$\ell_0 = 1, \ell_{n+1} = 2 \cdot \ell_n$$

donc $\ell_n = 2^n$.
4- Notons $P(p)$ la propriété :

$$L = 2^p, W = \varphi^p(0).$$

Avant le premier passage dans la boucle `while (p < n)` :

$$L = 1, W = 0$$

ce qui est la propriété $P(0)$.
Suposons que, après exécution de la dernière ligne du while $P(p)$ est vraie et que $p < n$.
Le corps de la boucle effectue successivement les actions :

$$W := \varphi(W); p := p + 1; L := 2 * L.$$

Après l'action 1 on a :

$$L = 2^p, W = \varphi^{p+1}(0)$$

puis, après l'action 2 :

$$L = 2^{p-1}, W = \varphi^p(0)$$

puis, après l'action 3 :

$$L = 2^p, W = \varphi^p(0)$$

Autrement dit : $P(p)$ est invariante par exécution du corps de la boucle, On en déduit, par récurrence, que $P(p)$ est toujours vraie en fin d'exécution de la boucle.
5- En début d'exécution : $\texttt{ptasl} = 1$ (car l'adresse 0 est réservée pour nil).
Après execution de `W := new array of integer[L];` comme $L = 1$ on a $\texttt{ptasl} = 2$. Avant le premier passage dans la boucle `while (p < n)`, on a donc

$$p = 0, L = 1, \texttt{ptasl} = 2$$

Notons $Q(p)$ la propriété :

$$L = 2^p, \texttt{ptasl} = 2^{p+1}.$$

Juste avant le premier passage dans la boucle `while (p < n)` : $Q(0)$ est donc valide.
Le corps de la boucle effectue successivement les actions :

```
nW := new array of integer[2*L];p := p+1;  L := 2*L.
```

Après l'action 1 on a :

$$L = 2^p, \texttt{ptasl} = 2^{p+1} + 2 * L = 2^{p+2}$$

puis, après l'action 2 :

$$L = 2^{p-1}, \texttt{ptasl} = 2^{p+1}$$

puis, après l'action 3 :

$$L = 2^p, \texttt{ptasl} = 2^{p+1}$$

Autrement dit : $Q(p)$ est invariante par exécution du corps de la boucle,. On en déduit, par récurrence, que $Q(p)$ est toujours vraie en fin d'exécution de la boucle.
6- En ce point : $\texttt{p} = \texttt{n}$ et $\texttt{L} = 2^p, \texttt{ptasl} = 2^{p+1}$ La zone de `TAS` qui stocke la variable W est de longueur L, et elle se termine à l'indice $\texttt{ptasl} - 1$: c'est donc la suite de places

$$\texttt{TAS}[2^n], \ldots, \texttt{TAS}[2^{n+1} - 1].$$

La partie qui reste accessible au programme est la zone d'indices $i \geq 2^n$. La partie devenue inaccessible au programme est la zone d'indices $i < 2^n$. Lorsque $p = n$ vaut 20, la taille de la zone du tableau `TAS` devenue inaccessible est donc 2^{20}.

7-

```
            taille=valeur(e);            /* taille du tableau                 */
            res=min(PADRL);
            PADRL=PADRL- {res};
            ADR[res]=ptasl;
            TAL[res]=taille;
            ptasl+=taille;              /* mise a jour allocateur  memoire   */
            return(res);
```

8-

```
int NINDL,i,j;
j=0;
for (i=0;i<ADMAX;i++)
    if (TAL[i]!=0)
        {INDL[j]=i;j++;
        };
NINDL=j; /* nbe d'indices i tq TAL[i] != 0 */
```

On dispose d'une fonction de tri de profil :

```
qsort(void * tableau, size_t nb_element, size_t taille_element,
      int (*compare)(void const *a, void const *b));
```

On définit alors la fonction auxiliaire `compare` puis on lance le tri :

```
int compare( int *a, int *b)
    if ADR[*a] < ADR[*b]
        return(-1);
     else if (ADR[*a] == ADR[*b])
        return(0);
     else
        return(1);


qsort(INDL,NINDL,sizeof(int), &compare);
```

9-

```
void TASSERG()
  int nouv_ad=1; /* nouvelle adresse dans le TAS */
  for(j=0; j < NINDL;j++)
      {ADR[INDL[j]]=nouv_ad;
       nouv_ad += TAL[INDL[j]];
      }
```

10- La mise à jour du compteur de références (REF) est désormais fonction de la dimension du tableau (affecté ou supprimé).

```
/* k est un indice de ADR, d est la dimension du tableau de valeur k */
void incref(int k, int d)
    {REF[k] = REF[k]+1;
     if d >= 2
        for(i=0;i<TAL[k];i++)
             incref(TAS[ADR[k]+i],d-1);
    }

/* k est un indice de ADR, d est la dimension du tableau de valeur k */
void decref(int k, int d)
    {REF[k] = REF[k]-1;
     if (REF[k]==0)
        {insert(k,ADRL);/* inserer l'entier k dans ADRL*/
         TAL[k]=0;
        };
     if d >= 2
        for(i=0;i<TAL[k];i++)
             decref(TAS[ADR[k]+i],d-1);
    }
```

L'instruction `T := T'` (où `T`,`T'` sont des variables de type tableau de dimension $d \geq 1$) est interprétée par :

```
incref(T');
decref(T);
T=T';
```

`NewAr(e)` est interprétée comme dans les Q7,8,9 et la fonction `TASSERG()` est la même.