Polytech Nice Sophia
SI4 / 2021-2022
Erick Gallesio

Corrigé TD 2

Grammaires

Analyse Descendante

1 Généralités grammaires

1.1 Une première grammaire

Une grammaire possible pour produire le langage

$a^n b^{m+n} c^m$:

```
S → AB
A → aAb | ε
B → bBc | ε
```

1.2 Appels de fonction

On avait la grammaire G1 définie comme:

```
call   → ident ( liste ) | ident ( )
liste  → liste , expr | expr
```

Factorisation à gauche de G_1

On travaille sur la première règle:

```
call   → ident ( params
params → liste ) | )
```

Note: au prix d'une ε-production, on peut aussi factoriser à droite, et obtenir:

```
call   → ident ( params )
params → liste | ε
```

qui est un peu plus lisible.

Suppression de la récursivité gauche de G_1

La troisième règle peut se réécrire en:

```
liste → expr suite
suite → , expr suite | ε
```

La grammaire G_2 est donc:

```
call   → ident ( params )
params → liste | ε
liste → expr suite
suite → , expr suite | ε
```

Forme BNF

La forme BNF de G_1

```
call  ::= ident ( [liste] )
liste ::= expr {, expr}
```

On peut voir que la forme BNF de G_2 est bien sûr identique et que, par conséquent, les deux grammaires sont bien équivalentes.

1.3 Grammaire ambiguë

Grammaire G

```
   S →  a S b S                  (r1)
     |  b S a S                  (r2)
     |  ε                        (r3)
```

Prenons par exemple le mot `abab` ; il peut être produit par les deux suites de dérivations suivantes:

```
S → aSbS → abS → abaSbS → ababS → abab
     r1     r3     r1      r3     r3
```

et

```
S -> aSbS → abSaSb → abaSb → abab
     r1      r2       r3       r3
```

Le langage engendré par G est l'ensemble des mots formés d'autant de `a` que de `b`.

1.4 S-expressions

La grammaire des *s-expressions* de Lisp est très simple:

```
PROG    → SEXPR PROG | ε
SEXPR   → atom    | ( MEMBERS )
MEMBERS → SEXPR MEMBERS | ε
```

1.5 Déclarations C

La forme BNF du sous-langage des déclarations pourrait être:

```
decl     ::= typename var { ',' var } ';'
typename ::= ident | int | float | ...
var      ::= { '*' } ident { '[' integer ']' }
```

A partir de cette forme, on obtient la grammaire suivante:

```
decl     → typename var listevar ';'
typename → ident | int | float | ...
listevar → ',' var listvar | ε
var      → star ident brackets
star     → '*' star | ε
brackets → '[' integer ']' brackets | ε
```

2 PREMIERs et SUIVANTs

2.1 Grammaire LL(k)

Rappel: on a la grammaire

```
S → A B C e
A → a A | ε
B → b B | c B | ε
C → d e | d a | d A
```

Pour les PREMIERs, on a:

```
PREMIER(S) = PREMIER(A) ∪ PREMIER(B) ∪ PREMIER(C) = { a, b, c, d }
PREMIER(A) = { a, ε }
PREMIER(B) = { b, c, ε }
PREMIER(C) = { d }
```

Pour les SUIVANTS, on a:

```
SUIVANT(S) = { $ }
SUIVANT(A) = PREMIER(B) ∪ PREMIER(C) ∪ SUIVANT(C) =  { b, c, d ,e }
SUIVANT(B) = PREMIER(C) = { d }
SUIVANT(C) = { e }
```

On peut donc construire la table d'analyse de cette grammaire.

	a	**b**	**c**	**d**	**e**	**$**
S	ABCe	ABCe	ABCe	ABCe		
A	aA	ε	ε	ε	ε	
B		bB	cB	ε		
C				de da dA		

Cette grammaire n'est pas LL(1) puisqu'il y a 3 règles dans la case T[C, d]. On peut voir que si on avait 2 symboles d'avance, on ne pourrait toujours pas discriminer entre les règles qui ont en partie droite `da` et `dA` puisque le non terminal `A` possède `a` dans ses premiers. La grammaire n'est donc pas LL(2), non plus. A vérifier: elle est LL(3).

2.2 Grammaire de blocs

On a donc la grammaire:

```
Bloc    → '{' L_decl L_instr '}'
L_decl  → d ';' L_decl | ε
L_instr → i ';' L_instr | Bloc L_instr | ';' | ε
```

Pour les PREMIERs, nous avons:

```
PREMIER(Bloc)    = { '{' }
PREMIER(L_decl)  = { d, ε }
PREMIER(L_instr) = { i, '{', ';', ε }
```

Pour les SUIVANTs:

```
SUIVANT(Bloc)    = { i, '{', '}', ';', '$' }
SUIVANT(L_decl)  = { i, '{', '}', ';' }
SUIVANT(L_instr) = { '}' }
```

Nous pouvons donc construire la table d'analyse:

	i	d	{	}	;	$
Bloc			'{' L_decl L_instr '}'			
L_decl	ε	d ';' L_decl	ε	ε	ε	
L_instr	i ; L_instr		Bloc L_instr	ε	;	

Comme il n'y a aucun conflit dans la table, la grammaire est bien LL(1).

Séparateur vs terminateur:

Pour considérer `';'` somme un séparateur (et non plus comme un terminateur), il suffit de transformer la règle pour `L_instr`. Celle-ci devient:

```
L_instr → i | i ; L_instr | Bloc L_instr | ε
```

Nous avons maintenant une règle qui n'est pas factorisée à gauche. Il faut donc la transformer en :

```
L_instr → i Suite | Bloc L_instr | ';'
Suite   → ';' i Suite | ε
```

On peut vérifier que cette nouvelle grammaire est aussi LL(1). Ceci est laissé en exercice.

3 Connexion de Lex à un analyseur LL(1)

Pour cet exercice, nous reprenons l'analyseur lexical du TD 1 (en le simplifiant car on a pas de variables ici). Du coup, la variable `yylval` n'est plus une union, mais simplement un entier. Notez que le calcul de la valeur entière dans `yylval` est complètement inutile ici puisque nous nous bornons à ne faire que l'analyse syntaxique dans cet exercice. Toutefois, nous la calculons quand même ici car nous en aurons besoin pour l'exercice suivant.

Le code de l'analyseur lexical en lex peut donc être:

```
/*
 *  lex.l                       -- Un lexical simple pour implémenter ETF
*/

%{

#include <stdio.h>
#include "analyseur-ll.h"
extern int yylval;

%}

%option noyywrap

%%

[0-9]+                  { yylval = atoi(yytext); return INT; }
"+"                     { return PLUS;  }
"-"                     { return MINUS; }
"*"                     { return MULT;  }
"/"                     { return DIV;   }
"("                     { return OPEN;  }
")"                     { return CLOSE; }
\n                      { return EOL; }
<<EOF>>                 { return EOF; }

[ \t]                   { }
.                       { fprintf(stderr, "Unexpected character %c (%d)\n",
                                  *yytext, *yytext); }
%%
```

Par ailleurs, il nous faut un fichier permettant de communiquer les valeurs choisies pour les constantes représentant les unités syntaxiques (OPEN, CLOSE, PLUS, …). Ceci est fait dans le fichier `analyseur-ll.h` :

```
/*
 * automate.c        -- Automate déterministe pour ETF
 */

#define OPEN  300
#define CLOSE 301
#define PLUS  302
#define MINUS 303
#define MULT  304
#define DIV   305

#define INT   310
#define EOL   311

extern int yylex(void);
```

Il ne nous reste plus maintenant qu'à définir la fonction `verify`. Celle-ci est assez proche de celle du cours:

```
// Verify current token & advance
static bool verify(token tok) {
  if (next != tok) {
    fprintf(stderr, "Invalid token '%d'\n", next);
    return false;
  }
  return advance();
}
```

avec une fonction `advance` définie simplement comme:

```
// Advance
static bool advance(void) {
  next = yylex();
  return true;
}
```

Remarque:

Nous avons ici deux fonctions qui permettent d'avancer sur la phrase à analyser:

- `verify` qui vérifie que l'on a bien l'unité attendue et qui avance si c'est bien le cas
- `advance` qui passe à l'unité syntaxique suivante de façon systématique.

Pour pouvoir analyser une phrase, le code minimal pourrait donc être :

```
   advance();       // pour initialiser le token d'avance (dans next)
   E();             // pour analyser l'expression
   verify(EOL);     // pour vérifier qu'on a bien un '\n' après l'expression
```

Nous devons ensuite définir une fonction par non-terminal de la grammaire. A titre d'exemple, le code de la fonction F est:

```
static bool F() {
  switch (next) {
    case INT:
      return advance();
    case OPEN:
      return advance() && E()&& verify(CLOSE);
    default:
      fprintf(stderr, "Expected INT or OPEN\n");
      return false;
  }
}
```

Code complet de l'analyseur:

- Analyse lexicale: **lex.l**;
- Analyse syntaxique: **analyseur-ll.c**.

4 Production de code pour une machine à pile

Nous allons ici modifier un peu l'analyseur précédent pour lui faire produire du code pour une machine à pile. Pour cela, nous allons introduire la production de code directement dans l'analyseur. OK, dans l'absolu, ce n'est pas très beau, mais nous n'avons pas besoin de construire un arbre pour produire le code et on peut se contenter de produire les instruction directement à la volée.

Par exemple, la fonction **F**, qui doit produire un `PUSH` pour les entiers reconnus, pourrait être:

```
// ---------- F Non Terminal
static bool F() {
  switch (next) {
    case INT:
      printf("\tPUSH %d\n", yylval);
      return advance();
    case OPEN:
      return advance() && E()&& verify(CLOSE);
    default:
      fprintf(stderr, "Expected INT or OPEN\n");
      return false;
  }
}
```

Quant au code de la fonction *Ep* qui était

```
// ---------- E Non Terminal
static bool Ep() {
  switch (next) {
    case PLUS:
    case MINUS:
      return advance() && T() && Ep();
    case CLOSE:
    case EOL:
      return true;
    default:
      fprintf(stderr, "Expected PLUS or CLOSE\n");
      return false;
  }
}
```

il devient tout simplement:

```
// ---------- E Non Terminal
static bool Ep() {
  int  op;

  switch (op = next) {
    case PLUS:
    case MINUS:
      if (advance() && T()) {
        printf("\t%s\n", (op == PLUS) ? "PLUS" : "MINUS");
        return Ep();
      }
      return false;
    case CLOSE:
    case EOL:
      return true;
    default:
      fprintf(stderr, "Expected PLUS or CLOSE\n");
      return false;
  }
}
```

Code complet de notre mini compilateur:

- Analyse lexicale (le même que précédemment): **lex.l**;
- Analyse syntaxique & “production de code”: **comp.c**.

5 Calculatrice

> *Cette version n'était **pas demandée**. Elle est donnée ici pour terminer complètement la calculatrice.*
>
> ***Attention**: Pour simplifier, aucun contrôle de débordement de la pile n'est fait ici!*

On a juste ajouté quelque fonctions qui manipulent effectivement une pile. Le code est très semblable à ce que l'on avait dans l'analyseur précédent.

Code complet de la calculatrice:

- Analyse lexicale (le même que précédemment): **lex.l**.
- Analyse syntaxique & évaluation: **calc.c**.