Polytech Nice Sophia
SI4 / 2021-2022
Erick Gallesio

Feuille 2

Grammaires

Analyse Descendante

1 Généralités grammaires

1.1 Une première grammaire

Donnez une grammaire qui engendre le langage

> $a^n b^{m+n} c^m$.

1.2 Appels de fonction

La grammaire G_1 d'un appel de fonction d'un langage de programmation est définie comme:

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

1. Factoriser à gauche cette grammaire
2. Enlever la récursivité gauche.

Soit G_2 cette nouvelle grammaire.

1. Donner la forme BNF de G1 et de G2.
2. Utiliser les formes BNF précédentes pour montrer l'équivalence de G_1 et de G_2.

1.3 Grammaire ambiguë

Soit la grammaire G dont les productions sont:

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

1. Montrer que cette grammaire est ambiguë en donnant une phrase pour laquelle il existe deux arbres de dérivation distincts.

2. Quel langage engendre cette grammaire?

1.4 S-expressions

Les langages de la famille Lisp utilisent une notation unique pour représenter les programmes ou les données que l'on appelle *s-expressions*. Une *s-expression* est en fait

- soit une forme atomique (symbole, nombre, chaîne, …),
- soit une forme parenthésée constituée, éventuellement d'éléments atomiques (entiers, symboles, chaînes, …) et de *s-expressions*.

Un programme dans ces langages est constitué d'une suite de *s-expressions*.

Ainsi, le programme Scheme (un dialecte de Lisp) suivant permet de définir la fonction `fact` et de calculer ensuite `fact(1000)` :

```
(define fact
  (λ (n)
    (if (< n 2)
        1
        (* n (fact (- n 1))))))

(fact 1000)
```

Donner la grammaire d'un programme Scheme / Lisp.

1.5 Déclarations C

Donnez une grammaire qui engendre un sous-langage des déclarations de C, par exemple :

```
    int i, a[3][5], *p;
```

On ne traitera ici que les déclarations de variables de type simples, des tableaux et des pointeurs. Par ailleurs, on supposera que les tableaux ont toujours toutes leurs dimensions exprimées et qu'elles sont entières.

Construire tout d'abord la forme BNF qui engendre le sous-langage demandé. A partir de cette forme BNF, construire la grammaire demandée.

2 PREMIERs et SUIVANTs

2.1 Grammaire LL(k)

Soit G La grammaire dont les productions sont:

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

1. Calculer l'ensemble des PREMIERs et des SUIVANTs de G.

2. Est-ce que cette grammaire est LL(1)? Si ce n'est pas le cas, est-elle LL(k) et pour quelle valeur de k?

2.2 Grammaires de blocs

Les langages de programmations à blocs permettent en général la déclaration de variables dans le bloc avant les instructions. La grammaire ci-dessous permet de représenter de tels blocs:

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

Ici, les terminaux `d` et `i` symbolisent une déclaration et une instruction du langage.

1. Calculer l'ensemble des PREMIERs et des SUIVANTs de la grammaire de blocs

2. Est-ce que cette grammaire est LL(1)? Si ce n'est pas le cas, est-elle LL(k) et pour quelle valeur de k?

Dans les langages de la famille de C, le terminal `';'` est un terminateur. Cela veut dire qu'il est obligatoirement présent à la fin d'un énoncé. Dans les langages issus de Pascal (Modula, Ada, ..) le point virgule est un séparateur, (c'est à dire qu'il n'est pas obligatoire à la fin d'un bloc). Dans ces langages, on peut donc écrire (plus ou moins, au sucre syntaxique près):

```
if (cond) { x=1; y=2 } else { x=2; y = 3; }
```

Modifiez la grammaire des blocs pour permettre l'utilisation du point virgule comme un séparateur.

3 Connexion de Lex à un analyseur LL(1).

En partant de l'analyseur LL déterministe du cours d'une version minimale de ETF, construisez un analyseur utilisant *lex* pour construire les lexèmes du langage.

Votre analyseur devra reconnaître les lexèmes OPEN, CLOSE, PLUS, MINUS, MULT, DIV, INT, EOL. Ces constantes pourront être définies comme des macros C (dont les valeurs seront supérieures à 257).

Connectez ensuite cet analyseur lexical à un analyseur LL déterministe, écrit en C, qui reconnaît la grammaire ETF définie dans l'exercice précédent.

Note: La version de l'analyseur fournie en cours ne traite pas les opérateurs `'-'` et `'/'`. Étendez la grammaire pour reconnaître les 4 opérations.

4 Un premier compilateur

Nous allons construire notre premier compilateur. Pour cela, reprendre l'analyseur précédent pour lui faire produire du **code pour une machine à pile**. Les instuctions de notre machine à pile sont:

- `PUSH v` qui permet d'empiler la valeur v
- `PLUS`, `MINUS`, `MULT`, `DIV` pour les 4 opérateurs
- `PRINT` pour afficher la valeur en sommet de pile.

Un exemple de sortie de ce programme est donné ci-dessous (observer que les priorités son bien gérées).

```
> 2+3*4
    PUSH 2
    PUSH 3
    PUSH 4
    MULT
    PLUS
    PRINT
> 2*3+4
    PUSH 2
    PUSH 3
    MULT
    PUSH 4
    PLUS
    PRINT
> (2+3)*4
    PUSH 2
    PUSH 3
    PLUS
    PUSH 4
    MULT
    PRINT
> (((5)))
    PUSH 5
    PRINT
```

5 Calculatrice

Modifier le programme précedent pour qu'il affiche le résultat des expressions lues (on passe bien sûr ici par une pile).