Tutorial 3 : Syntax Analysis

Khoi Dang Do - 1711807
khoi.do.1711807@hcmut.edu.vn

September 2019

Base Question

Question 1

Given the grammar

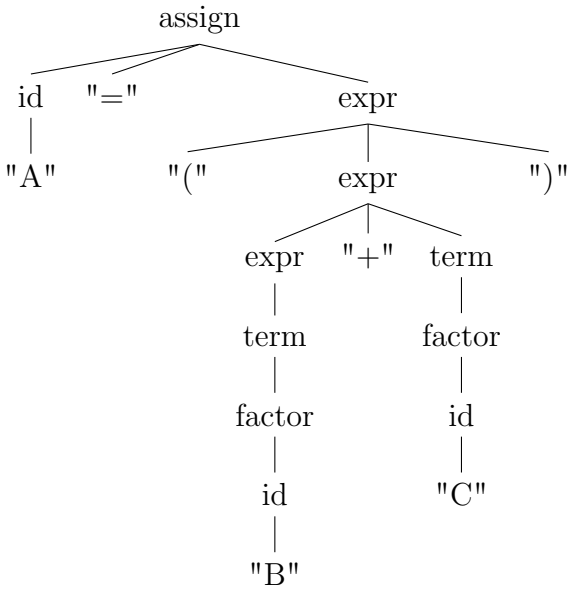
assign	→	id "=" expr
id	→	"A" | "B" | "C"
expr	→	expr "+" term | term
term	→	term "*" factor | factor
factor	→	"(" expr ")" | id

Show a parse tree and a leftmost derivation for each of the following statements:

(a) $A = A * (B + C)$

Leftmost derivation :

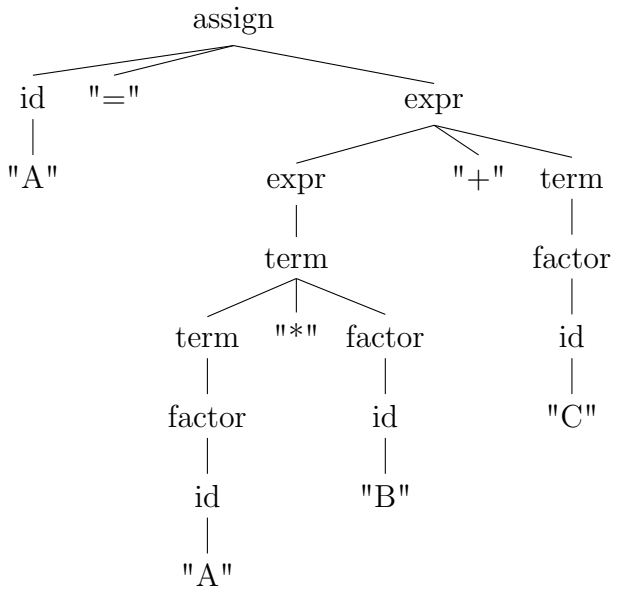
assign → id "=" expr
→ "A" "=" expr
→ "A" "=" "(" expr ")"
→ "A" "=" "(" expr "+" term ")"
→ "A" "=" "(" term "+" term ")"
→ "A" "=" "(" factor "+" term ")"
→ "A" "=" "(" factor "+" factor ")"
→ "A" "=" "(" id "+" factor ")"
→ "A" "=" "(" id "+" id ")"
→ "A" "=" "(" "B" "+" id ")"
→ "A" "=" "(" "B" "+" "C" ")"



(b) $A = A * B + C$

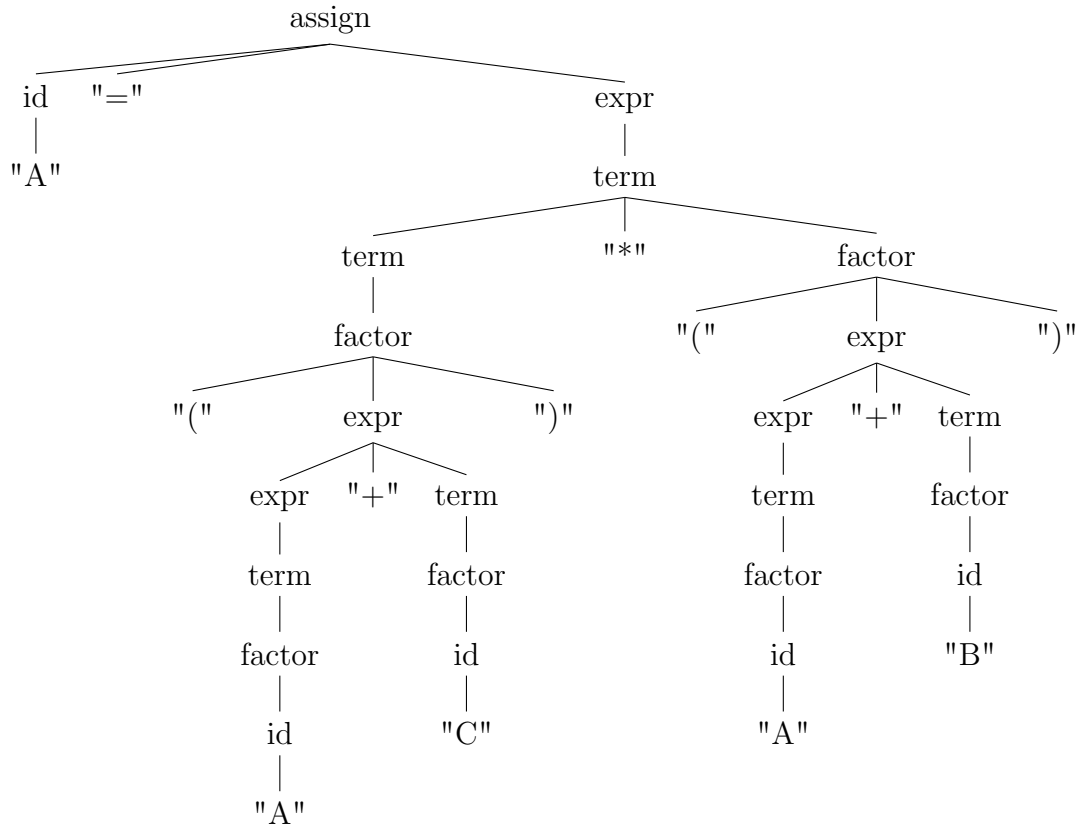
Leftmost derivation :

assign → id "=" expr
→ "A" "=" expr
→ "A" "=" expr "+" term
→ "A" "=" term "+" term
→ "A" "=" term "*" factor "+" term
→ "A" "=" factor "*" factor "+" term
→ "A" "=" id "*" factor "+" term
→ "A" "=" "A" "*" factor "+" term
→ "A" "=" "A" "*" id "+" term
→ "A" "=" "A" "*" "B" "+" term
→ "A" "=" "A" "*" "B" "+" factor
→ "A" "=" "A" "*" "B" "+" id
→ "A" "=" "A" "*" "B" "+" "C"



(c) $A = (A + C) * (A + B)$

assign → id "=" expr
→ "A" "=" expr → "A" "=" term
→ "A" "=" term "*" factor
→ "A" "=" factor "*" factor
→ "A" "=" "(" expr ")" "*" factor
→ "A" "=" "(" expr "+" term ")" "*" factor
→ "A" "=" "(" term "+" term ")" "*" factor
→ "A" "=" "(" factor "+" term ")" "*" factor
→ "A" "=" "(" id "+" term ")" "*" factor
→ "A" "=" "(" "A" "+" term ")" "*" factor
→ "A" "=" "(" "A" "+" factor ")" "*" factor
→ "A" "=" "(" "A" "+" id ")" "*" factor
→ "A" "=" "(" "A" "+" "C" ")" "*" factor
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" expr ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" expr "+" term ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" term "+" term ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" factor "+" term ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" id "+" term ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" "A" "+" term ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" "A" "+" factor ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" "A" "+" id ")"
→ "A" "=" "(" "A" "+" "C" ")" "*" "(" "A" "+" "B" ")"



Question 2

Write grammar for the Boolean expressions of Java, including following operators with precedence in descending order and associativity in this table:

Precedence	Operator	Description	Kind
1 (highest)	!	Logical NOT	Unary-Prefix-Right
2	== !=	Relational “equal to” and “not equal to”	Binary-Infix-None
3	$< \leq > \geq$	Relational "less than", "less than or equal to", "greater than" and "greater than or equal to"	Binary-Infix-None
4	||	Logical conditional-OR	Binary-Infix-Left
5 (lowest)	&&	Logical conditional-AND	Binary-Infix-Left

Explanation:

- Unary/Binary: Number of operands: one or two
- Prefix/Infix: Position of operator: before or in between its operands
- Right/None/Left: Association

Grammar:

expr	→	expr "&&" term | term
term	→	term "||" comp | comp
comp	→	equa ("<" | "≤ | ">" | "≥") equa | equa
equa	→	fact ("==" | "!=") fact | fact
fact	→	"!" fact | prim
prim	→	"false" | "true" | "(" expr ")"

Question 3

Convert the following EBNF to BNF

- EBNF :

s	→	a(Ba)*
a	→	A(B)?a

- BNF :

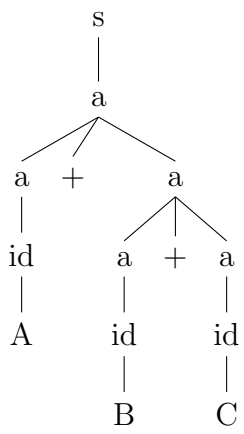
s	→	a | aBs
a	→	ABa | Aa

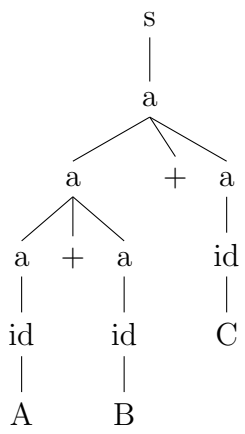
Question 4

(a) Prove that the following grammar is ambiguous:

s	→	a
a	→	a + a | id
id	→	A | B | C

Example: A + B + C, they have 2 or more parse trees. So this grammar is ambiguous.





(b) Find out what "Left recursion removal" means and perform the left recursion elimination for the above grammar

s	→	a
a	→	id bonus
bonus	→	+ a bonus | ε
id	→	A|B|C

Question 5

Find out what "Left factoring" means and perform left factoring for the following grammar

stmt	→	IF expr THEN {stmt} ELSE {stmt}
	|	IF expr THEN {stmt}
	|	other
expr	→	TRUE | FALSE

By left factoring we obtain :

stmt	→	IF expr THEN {stmt} rest
	|	other
rest	→	ELSE {stmt} | ε
expr	→	TRUE | FALSE

Question 6

Convert the BNF in Question 4 and 5 to EBNF

- Question 4 EBNF :

s	→	id(+id)*
id	→	A | B | C

- Question 5 EBNF :

stmt	→	IF expr THEN {stmt} (ELSE {stmt})?
expr	→	TRUE | FALSE

Extension question

Given the description of a program in mC as follows:

A program in mC consists of many declarations, which are variable and function declarations.

A variable declaration starts with a type, which is int or float, then a comma-separated list of identifiers and ends with a semicolon.

A function declaration also start with a type and then an identifier, which is the function name, and then parameter declaration and ends with a body. The parameter declaration starts with a left round bracket '(' and a null-able semicolon-separated list of parameters and ends with a right round bracket ')'. Each parameter always starts with a type and then a comma-separated list of identifier. A body starts with a left curly bracket '{', follows by a null-able list of variable declarations or statements and ends with a right curly bracket '}'.

There are 3 kinds of statements: assignment, call and return. All statements must end with a semicolon. An assignment statement starts with an identifier, then an equal '=', then an expression. A call starts with an identifier and then follows by a null-able comma-separated list of expressions enclosed by round brackets. A return statement starts with a symbol 'return' and then an expression.

An expression is a construct which is made up of operators and operands. They calculate on their operands and return new value. There are four kinds of infix operators: '+', '-', '*' and '/' where '+' have lower precedence than '-' while '*' and '/' have the highest precedence among these operators. The '+' operator is right associative, '-' is non-associative while '*' and '/' is left-associative. To change the precedence, a sub-expression is enclosed in round brackets. The operands can be an integer literal, float literal, an identifier, a call or a sub-expression.

For example,

```
int a, b, c ;
float foo(int a ; float c, d) {
    int e ;
    e = a + 4;
    c = a * d / 2.0;
    return c + 1 ;
}
float goo (float a, b) {
    return foo(1, a, b) ;
}
```

The following tokens can be used for the grammar:

ID (for identifiers), **INTLIT** (for integer literals), **FLOATLIT** (for float literals), **INT**, **FLOAT**, **RETURN**, **LB** (for ''), **RB** (for '') , **SM** (for ';'), **CM** (for ','), **EQ** (for '='), **LP** (for '('), **RP** (for ')'), **ADD** (for '+'), **SUB** (for '-'), **MUL** (for '*'), **DIV** (for '/')

a. Write the grammar of a program in mC in BNF format.

b. Write a recognizer in ANTLR to detect if a mC program is written correctly or not.

program	→	manydcls
manydcls	→	manydcls dcls | dcls
dcls	→	vardcls | funcdcls
vardcls	→	type idlist SM
type	→	INT | FLOAT
idlist	→	ID CM idlist | ID
funcdcls	→	type ID paradcls body
paradcls	→	LP paralist RP
paralist	→	para paratail |∈
paratail	→	SM para paratail |∈
para	→	type idlist
body	→	LB vardcl_stmt_list RB
vardcl_stmt_list	→	vardcl_stmt vardcl_stmt_list |∈
vardcl_stmt	→	vardcls | stmt
stmt	→	stmt_type SM
stmt_type	→	assign | call | return
assign	→	ID EQ exp
call	→	ID LP explist RP
explist	→	exp exptail |∈
exptail	→	CM exp exptail |∈
return	→	RETURN exp
exp	→	exp1 ADD exp | exp1
exp1	→	exp1 SUB exp1 | exp2
exp2	→	exp2 MUL exp3 | exp2 DIV exp3 | exp3
exp3	→	INTLIT | FLOATLIT | ID | call | subexp
subexp	→	LP exp RP