CS 333 : Compiler Design

TUTORIAL 3 (Grammars and Top down Parsing) : Released Monday February 05, 2024

The operators in C++ with their attributes given below are used in several problems on this sheet.

Precedence	Associativity	Arity	Operator	Function
16	L	binary	[]	array index
15	R	unary	++, --	increment, decrement
15	R	unary	~	bitwise NOT
15	R	unary	!	logical NOT
15	R	unary	+,	unary minus, plus
15	R	unary	*, &	dereference, address of
13	L	binary	*, /, %	multiplicative operators
12	L	binary	+, -	arithmetic operators
11	L	binary	<<, >>	bitwise shift
10	L	binary	<, <=, >, >=	relational operators
9	L	binary	==, !=	equality, inequality
8	L	binary	&	bitwise AND
7	L	binary	^	bitwise XOR
6	L	binary	|	bitwise OR
5	L	binary	&&	logical AND
4	L	binary	||	logical OR
3	L	ternary	?:	arithmetic if
2	R	binary	=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	assignment operators

Problems 1 through 7 are similar in nature. The more you practice these problems, your skills in grammar writing, applying grammar transformations, constructing FIRST)() and FOLLOW() sets, etc. will get sharpened.

P1. (a) Construct a parse tree for the following expression :

a += a *= a /= 5 + a && 4 || 10 − a

(b) Given that a has the value 5 at start, use your parse tree to determine the value of **a** after evaluation of the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the following operators {=, +=, *=, %=, | |, &&, binary +, binary –} and operands are tokens **num** and **id.**

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1)? Justify your answer.

(f) Parse the expression of part (a) using your parser and report the results.

P2. Consider the set of operators {=, +=, | |, &&, ==, !=, <, <=, >, >=, binary +} and operand set = {**num** ,**id }.**

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(**b)** Assume initial values for the **id's** used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ? Justify your answer.

(f) Parse the expression of part (a) using your parser and report the results.

P3. Consider the set of operators {=, | |, &&, ==, !=, binary +, binary –, *, /, %, unary –} and operand set = {**num** ,**id }. The * is the multiplicative operator.**

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id's** used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(**e)** Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P4. Consider the set of operators {=, | |, &&, ==, !=, <<, >>, bitwise and &, bitwise xor ^, bitwise or |, bitwise not ~, logical not !} and operand set = {**num** ,**id }.**

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id's** used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P5. Write an unambiguous expression grammar involving all the operators of precedence levels 4 to 6, 12 and 13. Eliminate left recursion and left factoring from your grammar, if possible.

P6. Consider the set of operators {=, ++, – –, dereference *, address of &, ==, !=, binary +, binary –, multiplicative *, multiplicative /} and operand set = {**num** ,**id }.**

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id's** used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator and operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P7. Consider the set of operators {=, ++, – –, dereference *, address of &, ==, !=, binary +, multiplicative *, array reference [ ] and ternary ? : } and operand set = {**num** ,**id }.**

(**a)** Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(**b)** Assume initial values for the **id's** used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(**e)** Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P8. Examine the following grammar for if-then-else ambiguity; assume the symbols if, then, else, expr to be terminals for this problem. In case the given grammar does not serve the required purpose, write an unambiguous grammar that works.

stmt → **if expr then** *stmt* | *matched_statement*
matched_statement → **if expr then** *matched_statement* **else** *stmt*

P9. Examine the following grammar for if-then-else. Assume that the symbols **{ if, then, else, expr, other}** to be terminals for this problem.

S → M | U
M → if expr then M else M | other
U → if expr then S | if expr then U else M

Consider the string **if expr then if expr then other else if expr then other else other**
Answer the following questions
(a) Find as many distinct leftmost derivations of the string as possible.

(b) Show the parse tree corresponding to each derivation of part (a) above
(c) Comment on ambiguity of this grammar ?

P10. Given the following specification of a language through regular expressions

variable → (*simple_or_array* **.**)* | *simple_or_array*
simple_or_array → id [*elist*] ?
elist → (e,) *e
where e is a terminal.
(a) Write a LL(1) grammar specifying the same language.
(b) Construct a LL(1) parsing table for the same grammar.

P11. Consider the following context free grammar.

S → X
X → Ma | bMc | dc | bda
M → d

(a) Determine whether the grammar as given is LL(1). Report all conflicts if the grammar is not LL(1).

(b) In case answer to (a) is no, Is it possible to rewrite this grammar io an equivalent LL(1)grammar ?

Write LL(1) parsers for the above grammar.

P12. Write a grammar for declarations in language C, that can generate one or more variables declared in the same statement and one or more declaration statements. The built-in types to be used are int and float and scalars only. Sample declarations are given below.

```
int a, b, c;
float x;
int * p,  q;
```

P13. Consider the following context free grammar. You have to determine whether the grammar is LL(k), where k =1 or 2. Construct and report the FIRST() and FOLLOW() sets. In case you find that the grammar is not LL(k), show any multiple-entry of the corresponding LL(k) table without constructing the entire table. In case you find that the grammar is indeed LL(k), argue it with respect to FIRST() AND FOLLOW() information.

1) S → G **id** 2) G → P G' 3,4) G' → G | ε 5) P → **id :** R
6) R → **id** R' 7,8) R' → R | **;**

P14. Identify which of the grammars given below is LL(1) and why ?

(a) S → A B c
A → a | ϵ
B → b | ϵ

(b) S → A B B A
A → a | ϵ
B → b | ϵ

(c) E → − E | (E) | V R
V → id T
R → −E | ϵ
T → (E) | ϵ

Write LL(1) parsers for each of the above grammars.

P15. Write a grammar for array declarations, which beside the declarations for scalar variables, also admits multi dimensional arrays of C. Sample declarations that are to be generated are :

```
int a, b[20], c;
float x, y[5][2];
int * p[2][2][5],  q;
```

********* End of Tutorial Sheet 3 *********