Imperial College London
Department of Computing

Compilers (221)

Exercises – LL Top Down Parsing

Check your answers with the tutorial helpers during tutorials and with each other on Piazza.

1.	Using EBNF write an LL(1) grammar for boolean expressions that consist of the constants **true** and **false**, parentheses **( )**, and the operators **and**, **or** and **not**. Ensure that: i) **or** has lower precedence than **and**, ii) **and** has lower precedence than **not**, iii) consecutive **not**'s *are* allowed, as in the expression **not not true**.	L2
2.	Suppose that an elevator is controlled by 2 commands: **up** to move the elevator up one floor, and **down** to move the elevator down one floor. Assume that the building is arbitrarily tall and that the elevator starts at floor **X**. i) Write an LL(1) grammar that recognises arbitrary command sequences that 1. never cause the elevator to go below floor **X**, and 2. always return the elevator to floor **X** at the end of the sequence, and For example, **up up down down** and **up down up down** and **up up down up down down** are valid command sequences, but **up down down up** is not. An empty (zero length) sequence is also valid. ii) Using the definition of LL(1) show that your grammar is LL(1).	L2
3.	Download the ANTLR examples from the website, 'compile' and run them. The makefile's will give various options. Think of some extensions and run them.	L2

4.	Consider the following grammar:
	
	`Method → method MethodName Block`
	`Block → '{' Sequence '}'`
	`Sequence → Statement | Sequence ';' Statement`
	`Statement → Declaration | Assignment | Call | IfStatement | WhileStatement | Return | Block`
	`Declaration → int Assignment`
	`Assignment → Variable '=' Expression`
	`Call → MethodName '(' ')'`
	`IfStatement → if '(' Expression ')' Statement | if '(' Expression ')' Statement else Statement`
	`WhileStatement → while '(' Expression ')' Statement`
	`Return → return Expression`
	`Expression → Expression Operator Operand | Operand`
	`Operand → Variable | integer | Call | '(' Expression ')'`
	`MethodName → identifier`
	`Variable → identifier`
	`Operator → '+' | '-' | '*' | '/' | '='`
	
	For this grammar identify the places that are not suitable for LL(1) parsing and then transform the grammar into a form that is suitable for LL(1) parsing. Use EBNF for your grammar and aim to produce a clear grammar that will produce a good AST, if necessary, by deleting rules or adding new rules.
		L4

5.	Now write parse functions for your LL(1) grammar for the previous question. Each function should return an appropriate AST object. You need not declare your AST classes. You do not need to perform error recovery.	L4

6.	In some programming languages the assignment operation, for example, :=, is allowed in expressions. The result of an assignment expression is the value of the right-hand side of the assignment which is also copied into the left-hand side of the assignment as a side effect. Consider the following grammar for such expressions: `Expr` → `ID ':=' Expr	Term TermTail` `Term` → `Factor FactorTail` `TermTail` → `'+' Term TermTail	ε` `Factor` → `'(' Expr ')'	ID` `FactorTail` → `'*' Factor FactorTail	ε` Explain why this grammar is not LL(1) and rewrite it to make it LL(1).	L2
7.	Consider the following grammar for an expression: `Expr` → `Operand	List` `List` → `'[' Seq ']'` `Seq` → `Expr ',' Seq	Expr` `Operand` → *num*	*id* i) Transform the grammar to LL(1). Give your answer in BNF not EBNF. ii) Derive the FIRST and FOLLOW sets for the non-terminals of your transformed grammar. Show your working. iii) Using the definition of LL(1) show that your transformed grammar is LL(1).	L3