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.

Let me know if you find a mistake or have a better answer.

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**.
	
	++
	
	`boolexpr → andexpr { or andexpr }`
	`andexpr → notexpr { and notexpr }`
	`notexpr → not notexpr | operand`
	`operand → true | false | '(' boolexpr ')'`
		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).
	
	++
	(i)
	
	`seq → move seq | ε` `seq → { move } in EBNF`
	`move → up seq down`
	
	(ii)
	First (seq) = { **up**, ε }
	Follow (seq) = **down,** $ First sets of alternatives of seq are disjoint, i.e. **up** and ε
	First(seq) and Follow(seq) are disjoint.
	First (move) = **up**
		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..	

4.	Consider the following grammar:
	
	`Method → method MethodName Block`
	`Block → '{' Sequence '}'`
	`Sequence → Statement | Sequence ';' Statement`
	`Statement → Declaration | Assignment | Call | IfStatement | WhileStatement |`
		L4

```
                     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.

++

Below is one possible LL(1) grammar.

```
Method           →   method identifier Block
Block            →   '{' Sequence '}'
Sequence         →   Statement { ';' Statement}
Statement        →   Declaration | identifier ( '=' Expression | '(' ')' ) |
                     IfStatement | WhileStatement | Return | Block
Declaration      →   int identifier '=' Expression
IfStatement      →   if '(' Expression ')' Statement [else Statement]
WhileStatement   →   while '(' Expression ')' Statement
Return           →   return Expression
Expression       →   Operand { Operator Operand }
Operand          →   identifier [ '('  ')' ]  | integer | '(' Expression ')'
Operator         →   '+' | '-' | '*' | '/' | '='
```

Although we can write parse functions for this grammar, let's see if we can tidy up the grammar a bit before doing so. In particular the substitutions of `MethodName` and `Variable` give rise to some ugliness. So we'll create new rules `AssignCall` and `VarCall`, e.g. :

```
Statement        →   Declaration | AssignCall | IfStatement | WhileStatement | Return |
                     Block
AssignCall       →   identifier ( '=' Expression | '(' ')' )
Operand          →   VarCall | integer | '(' Expression ')'
VarCall          →   identifier [ '('  ')' ]
```

Although the above is better, we can go one step further and re-introduce slightly different rules for Assignment and method Call:

```
AssignCall       →   identifier (Assignment | Call)
Assignment       →   '=' Expression
Call             →   '(' ')'
```

This is nearer to the original and by passing *identifier* to the parse functions for `Assignment` and `Call` we get a clearer AST. The grammar could also be rewritten to handle operator precedence.

Although the strategies above keep the grammar as LL(1) there is a much simpler practical solution – use LL(2) or if writing the parser by hand just peek ahead and make the decision based on the next token, for example, is the next token '=' or '('.

5.	Now write parse functions for your LL(1) grammar for the previous question. Each function should return an appropriate AST object. You do not need to declare your AST classes nor perform error recovery.	L4

++

```
def Statement():
  switch (token):
```

```
      INT:        return Declaration()
      IDENTIFIER: return AssignCall()
      IF:         return IfStatement()
      WHILE:      return WhileStatement()
      RETURN:     return ReturnStatement()
      LBRACE:     return Block()
      else:       error()

def Method():
  match(METHOD)
  methodname = token   -- could extract string from token
  match(IDENTIFIER)
  return MethodAST(methodname, Block())

def Block():
   -- similar to begin statement in lecture slides
   ...
  return BlockAST(seq)

def Declaration():
  match(INT)
  variable = token
  match(IDENTIFIER)
  match(EQUALS)
  return DeclarationAST(variable, Expression())

def IfStatement():
 -- similar to lecture slides

def Expression():
  expr = Operand()
  while token in {PLUS, MINUS, STAR, DIVIDE, EQUAL}:
      operator = Operator()
      expr = ExpressionAST(expr, operator, Expression())
   return expr

def Operator():
  op = token
  match(token)
  return OperatorAST(op)   # Yuck!

def AssignCall():
  if token == IDENTIFIER: id = token
  match(IDENTIFIER)
  if token == EQUALS:
      return AssignmentAST(id,Expression())
  elif token == LPAREN:
      match(LPAREN)
      match(RPAREN)
      return CallAST(id)
  else error()
```

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: L2

```
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).

++

The grammar isn't LL(1) because ID is in the first set of `ID := Expr` and `Term TermTail` i.e. the two alternatives are not disjoint. We can rewrite by substituting `Factor` into `Term` for `Expr` and then left-factor `ID` giving:

```
Expr     → ID VarTail | '(' Expr ')' FactorTail TermTail
VarTail  → ':=' Expr | FactorTail TermTail
Other rules are as before
```

7. Consider the following grammar for an expression: L3

```
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).

++

(i) We only need to left factor Seq

```
Expr      → Operand | List
List      → '[' Seq ']'
Seq       → Expr  Rest
Rest      → ','  Seq | ε
Operand   → num | id
```

(ii)

```
FIRST (Expr)     = FIRST (Operand) + FIRST (List) = {num, id} + {[} = {num, id, [}
FIRST (List)     = {[}
FIRST (Seq)      = FIRST (Expr) = {num, id, [}
FIRST (Rest)     = {','} + {ε} = {',', ε}
FIRST (Operand)  = {num, id}
FOLLOW (Expr)    = FIRST (Rest) + FOLLOW(Rest) + {$} = {',', ], $}
FOLLOW (List)    = FOLLOW (Expr) + {$}   = {',', ], $}
FOLLOW (Seq)     = {]} + FOLLOW (Rest)   = { ] }
FOLLOW (Rest)    = FOLLOW (Seq) = { ] }
FOLLOW (Operand) = FOLLOW (Expr) = {',', ], $}
```

(iii) Consider rules with alternatives

```
Expr    →  Operand | List         FIRST (Operand) and FIRST (List) are disjoint
Rest    →  ','  Seq | ε           FIRST (',' Seq) and FIRST (ε) are disjoint
                                  FIRST (Rest) and FOLLOW (Rest) are disjoint
Operand → num | id                FIRST (num) and FIRST (id) are disjoint
```