Omar Ahmadi Section 05

CS 314 Homework 3

1. A nonterminal with one production rule can't undo the LL(1) property. The nonterminals with multiple productions are <morestmts>, <stmt>, <expr>, <variable> and <digit>.

   LL(1) property – Anytime the grammar allows a choice of multiple productions for a single non-terminal symbol, the FIRST+ sets of the right hand sides of the productions for the respective non-terminal symbol have to be pairwise disjoint. This allows a deterministic selection among the rules using a single input look ahead symbol.

   <digit> FIRST sets: {0} INTERSECT {1} = { }, {0} INTERSECT {2} = { }, …
   <variables> FIRST sets: {a}, {b}, and {c}.
   <expr> FIRST sets: {+}, {*}, {a, b, c} and {0, 1, 2}.
   <stmts> FIRST sets: {a, b, c}, {if}, {while} and {begin}.

   All of the above FIRST sets of the productions right hands sides are pairwise disjoint. We must make a decision for the right hand sides for the two productions of the nonterminal <morestmts>, FIRST(; <stmtlist>) = {;}. Since <morestmts> :: ε look at the FOLLOW set of <morestmts> and compute FIRST+. Every symbol that is in FOLLOW (<stmtlist>) has to be added to FOLLOW (<morestmts>) because of the rule <stmtlist> ::= <stmt><morestmts>. FOLLOW (<stmtlist>) must contain "end" because of the rule <block> ::= begin <stmtlist> end. FOLLOW (<stmtlist>) = {end}. as a result, FOLLOW (<morestmts>) = {end}. FIRST + (ε) = {end} for the right hand side ε for nonterminal symbol <morestmts>. We can make a deterministic decision to pick a single rule for nonterminal <morestmts> because the sets {;} and {end} are pairwise disjoint. Finally we can conclude that the grammar is LL(1).

2. Show the LL(1) parse table

NT/T	program	begin	end	;	if	then	else	while	do
<program>	program <block>								
<block>		begin <stmtlist> end							
<stmlist>		<stmt> <morestmts>			<stmt> <morestmts>			<stmt> <morestmts>	
<morestmts>				; <stmtlist>					
<stmt>		<block>			<ifstmt>			<whilestmt>	
<assign>									
<ifstmt>					If<testexpr> then<stmt> else<stmt>				
<whilestmt>								While <testexpr> do<stmt>	
<testexpr>									
<expr>									
<variable>									
<digit>									

NT/T	<=	+	*	;	a | b | c	0 | 1 | 2	end	EOF
<program>					<stmt><morestmts>			
<block>								
<stmlist>								
<morestmts>							epsilon	
<stmt>					<assign>			
<assign>					<variable>=<expr>			
<ifstmt>								
<whilestmt>								
<testexpr>					<variable> <= <expr>			
<expr>		+ <expr> <expr>	* <expr> <expr>		<variable>	<digit>		
<variable>					a | b | c			
<digit>						0 | 1 | 2		

3. Assume next_token() exists, global variable token exists and is initialized before entering the function representing the nonterminal program.
4. Code for problem 4 is bolded.

```
program {
        #asgn := 0;
        #add := 0;
        #mult := 0;
        switch token {
                case “program”:
                        token := next_token();
                        call block();
                        if “.” != token{
                                error();
                        }
                        token := next_token();
                        if eof != token {
                                error();
                        }
                        break;
                default:
                        error();
        }
        print(%d assignments, %additions, %multiplications), #asgn, #add, #mult);
}

block{
        switch token{
                case “begin”:
                        token := next_token();
                        call stmtlist();
                        if “end” != token
                        {
                                error();
                        }
                        token := next_token();
                        break;
                default:
                        error();
        }
}
```

```
stmtlist{
        switch token{
                case a: case b: case c: case “if”: case “while”: case “begin”:
                        call stmt();
                        call morestmts();
                        break;
                default:
                        error();
        }
}

morestmts{
        switch token{
                case “;”
                        token := next_token();
                        call stmtlist();
                        break;
                case “end”;
                        break;
                default:
                        error();
        }
}

stmt{
        switch token{
                case a: case b: case c:
                        call assign();
                        break;
                case “if”:
                        call ifstmt();
                        break;
                case “while”:
                        call whilestmt();
                        break;
                case “begin”:
                        call block();
                        break;
                default:
                        error();
        }
}
```

```
assign{
        switch token{
                case a: case b: case c:
                        #asgn = #asgn + 1;
                        call expr();
                        if “+” != token || “*” != token{
                                error();
                        }
                        token := next_token();
                        call expr();
                        break;
                default:
                        error();
        }
}

ifstmt{
        switch token{
                case “if”
                        token := next_token();
                        call testexpr();
                        if “then” != token{
                                error();
                        }
                        token := next_token();
                        call stmt();
                        if “else” != token{
                                error();
                        }
                        token := next_token();
                        call stmt();
                        break;
                default:
                        error();
                }
}
```

```
whilestmt{
        switch token{
                case “while”:
                        token := next_token();
                        call testexpr();
                        if “do” != token{
                                error();
                        }
                        token := next_token();
                        call stmt();
                        break;
                default:
                        error();
        }
}

testexpr{
        switch token{
                case a: case b: case c:
                        call variable();
                        if “<=” != token{
                                error();
                        }
                        token := next_token();
                        call expr();
                        break;
                default:
                        error();
        }
}

expr{
        switch token{
                case +: case *:
                        token := next_token();
                        call expr();
                        call expr();
                        break;
                case a: case b: case c:
                        call variable();
                        break;
                case 0: case 1: case 2:
                        call digit();
                        break;
                default:
                        error();
```

```
        }
}

variable
{
        switch token{
                case a: case b: case c:
                        #refs = #refs + 1;
                        token := next_token();
                        break;
                default:
                        error();
        }
}

digit
{
        switch token{
                case 0: case 1: case 2:
                        token := next_token();
                        break;
                default;
                        error();
        }
}
```