CS314 Spring 14
HW #3 Sample Solution

1.1: LL(1) property -- whenever 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 that non-terminal symbol have to be pairwise disjoint. This allows a deterministic selection among the rules using only a single input look ahead symbol.

Any nonterminal with only one production rule cannot break the LL(1) property. The nonterminals with multiple productions are <morestmts>, <stmt>, <expr>, <variable>, and, <digit>.

<digit>'s FIRST sets of its productions' right hand sides are pairwise disjoint:
{0} ∩ {1} = { }, {0} ∩ {2} = { }, …

<variables>'s FIRST sets of its productions right hand sides are pairwise disjoint: They are {a}, {b}, and {c}.

<expr>'s FIRST sets of its productions right hand sides are pairwise disjoint: They are {+}, {*}, {a, b, c} and {0, 1, 2}.

<stmts>'s FIRST sets of its productions right hand sides are pairwise disjoint: They are {a, b, c}, {if}, {while} and {begin}.

The tricky part is to show that we can make a unique decision for the right hand sides for the two productions of the nonterminal <morestmts>. FIRST(; <stmtlist>) = {;}, so that's not a problem. However, since <morestmts> :: ε , we need to look at the FOLLOW set of <morestmts>, i.e., we have to compute FIRST+. Due to the rule
<stmtlist> ::= <stmt> <morestmts>, every symbol that is in FOLLOW(<stmtlist>) has to be added to FOLLOW(<morestmts>). Due to the rule
<block> ::= begin <stmtlist> end, FOLLOW(<stmtlist>) has to contain symbol "end". In fact, FOLLOW(<stmtlist>) = {end}. Therefore, FOLLOW(<morestmts>) = {end} as well.
So, FIRST+(ε) = {end} for the right hand side ε for nonterminal symbol <morestmts>. Since the sets {;} and {end} are pairwise disjoint, we can make a deterministic decision to pick a single rule for nonterminal <morestmts>.

Therefore, our grammar is LL(1).

1.2:

All empty cells are error states.

Parse table is listed in two parts.
Part 1:

NT\T	program	begin	end	;	if	then	else	while	do
<program>	**program <block>**								
<block>		**begin <stmtlist> end**							
<stmtlist>		<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>									

Part 2:

NT\T	<=	+	*	;	a	b	c	0	1	2	end	EOF
<program>					<stmt> <morestmts>							
<block>												
<stmtlist>												

<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**		

1.3 / 1.4:

Assume global variable token exists and that it is initialized before entering the function representing the nonterminal program. Also assume the procedure next_token() exists.

One possible solution for 1.4 uses global integer variables to keep track of the number of assignments (#asgn) and numbers of references (#refs).

Added code for problem 1.4 is shown in bold face.

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

}

block
```

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

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

morestmts
{
        switch token
        {
                case “;”:
                        token := next_token();
                        call stmtlist();
                        break;
                case “end”:  // this is how you handle epsilon productions
                        break;
                default:
                        error; die;
        }
}

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; die;
        }
}

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

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

```
                break;
        default:
                error; die;
    }
}

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

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

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; die;
        }
}

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

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