HW3 CS 314

1.

LL(1) parsing definition

(A ::= α and A ::= β) implies F IRST +(α) ∩ F IRST +(β) = ∅

Therefore, we need to check First+ set for non-terminal symbol that have 2 rules, and its rules are mutually disjoint.

- we need to check <morestmts>, <stmt>, <expr>, <var>, <digit>

1. <program>::=prog<block>.
2. <block>::=begin<stmtlist>end
3. <stmtlist>::=<stmt><morestmts>
4. <morestmts>::= ;<stmtlist> |
5. ε
6. <stmt>::= <assign> |
7. <ifstmt>|
8. <repeatstmt>|
9. <block>
10. <assign>::= <var> = <expr>
11. <ifstmt>::= if <testexpr> then <stmt> else <stmt>
12. <repeatstmt> ::= repeat<stmt> until <testexpr>
13. <testexpr> ::= <var> <= <expr>
14. <expr>::= +<expr><expr> |
15. -<expr><expr> |
16. *<expr><expr> |
17. <var> |
18. <digit>
19. <var>::=a |
20. b |
21. c |

22. <digit>::= 0 |

23. 1 |

24. 2 |

<morestmts>

rule 4:

First+(;<stmtlist>) = First+(;) = First(;) = {;}

rule 5:

First+(ε) = { ε }-{ ε }+Follow(<morestmts>)

= {end}

Therefore, First+ set are disjoint

<stmt>

rule 6:

First+(<assign>) = First+(<var>) = First(<var>) = {a,b,c}

rule 7:

First+(<ifstmt>) = First+(if) = {if}

rule 8:

First+(<repeatstmt>) = First+(repeat) = {repeat}

rule 9:

First+(<block>) = First+(begin) = {begin}

Therefore, First+ set are disjoint

<expr>

rule 14:

First+ (+<expr><expr>)= First+(+) = {+}

rule 15:

First+ (-<expr><expr>)= First+(-) = {-}

rule 16:

First+ (*<expr><expr>)= First+(*) = {*}

rule 17:

First+ (<var>)= First(<var>) = {a,b,c}

rule 18:

First+ (<digit>)= First(<digit>) = {0,1,2}

Therefore, First+ set are disjoint

<var>

rule 19:

First+(a) = {a}

rule 20:

First+(b) = {b}

rule 21:

First+(c) = {c}

Therefore, First+ set are disjoint

<digit>

rule 22:

First+(0) = {0}

rule 23:

First+(1) = {1}

rule 24:

First+(2) = {2}

Therefore, First+ set are disjoint

Therefore, this grammar is LL(1)

2.parse table

<program>

rule 1:

First+(prog <block> .) = First+(prog) = {prog}

<block>

rule 2:

First+ (begin<stmtlist>end) = First+(begin) = {begin}

<stmtlist>

rule 3:

First+ (<stmt><morestmts>) = First+(<stmt>) = First+(<assign>) + First+(<ifstmt>) + First(<repeatstmt>)+ First(<block>)

= {a,b,c,if,repeat,begin}

<assign>

rule 10:

First+(<var>=<expr>) = First+(<var>) = {a,b,c}

<ifstmt>

rule 11:

First+ (if<testexpr>then<stmt>else<stmt>)= First+(if) = {if}

<repeatstmt>

rule 12:

First+ (repat<stmt>until<testexpr>) = First+(repeat) = {repeat}

<testexpr>

rule 13:

First+(<var><=<expr>) = First+ (<var>) = {a,b,c}

	prog	begin	end	;	if	then	else	repeat	until
Program	1								
Block		2							
Stmtlist		3			3			3	
Morestmts			5	4					
Stmt		9			7			8	
Assign									
Ifstmt					11				
Repeatstmt								12	
Testexpr									
expr									
Var									
digit									

	<=	+	-	*	=	a	b	c	0	1	2	.	eof
Program													
Block													
Stmtlist						3	3	3					
Morestmts													
Stmt						6	6	6					
Assign						10	10	10					
Ifstmt													
Repeatstmt													
Testexpr						13	13	13					
expr		14	15	16		17	17	17	18	18	18		
Var						19	20	21					
digit									22	23	24		

3+4 (red line is for #4)

```
main() {
    int num_biop = 0;


    token := next_token();
    call program();
    if (token == eof) {
        print (‘number of binary operators:’+ num_biop)
        accept;
    }else{
      error;
    }
}


program(){
  switch token{
    case 'prog':
      token := next_token();
      call block()
      if token == ‘.’{
        token := next_token();
        break;
      }else{
        error;
        exit;
      }
    default:
      error; exit;
  }
```

```
}

block(){
  switch token{
    case 'begin':
      token := next_token();
      call stmtlist();
      if token == ‘end’{
        token := next_token();
        break;
      }else{
        error;
        exit;
      }
    default:
      error; exit;
  }
}

stmtlist(){
  switch token{
    case 'begin':
    case 'if':
    case 'repeat':
    case 'a':
    case 'b':
    case 'c':
      call stmt();
      call morestmts();
      break;
    default:
      error; exit;
  }
}

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

stmt(){
  switch token{
    case 'begin':
      call block();
      break;
    case 'if':
      call ifstmt();
      break;
    case 'repeat':
      call repeatstmt();
      break;
    case ‘a':
     case ‘b':
     case ‘c':
       call var();
      break;
    default:
      error; exit;
  }
}

assign(){
  switch token{
    case 'a':
    case 'b':
    case 'c':
      call var();
      if token == ‘=’{
        token := next_token();
        call expr();
        break;
      }else{
        error;
        exit;
      }
    default:
      error; exit;
  }
}
```

```
ifstmt(){
  switch token{
    case 'if':
      token := next_token();
      call testexpr();
      if token != ‘then’{
        error; exit;
      }
      token := next_token();
      call stmt();
      if token != ‘else’{
        error; exit;
      }
      token := next_token();
      call stmt();
      break;
    default:
      error; exit;
  }
}

repeatstmt(){
  switch token{
    case 'repeat':
      token := next_token();
      call stmt();
      if token != 'until’{
        error; exit;
      }
      token := next_token();
      call testexpr();
      break;
    default:
      error; exit;
  }
}

testexpr(){
  switch token{
    case 'a':
    case 'b':
    case 'c':
      call var();
      if token != '<=' {
         error; exit;
      }
      token := next_token();
      call expr();
      num_biop++;
      break;
    default:
      error; exit;
  }
}

expr(){
  switch token{
```

```
  case '+':
  case '-':
  case '*':
   token := next_token();
   call expr();
   call expr();
   num_biop++;
   break;
  case 'a':
  case 'b':
  case 'c':
   call var();
   break;
  case '0':
  case '1':
  case '2':
   call digit();
   break;
  default:
   error; exit;
 }
}

var(){
  case 'a':
  case 'b':
  case 'c':
   token:= next_token();
   break;
```

```
    default:
      error; exit;
  }
}

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