CS314 Homework 3

Sample Solution

Spring 2023

1 Problem — LL(1) Recursive Descent Parsing

1: <program> ::= **program** <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: <variable> |
18: <digit>
19: <var> :: a |
20: b |
21: c
22: <digit> :: 0 |
23: 1 |
24: 2

1. LL(1)?

A grammar is LL(1) iff, for each Non-Terminal, the FIRST+ sets of the right-hand-sides of all its rules are mutually disjoint.
Non-terminal symbols with only 1 rulle (rhs) won't hurt the LL(1) property.
So only those with multiple rules will be considered here to prove the LL(1) property of the grammar. However, parse tables typically include the FIRST+ sets of Non-Terminal symbols even in the case of a single rule. This helps in detecting problems more quickly.
The Non-Terminals with multipe rules are:
<morestmts>, <stmt>, <expr>, <var>, <digit>

<morestmts>
rule 4:
FIRST+(; <stmtlist>) = { ; }
rule 5:
FIRST (epsilon) = {epsilon}
FIRST+(epsilon) =
   FIRST(epsilon) - {epsilon} + FOLLOW(<morestmts>) = { end }

FOLLOW(<morestmts>) = FOLLOW(<stmtlist>) = { end }

FIRST+ sets are disjoint

<stmt>
rule 6: FIRST+(<assign>) = FIRST<variable> = {a,b,c}
rule 7: FIRST+(<ifstmt>) = { if }
rule 8: FIRST+(<repeatstmt>) = { repeat }
rule 9: FIRST+(<block>) = { begin }

FIRST+ sets are disjoint

<expr>
rule 14:FIRST+(+<expr><expr>) = {+}
rule 15:FIRST+(-<expr><expr>) = {-}
rule 16:FIRST+(*<expr><expr>) = {*}
rule 17:FIRST+{<var>} = {a,b,c}
rule 18:FIRST+(<digit>) = {0,1,2}

FIRST+ sets are disjoint

<variable>
rule 19:FIRST+(a) = {a}
rule 20:FIRST+(b) = {b}
rule 21:FIRST+(c) = {c}

FIRST+ sets are disjoint

<digit>
rule 22:FIRST+(0) = {0}
rule 23:FIRST+(1) = {1}
rule 24:FIRST+(2) = {2}

FIRST+ sets are disjoint

2.Parse table

Here are the remaining FIRST+ sets for Non-Terminal symbols with single right-hand-sides.
NOTE: We could set up the parse table to always choose the rule without looking at the input symbol. If the input does not match, eventually this mismatch will be discovered when trying to compare the token on top of the stack with the next input symbol.

```
<program>
   rule1: FIRST+(program <block> .) = { program }
<block>
   rule2: FIRST+(begin <stmtlist> end) = { begin }
<stmtlist>
   rule3: FIRST+(<stmt> <morestmts>) = FIRST+(<stmt>) = FIRST+(<assign>) + FIRST+(<ifstmt>) +
                                                        FIRST+(<repeatstmt>) + FIRST+(<block>)
                                                      = { a, b, c, if, repeat, begin }
<assign>
   rule10: FIRST+(<var> = <expr>) = FIRST+(<var>) = { a, b, c }
<ifstmt>
   rule11: FIRST+(if <testexpr> then <stmt> else <stmt>) = { if }
<whilestmt>
   rule12: FIRST+(repeat <stmt> until <testexpr>) = { repeat }
<testepxr>
   rule13: FIRST+(<var> <= <expr>) = FIRST+(<var>) = { a, b, c }
```

Table 1: Parse Table

NT/T	**program**	**begin**	**end**	**;**	**if**	**then**	**else**	**repeat**	**until**	**<=**	**+**	**−**	**∗**	**=**	**a**	**b**	**c**	**0**	**1**	**2**	**.**	**eof**
program	1																					
block		2																				
stmtlist		3			3			3							3	3	3					
morestmts			5	4																		
stmt		9			7			8							6	6	6					
assign															10	10	10					
ifstmt					11																	
repeatstmt								12														
testexpr															13	13	13					
expr											14	15	16		17	17	17	18	18	18		
var															19	20	21					
digit																		22	23	24		

3/4.Recursive descent parser / Syntax directed translation

```
// syntax-directed translation code for part 3 and part 4)

main{} {
   integer number_of_binary_operators = 0;

   token = next_token();
   number_of_binary_operators = program();
   if token == eof {
     PRINT (number_of_binary_operators + "binary operators"); /* part4 */
     accept;
   else
     error;
}

int program() /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case 'program:
            token := next_token();
            binary_ops = block(); /* part 3 & 4 */
            if '. != token  // period
            {
```

```
                error; exit;
            }
            token := next_token();
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int block() { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case 'begin:
            token := next_token();
            binary_ops = stmtlist(); /* part 3 & 4 */
            if 'end != token
            {
                error; exit;
            }
            token := next_token();
            break;
        default:
            error;
    }
    return binary_ops; /* part4 */
}

int stmtlist() { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case 'a:  case 'b: case 'c:
        case 'if: case 'while:
        case 'begin:
            binary_ops = stmt(); /* part 3 & 4 */
            binary_ops = binary_ops + morestmts(); /* part 3 & 4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int morestmts() { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case '; :
            token := next_token();
            binary_ops = stmtlist(); /* part 3 & 4 */
            break;
        case 'end: // epsilon production
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int stmt()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token  {
        case 'a: case 'b: case 'c:
            binary_ops = assign(); /* part 3 & 4 */
```

```
            break;
        case 'if:
            binary_ops = ifstmt(); /* part 3 & 4 */
            break;
        case 'repeat:
            binary_ops = repeatstmt(); /* part 3 & 4 */
            break;
        case 'begin:
            binary_ops = block(); /* part 3 & 4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int assign()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case 'a: case 'b: case 'c:
            call var();
            if '= != token
            {
                error; exit;
            }
            token := next_token();
            binary_ops = expr(); /* part 3 & 4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int ifstmt()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token {
        case 'if:
            token := next_token();
            binary_ops = testexpr(); /* part 3 & 4 */
            if 'then != token
            {
                error; exit;
            }
            token := next_token();
            binary_ops = binary_ops + stmt(); /* part 3 & 4 */
            if 'else != token
            {
                error; exit;
            }
            token := next_token();
            binary_ops = binary_ops + stmt(); /* part 3 & 4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int repeatstmt()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token  {
        case 'repeat:
```

```
            token := next_token();
            binary_ops = stmt(); /* part 3 & 4 */
            if 'until != token
            {
                error; exit;
            }
            token := next_token();
            binary_ops = binary_ops + testexpr(); /* part 3 & 4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int testexpr()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token  {
        case 'a: case 'b: case 'c:
            call var();
            if '<= != token /* THIS IS A BINARY OPERATOR */
            {
                error; exit;
            }
            token := next_token();
            binary_ops = expr(); /* part 3 & 4 */
            binary_ops++; /* part4 */
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

int expr()  { /* part 3 & 4 */
    int binary_ops = 0; /* part4 */
    switch token  {
        case '+: case '-: case '*: /* THIS IS A BINARY OPERATOR */
            token := next_token();
            binary_ops = expr(); /* part 3 & 4 */
            binary_ops = binary_ops + expr(); /* part 3 & 4 */
            binary_ops++; /* part4 */
            break;
        case 'a: case 'b: case c:
            call var();
            break;
        case '0: case '1: case '2:
            call digit();
            break;
        default:
            error; exit;
    }
    return binary_ops; /* part4 */
}

var()  {
    switch token {
        case 'a: case 'b: case 'c:
            token := next_token();
            break;
        default:
            error; exit;
    }
```

```
}

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