实 验 四 归下降预测分析程序

content

一、翻译模式……2

二、 自顶向下 vs. 自底向上……4

1、 程序调试……4

2、 处理语言范围……5

3、 与语法制导翻译结合……5

4、 error recovery……5

5、 表格驱动……5

6、 实现速度……5

一、翻译模式

```
module -->          "MODULE" ID1 ";"
                    declaration
                    module_begin
                    "END" ID2"."
                    {if(ID1 != ID2) throw moduleNameMismatched;}

module_begin -->    "BEGIN" statement_sequence

declaration -->     const_declare type_declare var_declare
                    procedure_declaration
const_declare -->   empty | "CONST" const_list
const_list -->      empty | ID "=" expression ";" const_list

type_declare -->    empty | "TYPE" type_list
type_list -->       empty | ID "=" type_id ";" type_list

var_declare -->     empty | "VAR" var_list
var_list -->        empty | id_list ":" type_id ";" var_list

procedure_declaration -->  empty | procedure_heading ";"
                                 procedure_body ";"
                                 procedure_declaration

procedure_body -->  declaration procedure_being "END" ID
procedure_begin --> empty | "BEGIN" statement_sequence

procedure_heading   -->"PROCEDURE" ID formal_parameters

formal_parameters   --> empty | "(" fp_section ")"

fp_section -->      var_if id_list ":" type_id |
                    var_if id_list ":" type_id ";" fp_section
var_if -->          "VAR" | empty
type_id -->         ID| array_type | record_type |
                    "INTEGER" | "BOOLEAN"

record_type -->     "RECORD" field_list "END"

field_list -->      field_one ";" field_list | field_one
field_one -->       empty | id_list ":" type_id

array_type -->      "ARRAY" expression "OF" type_id
```

```
                    {array_type.type = ARRAY}

id_list1 -->        ID
                    {id_list1 += ID}
                    | ID "," id_list2
                    {id_list2 = id_list1 + ID}

statement_sequence  --> statement | statement ";" statement_sequence

statement -->       empty | assignment | procedure_call |
                    if_statement | while_statement | rw_statement

while_statement --> "WHILE" expression "DO" statement_sequence "END"
                    {if(expression.type != BOOLEAN)
                              throw TypeMismatched}

if_statement -->    "IF" expression "THEN" statement_sequence
                    elsif_statement else_statement
                    {if(expression.type != BOOLEAN)
                              throw TypeMismatched}
elsif_statement --> empty |
                    "ELSIF" expression "THEN" statement_sequence
                    elsif_statement
els_statement -->   empty | "ELSE" statement_sequence "END"

rw_statement -->    "READ" "LPAREN" ID "RPAREN" |
                    "WRITE" "LPAREN" ID "RPAREN" |
                    "WRITELN" "LPAREN" ID "RPAREN" |
                    "WRITELN" "LPAREN" "LPAREN"

procedure_call -->  ID actual_parameters

actual_parameters   --> empty | "(" ")" | "(" ap_list ")"

ap_list1 -->        expression
                    {ap_list1 += expreesion.type}
                    | expression "," ap_list2
                    {ap_list2 = ap_list1 + expression.type}

assignment -->      ID selector ":=" expression
                    {
                    if (selector.type == null)
                     {
                        if (ID.type != expression.type)
```

```
                        throw mistype;
                    }
                    else
                    {
                        if (selector.type != expression.type)
                            throw mistype;
                    }
                    }

expression -->      simple_expression1 re_op simple_expression2
                    {if (simple_expression1.type != INTEGER ||
                            simple_expression2.type != INTEER)
                            throw mistype;
                    }
                    | simple_expression
re_op -->           "=" | "#" | "<" | "<=" | ">" | ">="

simple_expression   -->term | term low_op simple_expression
low_op -->          "+" | "-" | "OR"

term -->            factor | factor high_op term
high_op -->         "*" | "DIV" | "MOD" | "&"

factor -->          ID selector
                    {factor.type = selector.type}
                    | NUMBER
                    {factor.type = INTEGER}
                    |"(" expression ")"
                    {factor.type = expression.type}
                    | "~" factor1;
                    {if (factor1.type != BOOLEAN)
                        throw mistype;
                     else
                        factor.type = BOOLEAN;}
```

二、自顶向下 vs. 自底向上

1、程序调试

两者在调试方面没有多大的区别，但是因为实验三结合了 cup 工具的使用，很多语义的结合不是很熟练，导致了程序的调试也不太容易，当然这是针对本人来说的。从理论上，两者的调试应该不会相差太多。自顶向下的程序结构因为是看到了 lookahead 就采取程序的动作，所以不需要一个栈去保留那些 token，在调试方面也不需要更多的

信息去进行调试，所以更加适合于调试。当然这只是一点优势而已。

2、处理语言范围

自顶向下可以处理的是 LL（1）文法的语言，文法的属性属于继承属性的；而自底向上处理的语言，既可以是综合属性的，也可以使继承属性的。所以自底向上处理语言的范围更大，选择的空间也 LR(0), LR(1), SLR(1), LALR(1)的选择，这几种当中，最强大的是 LR(1)，而 LR(0)和 SLR(1)的报错会比较晚一些。

3、与语法制导翻译结合

自顶向下的必须每一次都把继承属性用参数的形式传到下一层函数中，而自底向上中，是用函数的返回值，将综合属性利用函数的返回值返回。所以在自顶向下当中，必须考虑参数与需要继承的属性的对应和参数的个数等；自底向上中，返回值只有一个，必须将全部需要用到的综合属性利用一个返回值传回，这个可以用一个 object 返回，里面包括需要用到的属性信息。

4、error recovery

自底向上在这方面能够更快发现语法错误，但是这只是一个错误的发现，如果需要 error recovery 的话，必须将当前部分 token 在栈中弹出，再继续程序的分析过程。

在自顶向下分析中，如果发现了错误，出来将栈顶的 token 弹出之外，还必须将一些还正在返回的 token 弹出，直到可以进行下一次正确的分析。所以自顶向下的错误分析必须计算好需要忽略的 token，弹出，再进行分析。

在自底向上的分析中，如果发现了错误，将栈中还没有进行规约的 token 弹出，然后继续进行分析，所以在自底向上的分析中，error recovery 比较容易操作。

5、表格驱动

每一个非终结符看到一个终结符采取的动作，如果语言的体系不是那么大，比如说这个 oberon，利用表格驱动也不失为一个好办法。空间因素的影响应该是比较小的，如果不特别考虑空间的要求。

6、实现速度

自顶向下的实现速度比自底向上的慢，这是因为前者是看到一个 token 就开始了动作的执行，而后者是看到了若干 token 才决定是否 reduce 或者 shift。