实验四、手工编写递归下降预测分析程序

1. 设计 Oberon-0 语言的翻译模式

   事实上在实验三中的cup文件中，我们以及在文法定义中规定了我们的翻译模式。但是由于在递归下降预测分析程序中，采用的文法定义和翻译模式有些许不同，因此采用的翻译模式与实验三还是有些不同。为了简洁起见，这里的翻译模式我只采用叙述的方式，而不使用对应的源代码，主要在于说明在当前位置需要执行什么动作。具体定义为(同一个标识符的多个有相同开头的产生式需要用if语句进行区分，但是所做的动作类似，为了便于实现，产生式中不消除Left Factoring，从而不需要增加新的非终结符号)：

```
module ::= MODULE IDENTIFIER SEMI declarations END IDENTIFIER DOT
{返回declaration的复杂度}
| MODULE IDENTIFIER SEMI declarations BEGIN statement_sequence END
IDENTIFIER DOT
{返回declaration和statement_sequence的复杂度之和};

declarations ::= const_declare type_declare var_declare procedure_declare
{返回所有declare语句的复杂度之和};

const_declare ::= CONST identifier_const
{返回identifier_const的复杂度+5}
| /*empty*/
{返回复杂度0};

identifier_const ::= IDENTIFIER EQUAL expression SEMI
{检查IDENTIFIER是否在符号表中以及插入符号表}
identifier_const
{返回expression加identifier_const的复杂度}
| /*empty*/
{返回复杂度0};

type_declare ::= TYPE identifier_type
{返回identifier_type的复杂度+10}
| /*empty*/
{返回复杂度0};

identifier_type ::= IDENTIFIER EQUAL type SEMI
{检查IDENTIFIER是否在符号表中以及插入符号表，IDENTIFIER的类型设置为type}
identifier_type
{返回type加identifier_type的复杂度}
| /*empty*/
{返回复杂度0};

var_declare ::= VAR identifier_var
{返回identifier_var的复杂度}
| /*empty*/
{返回复杂度0};

identifier_var ::= identifier_list COLON type SEMI
{对identifier_list中的逐个标识符进行符号表检查，依次加入符号表}
identifier_var
{返回identifier_list与type复杂度之和}
| /*empty*/
{返回复杂度0};
```

```
procedure_declare ::= procedure_declaration SEMI procedure_declare
{返回复杂度之和}
| /*empty*/
{返回复杂度0};

procedure_declaration ::= procedure_heading SEMI procedure_body
{检查两者的标识符名是否相同，返回复杂度之和};

procedure_body ::= declarations END IDENTIFIER
{设置标志符名，返回复杂度和标识符名，弹出一个符号表}
| declarations BEGIN statement_sequence END IDENTIFIER
{设置标志符名，返回复杂度和标识符名，弹出一个符号表};

procedure_heading ::= PROCEDURE IDENTIFIER
{设置标志符名，登记对应的过程，增加一个符号表，返回复杂度和标识符名}
| PROCEDURE IDENTIFIER formal_parameters
{设置标志符名，登记对应的过程，设置好参数列表，增加一个符号表，将参数列表加入当前符号表，返回复杂度和标识符名};

formal_parameters ::= LPATH RPATH
{返回空白参数列表}
| LPATH fp_section optional_fp RPATH
{合并fp和optional_fp的参数列表，返回参数列表，其中是所有的参数类型和参数名称};

optional_fp ::= SEMI fp_section optional_fp
{合并fp和optional_fp的参数列表，返回参数列表，其中是所有的参数类型和参数名称}
| /*empty*/
{返回空白参数列表};

fp_section ::= identifier_list COLON type
{将identifier_list中的所有参数增加到参数列表中，类型为type，返回参数列表和复杂度}
| VAR identifier_list COLON type
{将identifier_list中的所有参数增加到参数列表中，类型为type，设置为要求左值，返回参数列表和复杂度};

type ::= IDENTIFIER
{检查标识符是否存在及其是否为一个类型，返回标识符的类型Type，复杂度为2}
| array_type
{新创建一个Type类型，设置为array类型，复杂度为array_type+8}
| record_type
{新创建一个Type类型，设置为record类型，复杂度为record_type+3}
| INTEGER
{新创建一个Type类型，设置为integer类型，复杂度为1}
| BOOLEAN
{新创建一个Type类型，设置为boolean类型，复杂度为1};

record_type ::= RECORD field_list optional_field END
{合并field_list和optional_field的符号表，查重，复杂度设置为两者之和};

optional_field ::= SEMI field_list optional_field
{合并field_list和optional_field的符号表，查重，复杂度设置为两者之和}
| /*empty*/
{返回空白表};

field_list ::= identifier_list COLON type
{在符号表中查重后，将所有的identifier_list中的标识符设置为type}
| /*empty*/
```

```
{返回空白表};

array_type ::= ARRAY expression OF type
{设置数组类型为type，复杂度为表达式和类型之和};

identifier_list ::= IDENTIFIER optional_identifier
{连接IDENTIFIER名称和optional_identifier的标识符集合};

optional_identifier ::= COMMA IDENTIFIER optional_identifier
{连接IDENTIFIER名称和optional_identifier的标识符集合}
| /*empty*/
{返回空白集合};

statement_sequence ::= statement optional_statement
{返回复杂度为statement和optional_statement之和};

optional_statement ::= SEMI statement optional_statement
{返回复杂度为statement和optional_statement之和}
| /*empty*/
{返回复杂度0};

statement ::= assignment | procedure_call | if_statement | while_statement |
/*empty*/;{对于每种语句，都是在执行完后设置statement复杂度为其对应的复杂度，assignment
和procedure_call区分在于检测当前的identifier是一个procedure类型还是calculate(用于计
算的)类型}

while_statement ::= WHILE {WHILE层数+1} expression DO statement_sequence END;

if_statement ::= IF {IF层数+1} expression THEN statement_sequence
elsif_statement else_statement END
{复杂度为(IF层数+1)*(2^WHILE层数)*expression复杂度+其他复杂度};


elsif_statement ::= ELSIF expression THEN statement_sequence elsif_statement
{复杂度为(IF层数+1)*(2^WHILE层数)*expression复杂度+其他复杂度}
| /*empty*/
{返回复杂度0};

else_statement ::= ELSE statement_sequence
{返回statement_sequence复杂度}
| /*empty*/
{返回复杂度0};

procedure_call ::= IDENTIFIER
{进行参数检查(标识符存在性在statement处已经需要一次检查，这里不需要再检查)，返回复杂度8}
| IDENTIFIER actual_parameters
{进行参数类型检查，返回复杂度8+actual_parameters的复杂度};

actual_parameters ::= LPATH RPATH
{返回空白参数类型表}
| LPATH expression optional_expression RPATH
{返回类型表为连接expression类型和optional_expression的类型表};

optional_expression ::= COMMA expression optional_expression
{返回类型表为连接expression类型和optional_expression的类型表}
| /*empty*/
{返回空白类型表};
```

```
assignment ::= IDENTIFIER selector ASSIGN expression
{检查IDENTIFIER类型与selector是否匹配，检查筛选后类型与expression是否匹配，返回
2+selector和expression复杂度之和};

expression ::= simple_expression {直接返回simple_expression的类型Type}
| simple_expression EQUAL simple_expression | simple_expression NEQUAL
simple_expression | simple_expression LESS simple_expression |
simple_expression ELESS simple_expression | simple_expression MORE
simple_expression | simple_expression EMORE simple_expression
{对于每种计算，都进行类型检查，并创建一个新的类型Type，设置为结果的类型};

simple_expression ::= sign term optional_term
{判断optional_term是否为空，为空时返回term的类型Type，复杂度为三者之和；不为空时检查
term和optional_term类型是否匹配};

sign ::= ADD {返回复杂度2}| MINUS {返回复杂度2}| /*empty*/{返回复杂度0};

optional_term ::= ADD term optional_term | MINUS term optional_term | OR
term optional_term | /*empty*/
{对于每种计算，都需要检查对应的类型，并创建一个新的类型Type，设置为结果的类型};

term ::= factor optional_factor
{判断optional_factor是否为空，为空时返回factor的类型Type，复杂度为三者之和；不为空时检
查factor和optional_factor类型是否匹配};

optional_factor ::= MUL factor optional_factor | DIV factor optional_factor
| MOD factor optional_factor | AND factor optional_factor | /*empty*/
{对于每种计算，都需要检查对应的类型，并创建一个新的类型Type，设置为结果的类型};

factor ::= IDENTIFIER selector
{检查对应标识符的存在性，并进行筛选，返回对应类型}
| DIGIT
{创建一个类型Type，类型为integer}
| LPATH expression RPATH
{返回expression的类型Type，复杂度需要加上6}
| NOT factor
{检查factor是否为boolean类型，返回一个新建类型};

selector ::= DOT IDENTIFIER selector
{将"."+标识符名加入到selector的串中}
| LBRACK expression RBRACK selector
{将"[]"加入到selector的串中}
| /*empty*/
{返回空串};
```

2. 编写递归下降预测分析程序

编写递归下降预测分析程序时，首先我们需要最基本的match函数，用于进行匹配操作。我们将每个非终结符号都设置为一个函数，并将返回值设置为其对应的类型，从而使我们能在调用者处访问到对应非终结符号的属性，进行下一步的动作。除了部分动作的执行顺序与LR parser有些不同外，大部分的行为都可以直接套用cup中说明的动作。除了动作外，在某个非终结符号有多个产生式时，我们需要小心地选择其中一个进行递归下降分析，这里主要采用的是FIRST-FOLLOW的概念作为指导，并利用这个概念进行错误检查。对于每一个产生式的每一个元素，我们都需要去调用相应的match动作或是对应的过程。每个非终结符对应的过程形如

```
void module() throws Exception {
    Nonterminal RESULT = new Nonterminal();
```

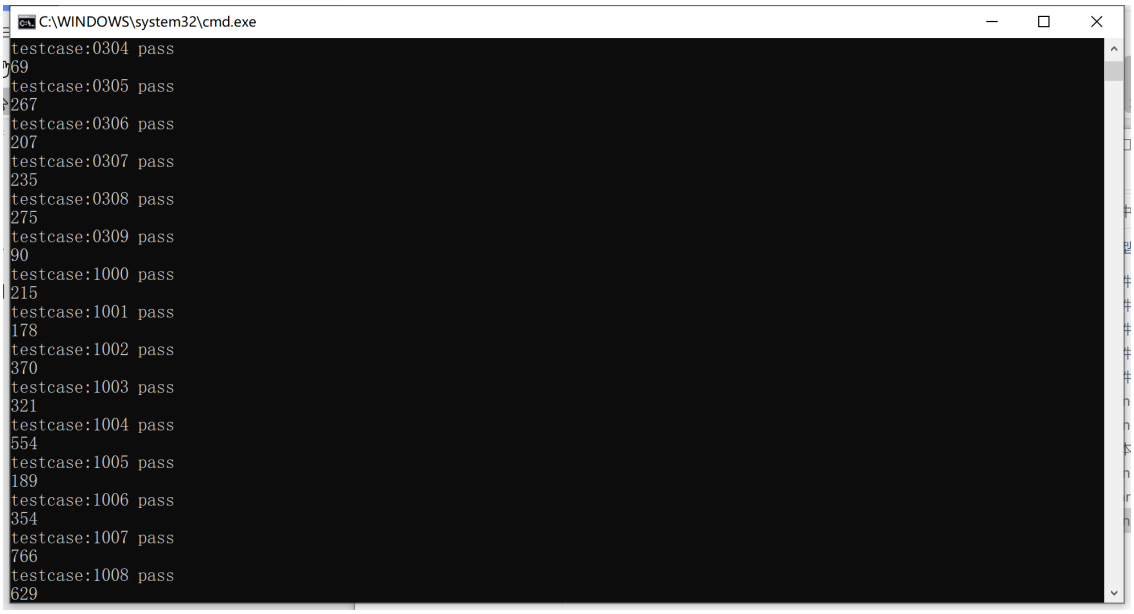
```
    match(16);
    String id1 = (String) lookahead.value;
    match(44);
    match(37);
    Nonterminal declar = declarations();
    if (lookahead.sym == 17) {
        match(17);
        Nonterminal ss = statement_sequence();
        RESULT.complexity += ss.complexity;
    }
    match(18);
    String id2 = (String) lookahead.value;
    match(44);
    match(36);
    if (!id1.equalsIgnoreCase(id2)) {
        throw new SemanticException("Module name mismatched!");
    }
    RESULT.complexity += declar.complexity;
    res = RESULT.complexity;
}
```

match函数为

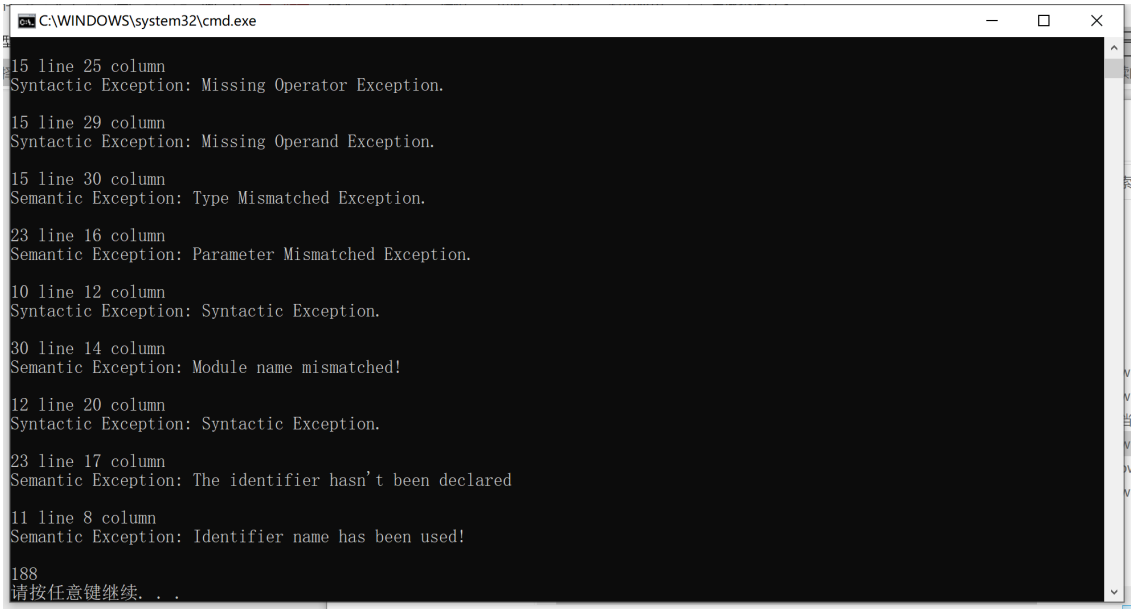
```
void match(int val) throws Exception {
    if (lookahead.sym != val) {
        throw new SyntacticException();
    }
    lookahead = obj.next_token();
}
```

3. 递归下降预测分析程序测试

首先使用实验软装置中的测试，检查在代码正确的情况下，能否得到正确结果。测试结果为



接下来检测能否正确识别出对应的语法、词法、语义错误，应用我实验三中的主函数和测试用例，查看是否能正确识别。结果如下



结果符合预期，代码实现正确

4. 语法分析讨论：自顶向下 vs. 自底向上
    - 从技术的简单性上来看，如果不使用对应的自动生成工具，自底向上的LR分析技术的实现复杂度是远远大于自顶向下的分析技术的，因为相对于LR分析而言，递归下降预测分析技术为每个非终结符号都创建了一个对应的过程，并且在分析过程中调用过程或是进行终结符匹配，相对而言更加地直观。并且，由于这种直观的特性，我们很快就能根据当前的lookahead找到对应的错误发生位置，从而方便我们进行调试。而对于LR分析而言，我们需要找到对应的产生式进行规约，需要根据对应的LR分析表做决策，且要做状态的维护等，实现难度大大提高。即使是采用了自动生成工具，我们还需要考虑要将语义动作放在最后的问题，不允许embedded action，从而需要修改产生式甚至是增加标记符marker，增加了不少的工作量。但是采用自动生成工具我们不需要手动去实现语法分析，还是比较方便的。整体上看，我实现了LR分析后，只花了一些时间就实现了递归下降预测分析技术，且不需要过多的调试即可，可以感受到递归下降预测分析技术更容易去实现
    - 技术的通用性上看，LR分析的通用性远大于自顶向下的分析技术。无论是文法上还是语言上，LR分析能处理的范围都要大于LL分析，因为自顶向下分析不支持左递归和left factoring等等，需要进行文法的修改。而自顶向下分析需要依据FIRST和FOLLOW进行判断，对于有些语言，很容易就出现有两个产生式的FIRST集相同的情况，导致无法使用自顶向下的技术。在本实验中，我修改了实验三中的文法后，才能进行自顶向下分析，可见在通用性上，LR分析适用范围更加广泛
    - 表达语义动作上来看，自顶向下的分析技术支持embedded action，而不需要将所有语义动作都放在产生式最后，从而我们可以更灵活地添加语义动作，完成语法制导翻译
    - 在本次实验实现的两种不同方法的语法分析技术中，我都采用了增加错误表达式的方法实现出错恢复。但是在LR分析技术中，增加错误表达式往往会导致一些状态出现conflict的情况，这导致可能在某些情况下，我们必须小心地选择一个语义动作，来避免conflict的情况(在本实验中采用javacup默认的选择，但是由于不知道内部情况，无法保障默认选择是正确的)，但是我们可能会有一些无法考虑到的情况，导致出现错误。但是在自顶向下的方法中，我们可以直接根据当前的lookahead来判断是否发生了预期的错误，往往不需要继续进行parsing就能找到错误，减少了发生预期之外的错误的可能。因此出错恢复上，自顶向下更加容易实现
    - 分析表大小方面，LR分析技术的表格记录的是状态，而LL分析技术只需要记录非终结符，状态的数量往往是非终结符数量的指数级，因此LL分析的分析表是远小于LR分析的分析表
    - 由于没法生成较大的程序，导致实际分析时间都过短，没办法直接进行比较，因此分析技术速度上不确定谁的速度更快。