实验三、自动生成语法分析程序

1. 配置和试用 JavaCUP

   为了测试java-cup的使用，我将官网中的算术表达式的自动生成copy到我自己的计算机上，并尝试运行。首先使用java -jar java-cup-11b.jar -interface -parser Parser calc.cup指令自动化生成算术表达式的语法分析器，再使用给定的基本词法分析器和main函数，测试词法分析器和语法分析器配合情况。运行情况为：

名称	修改日期	类型
calc.cup	2022/5/29 16:03	CUP 文件
Main	2022/5/29 16:15	JAVA 文件
Parser	2022/5/29 16:10	JAVA 文件
scanner	2022/5/29 16:19	JAVA 文件
sym	2022/5/29 16:10	JAVA 文件

   可以看到自动生成了Parser以及对应的符号表

   ```
   C:\WINDOWS\system32\cmd.exe
   Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
   Please type your arithmethic expression:
   1+2-3+4-5*6+7*8;
   1
   30
   ```

   能够生成正确结果，说明安装配置成功

2. 生成 Oberon-0 语法分析和语法制导翻译程序

   首先，我们需要将对应的EBNF修改为一个Context-free Grammer，修改后的结果为(为了按照cup的格式，这里用::=表示->)

```
module ::= MODULE IDENTIFIER SEMI declarations END IDENTIFIER DOT| MODULE
IDENTIFIER SEMI declarations BEGIN statement_sequence END IDENTIFIER DOT;

declarations ::= const_declare type_declare var_declare procedure_declare;

const_declare ::= CONST identifier_const | /*empty*/;

identifier_const ::= IDENTIFIER EQUAL expression SEMI identifier_const |
/*empty*/;

type_declare ::= TYPE identifier_type | /*empty*/;

identifier_type ::= IDENTIFIER EQUAL type SEMI identifier_type | /*empty*/;

var_declare ::= VAR identifier_var | /*empty*/;

identifier_var ::= identifier_list COLON type SEMI identifier_var |
/*empty*/;

procedure_declare ::= procedure_declaration SEMI procedure_declare |
/*empty*/;

procedure_declaration ::= procedure_heading SEMI procedure_body;
```

```
procedure_body ::= declarations END IDENTIFIER | declarations BEGIN
statement_sequence END IDENTIFIER;

procedure_heading ::= PROCEDURE IDENTIFIER | PROCEDURE IDENTIFIER
formal_parameters;

formal_parameters ::= LPATH RPATH | LPATH fp_section optional_fp RPATH ;

optional_fp ::= SEMI fp_section optional_fp | /*empty*/;

fp_section ::= identifier_list COLON type | VAR identifier_list COLON type;

type ::= IDENTIFIER | array_type | record_type | INTEGER | BOOLEAN;

record_type ::= RECORD field_list optional_field END;

optional_field ::= SEMI field_list optional_field | /*empty*/;

field_list ::= identifier_list COLON type | /*empty*/;

array_type ::= ARRAY expression OF type;

identifier_list ::= IDENTIFIER optional_identifier;

optional_identifier ::= COMMA IDENTIFIER optional_identifier | /*empty*/;

statement_sequence ::= statement optional_statement;

optional_statement ::= SEMI statement optional_statement | /*empty*/;

statement ::= assignment | procedure_call | if_statement | while_statement |
/*empty*/;

while_statement ::= WHILE M expression DO statement_sequence END;

M ::= /*empty*/;

if_statement ::= IF N expression THEN statement_sequence elsif_statement
else_statement END;

N ::= /*empty*/;

elsif_statement ::= ELSIF expression THEN statement_sequence elsif_statement
| /*empty*/;

else_statement ::= ELSE statement_sequence | /*empty*/;

procedure_call ::= IDENTIFIER | IDENTIFIER actual_parameters;

actual_parameters ::= LPATH RPATH | LPATH expression optional_expression
RPATH;

optional_expression ::= COMMA expression optional_expression | /*empty*/;

assignment ::= IDENTIFIER selector ASSIGN expression;
```

```
expression ::= simple_expression | 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;

simple_expression ::= sign term optional_term;

sign ::= ADD | MINUS | /*empty*/;

optional_term ::= ADD term optional_term | MINUS term optional_term | OR
term optional_term | /*empty*/;

term ::= factor optional_factor;

optional_factor ::= MUL factor optional_factor | DIV factor optional_factor
| MOD factor optional_factor | AND factor optional_factor | /*empty*/;

factor ::= IDENTIFIER selector | DIGIT | LPATH expression RPATH | NOT
factor;

selector ::= DOT IDENTIFIER selector | LBRACK expression RBRACK selector |
/*empty*/;
```

接下来需要考虑如何进行语义分析(语法分析直接通过CFG可以进行语法检测)。首先对于所有的终结符，事实上我们需要使用到值的只有DIGIT和CONSTANT。在词法分析器中，我们已经令DIGIT的值为Integer类型，IDENTIFIER的值为String类型，从而在后续定义中可以直接使用其对应的值。在cup文件中，我们定义对应的终结符，并为DIGIT和CONSTANT声明其类型

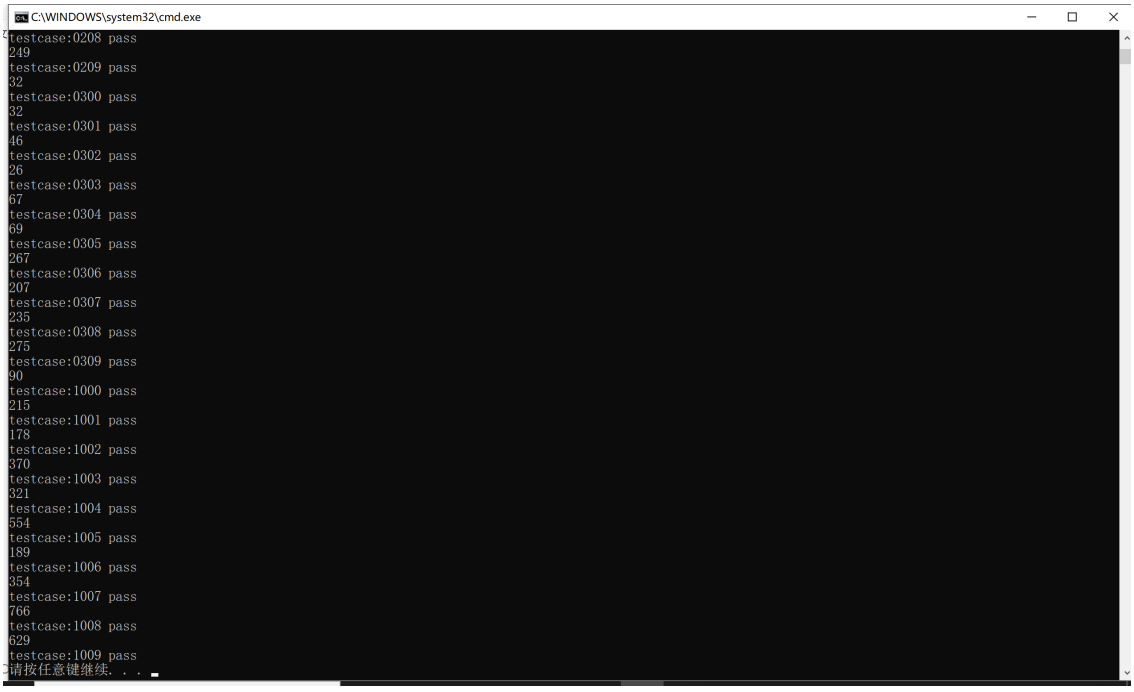
```
/*Terminals*/
/*operator*/
terminal EQUAL, NEQUAL, LESS, ELESS, MORE, EMORE, ADD, MINUS, OR, MUL, DIV,
MOD, AND, NOT;
/*reserved words*/
terminal MODULE, BEGIN, END, CONST, TYPE, VAR, RECORD, ARRAY, OF, WHILE, DO,
IF, THEN, ELSIF, ELSE, PROCEDURE;
/*type*/
terminal INTEGER, BOOLEAN;
/*punctuation*/
terminal LPATH,RPATH,DOT,SEMI,COLON,COMMA,ASSIGN,LBRACK,RBRACK;
/*constant*/
terminal Integer DIGIT;
/*identifier*/
terminal String IDENTIFIER;
```

接下来考虑非终结符号及其所需要的属性。对于所有的非终结符号，我们一定需要其复杂度，由于进行最后的复杂度计算，因此所有的非终结符号都需要有complexity属性；expression，simple_expression，term、factor、optional_term和optional_factor都需要记录类型，确保运算类型正确。同时，由于需要记录expression是否为左值，我们还需要left变量记录其左值的情况。只有当expression中不进行任何计算时它才可以为左值；由于procedure_body和procedure_heading是分开的，为了保证它们拥有相同的标识符名称，为两个终结符增加name属性。因此，expression的类可以与常规的标识符相同；对于参数列表formal_parameters、optional_fp、fp_section、actual_parameters和optional_expression，我们需要记录其中的参数类型，便于进行类型匹配；Type为我们字符表中存储的数据，它应当包括base_type(类型)，actual_type(具体类型)，length(长度，对于array有用)，symbol_list(对于record类型有用)，还包

括参数列表(对于过程有用)等。为了避免出现多余的属性，我们只令type包含一个symbol_list和一个包含有所有需要的属性的List，不同类型的标志符中的list存放的数据不同。另外，由于array中也需要用到type类型，我们还增加一个Type类的对象，用于存储array中的类型；record_type中应该保存对应的符号表，而field_list和optional_field中同样需要保存符号表，并在规约时进行查重合并；所有的statement都只需要complexity即可，但是由于while和if需要考虑层数，在WHILE和IF后增加一个空的非终结符M和N，用于增加层数。在语句结束时将层数去除；最后是selector符号，需要记录对应的参数等，在规约时进行参数检查和类型检测。定义非终结符号的代码为

```
/*Nonterminals*/
non terminal Nonterminal
module,declarations,const_declare,identifier_const,type_declare,identifier_t
ype,var_declare,identifier_var,procedure_declare,procedure_declaration,state
ment_sequence,optional_statement,statement,while_statement,if_statement,elsi
f_statement,else_statement,procedure_call,assignment,sign;
non terminal Procedure procedure_heading,procedure_body;
non terminal Parameter
formal_parameters,optional_fp,fp_section,actual_parameters,optional_expressi
on;
non terminal Type type;
non terminal Record record_type,optional_field,field_list;
non terminal Arr array_type;
non terminal Selector identifier_list,optional_identifier,selector;
non terminal Type
expression,simple_expression,optional_term,term,optional_factor,factor;
non terminal M,N,relaop;
```

接下来在cup文件中完成对文法的定义及其对应的翻译模式，具体的实现在cup中进行定义。这里的check_type函数用于检查两个类型是否匹配，select_field函数用于在selector中选择出对应的类型。翻译模式的核心思想在于：对于每一个定义的标识符，检查其是否在当前的符号表中，如果在则出错，否则加入符号表中；对于每一个要使用的标识符，检查其是否在当前及之前的符号表中，如果在则获得对应的Type类，否则报错；对于每一个运算式，我们都进行相应的类型检查，并在类型相同时进行相应的计算；对于每一个procedure调用，我们都进行完整地类型检查，确保调用是正确的；每一个产生式都要进行对应复杂度的计算，以便最终得到正确结果。首先我们设计出的程序能够处理所有的正确程序，并计算出对应的复杂度，通过测试得到



3. 设计拥有更加准确的纠错功能的程序

由于时间原因，这里我不在设计错误恢复，选择将错误检查变得更加细致，能够满足对应的要求。由于产生式只在规约时才能执行动作，我们没办法再翻译模式中找到对应的语法错误，只能单纯地报出Syntax Error。为了使错误更加准确，我们在CFG中加入特定的错误产生式，从而报出预期之内的错误。具体加入的错误产生式形如：

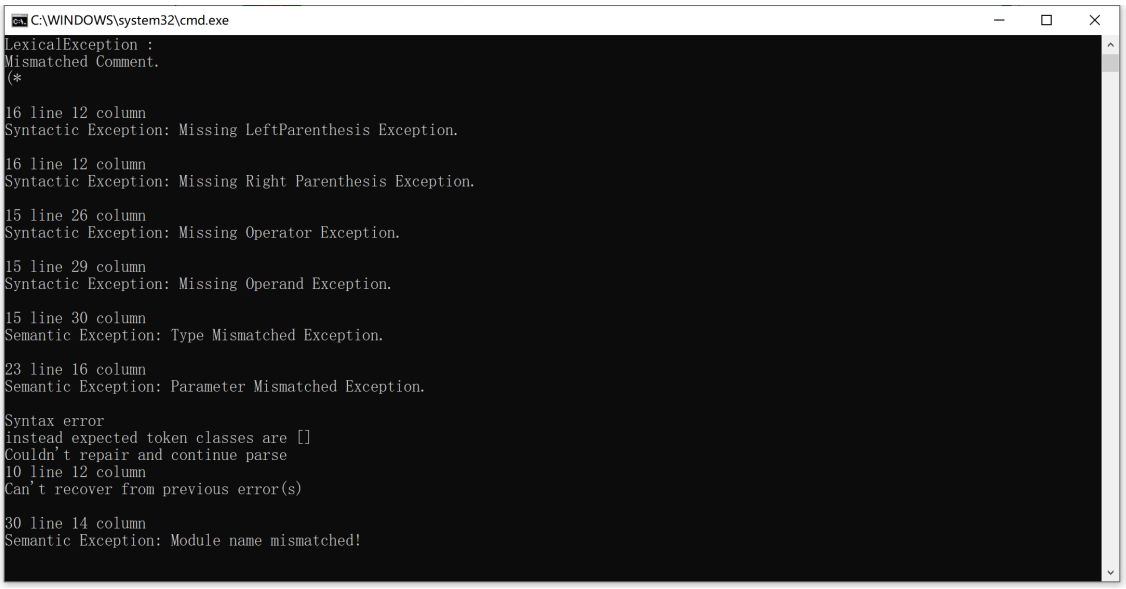
```
| factor:f optional_factor:of
{:
    if(true)
        throw new MissingOperatorException();
:}

| LPATH expression:e optional_expression:oe
{:
    if(true)
        throw new MissingRightParenthesisException();
:}
| expression:e optional_expression:oe RPATH
{:
    if(true)
        throw new MissingLeftParenthesisException();
:}
```

但是由于增加错误产生式很容易会出现一些parsing冲突，因此在gen中需要运行一些冲突用默认的方式执行，在gen中添加参数-expect 100增加冲突预期，从而在增加错误产生式产生冲突时，不会出现报错。但是由于错误产生式产生的冲突始终用默认的方法进行解决，可能会出现问题，导致正确的程序没能运行成功，因此这事实上不是一个很好的解决方法。我尝试去更改lr_parser中的对应报错函数，但是由于没有注释和文档，始终没有找到对应的调用链，因此没能用更好的方法解决问题。

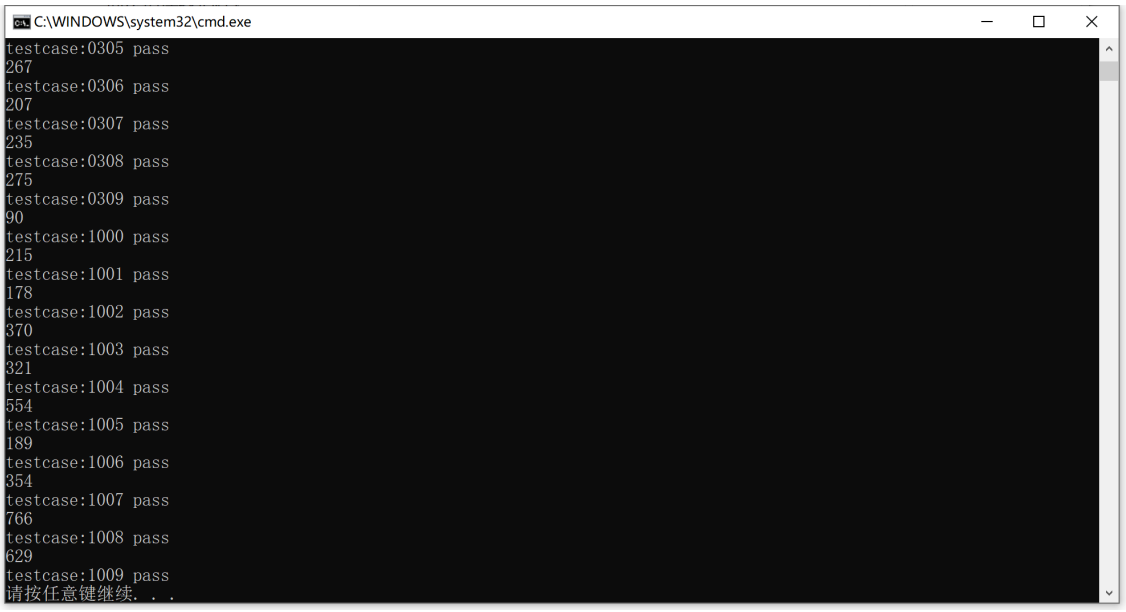
另外，在规约时，我们可以执行对应的类型检查和参数检查，从而精确地报出语义错误。具体行为形如(这里举出过程调用的判定方法)

```
int pos = -1;
for(int i = top;i>=0;i--){
    if(symbol_list.get(i).containsKey(id)){
        pos = i;
        break;
    }
}
if(pos==-1){
    throw new SemanticException("Identifier isn't been declared!");
}
Type t = symbol_list.get(pos).get(id);
if(!t.type.get(0).equalsIgnoreCase("procedure")){
    throw new TypeMismatchedException();
}
List<Type> para1 = t.para;
List<Type> para2 = ap.para;
if(para1.size()!=para2.size()){
    throw new ParameterMismatchedException();
}
for(int i = 0;i<para1.size();i++){
    if(!check_type(para1.get(i),para2.get(i))){
        throw new TypeMismatchedException();
    }
}
```

词法错误我们在词法分析器中已经完成。至此，利用javacup实现的语法分析器实现完毕。完成错误处理后，我尝试运行我在实验一中定义的所有程序，能够获得正确的报错，且能够运行正确的程序



同时运行软装置中的所有用例，也能够获得正确的结果。这说明成功实现了一个正确的语法分析器



4. 比较两种流行的语法分析程序自动生成工具之间的差异：JavaCUP 和 GNU Bison，主要讨论这两种软件

   工具接收输入源文件时，在语法规则定义方面存在的差异

   - 在输入源文件的组织上，两种自动生成工具之间有差异。

     GNU Bison的源文件共分为五个部分

     The Prologue：定义生成文件中的宏定义、预处理指令、全局变量和全局函数

     Prologue Alternatives：对The Prologue部分的补充

     The Bison Declaration Section：对终结符、非终结符和优先级等的声明

     The Grammar Rules Section：对应的文法规则，至少需要有一个文法规则

     The Epilogue：直接被复制到生成的parser尾部的代码

     javaCUP的源文件分为五个部分

     Package and import specification：定义自动生成的parser所处的package以及其相关引用

User code componentes：定义自动生成的parser中包含的用户代码和连接Scanner，包括action code、parser code、init with和scan with

Symbol lists：定义对应的符号表，包括终结符和非终结符

Precedence and associativity declarations：定义运算符的结合性质和优先级

The grammar：定义对应的文法和翻译模式

- 在使用和支持的语言上，JavaCUP采用的是java语言，GNU Bison采用的是C语言
- 定义符号的方式存在差异。在JavaCUP中，定义终结符和非终结符采用的方式为

```
terminal classname name1, name2, ...;
non terminal classname name1, name2, ...
```

而在GNU Bison中，定义符号采用的方式为

```
%token <classtype> name
%type <type> nonterminal
%nterm <type> nonterminal
```

- 定义优先级的方式存在差异。在JavaCUP中，优先级定义在统一的位置，定义方式为

```
precedence left     terminal[, terminal...];
precedence right    terminal[, terminal...];
precedence nonassoc terminal[, terminal...];
```

在GNU Bison中，定义优先级的方式为(与定义符号的方式类似)

```
%left <type> symbols
```

- 文法和动作的定义上存在差异。

在GNU Bison中，文法和动作定义形如

```
exp[result]:exp[left] '+' exp[right]  { $result = $left + $right; }

exp:exp '+' exp     { $$ = $1 + $3; }
```

Bison中的属性值由$返回，获取对应的属性值。在javaCUP中，文法和动作的定义形如

```
expr ::=  MINUS expr:e
              {: RESULT = new Integer(0 - e.intValue()); :}
```

JavaCUP中的结果通过RESULT返回