LAB3 ROSE

ex3

21307099 李英骏

目录

1 yacc 族工具语法规则定义的差异

1.1 JavaCUP vs GNU Bison

JavaCUP 和 GNU Bison 分别用于 Java 和 C/C++ 语言的编译器构造. 语法规则定义差异具体如下:

1.1.1 语法规则文件的结构

JavaCUP 语法规则文件 (.cup)

- **Package 和 import 声明**: 指定包和导入的类.
- **用户初始化代码**: 用户定义的初始化代码.
- **终结符与非终结符的声明**: 声明语法分析中的终结符和非终结符.
- **定义文法规则**: 定义实际的语法规则, 使用 {} 包围的 Java 动作代码.

```
1  package mypackage;
2  import java_cup.runtime.*;
3
4  terminal PLUS, MINUS, TIMES, DIVIDE;
5  terminal LPAREN, RPAREN;
6  terminal NUMBER;
7
8  non terminal expr, term, factor;
9
10 expr ::= expr PLUS term
11        | expr MINUS term
12        | term;
13
14 term ::= term TIMES factor
15        | term DIVIDE factor
16        | factor;
17
18 factor ::= NUMBER
19          | LPAREN expr RPAREN;
```

Bison 语法规则文件 (.y)

- **Prologue**：C 语言的头文件和宏定义.
- **Bison declarations**：终结符与非终结符的声明, 使用%token 等关键字.
- **Grammar rules**：定义实际的语法规则, 使用 {} 包围的 C 动作代码.
- **Epilogue**：C 语言的尾代码, 通常包含辅助函数.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token NUMBER
%token PLUS MINUS TIMES DIVIDE
%token LPAREN RPAREN

%left PLUS MINUS
%left TIMES DIVIDE

%%
expr: expr PLUS term { $$ = $1 + $3; }
    | expr MINUS term { $$ = $1 - $3; }
    | term;

term: term TIMES factor { $$ = $1 * $3; }
    | term DIVIDE factor { $$ = $1 / $3; }
    | factor;

factor: NUMBER
      | LPAREN expr RPAREN { $$ = $2; }
;

%%

int main() {
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

1.1.2 动作代码

- **JavaCUP**：动作代码使用 Java 语法, 通过 RESULT 变量返回值. 例如：

```
expr ::= expr:e1 PLUS term:e2
        {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
        | expr:e1 MINUS term:e2
        {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
        | term:e;
```

- **Bison**：动作代码使用 C 语法, 通过 $$ 返回值. 例如：

```
expr: expr PLUS term { $$ = $1 + $3; }
    | expr MINUS term { $$ = $1 - $3; }
    | term;
```

1.1.3 优先级和结合性

- **JavaCUP**：使用 precedence 声明来定义. 例如：

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

- **Bison**：使用%left 和%right 声明来定义. 例如：

```
%left PLUS MINUS
%left TIMES DIVIDE
```

1.1.4 错误处理

- **JavaCUP**：使用 error 关键字和 Java 异常机制处理语法错误.
- **Bison**：使用 error 关键字和 yyerror 函数处理语法错误.

1.2 JavaCUP vs JavaCC

JavaCUP 和 JavaCC 是两种用于 Java 语言的语法分析生成器。它们在功能和使用方式上存在一些显著差异。

1.2.1 语法规则文件的结构

JavaCUP 语法规则文件 (.cup)

- **Package 和 import 声明**：指定包和导入的类。
- **用户初始化代码**：用户定义的初始化代码。
- **终结符与非终结符的声明**：声明语法分析中的终结符和非终结符。
- **定义文法规则**：定义实际的语法规则，使用 {} 包围的 Java 动作代码。

JavaCC 语法规则文件 (.jj)

- **Options 和用户定义部分**：定义选项和用户自定义的 Java 代码。
- **语法规则**：定义实际的语法规则，包含解析方法和动作代码。
- **词法规则**：使用正则表达式定义词法规则。

```
options {
  STATIC = false;
}

PARSER_BEGIN(MyParser)
public class MyParser {
  public static void main(String[] args) throws ParseException {
    MyParser parser = new MyParser(System.in);
    parser.Start();
  }
}
PARSER_END(MyParser)

TOKEN: { < PLUS: "+" > | < MINUS: "-" > | < TIMES: "*" > | < DIVIDE: "/" >
   }

void Start() :
{}
{
  Expression() <EOF>
}
```

1 YACC 族工具语法规则定义的差异

```
21
22 void Expression() :
23 {}
24 {
25   Term()
26   (
27     ( <PLUS> | <MINUS> ) Term()
28   )*
29 }
30
31 void Term() :
32 {}
33 {
34   Factor()
35   (
36     ( <TIMES> | <DIVIDE> ) Factor()
37   )*
38 }
39
40 void Factor() :
41 {}
42 {
43   <NUMBER> | <LPAREN> Expression() <RPAREN>
44 }
```

1.2.2 动作代码

- **JavaCUP**：动作代码使用 Java 语法，通过 RESULT 变量返回值。例如：

```
1 expr ::= expr:e1 PLUS term:e2
2         {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
3         | expr:e1 MINUS term:e2
4         {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
5         | term:e;
```

- **JavaCC**：动作代码直接嵌入在语法规则中，使用 Java 语法。例如：

```
1 void Expression() :
2 {}
3 {
4   Term()
5   (
6     ( <PLUS> { System.out.print("+"); } | <MINUS> { System.out.print("-");
        } ) Term()
```

```
  )*
}
```

1.2.3 优先级和结合性

- **JavaCUP**：使用 precedence 声明来定义。例如：

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

- **JavaCC**：通过在语法规则中显式地处理优先级和结合性。例如：

```
void Expression() :
{}
{
  Term()
  (
    ( <PLUS> | <MINUS> ) Term()
  )*
}
```

1.2.4 错误处理

- **JavaCUP**：使用 error 关键字和 Java 异常机制处理语法错误。
- **JavaCC**：通过抛出和捕获 'ParseException' 来处理语法错误，支持在语法规则中定义错误恢复策略。

2 实验记录: 使用 JavaCUP 生成语法分析程序

从JavaCUP上下载 JavaCUP0.11b. 参考了Princeton JavaCUP文档.

2.1 对 Scanner 部分的修改

尽管文档中说可以直接用 ex2 的 Scanner, 为了实现对错误的定位, 我们需要加入

```
int getLine()  {  return yyline;}
int getColumn(){  return yycolumn;}
```

然后封装进 JavaCUP 可用的函数即可:

```
    private java_cup.runtime.Symbol symbol(int type, Object value) {
        return new java_cup.runtime.Symbol(type, yyline, yycolumn, value);
    }
```

因此我们的 javacup 文件夹中放了一个 jflex 的 jar.

2.2 实验结果

对正确样例:

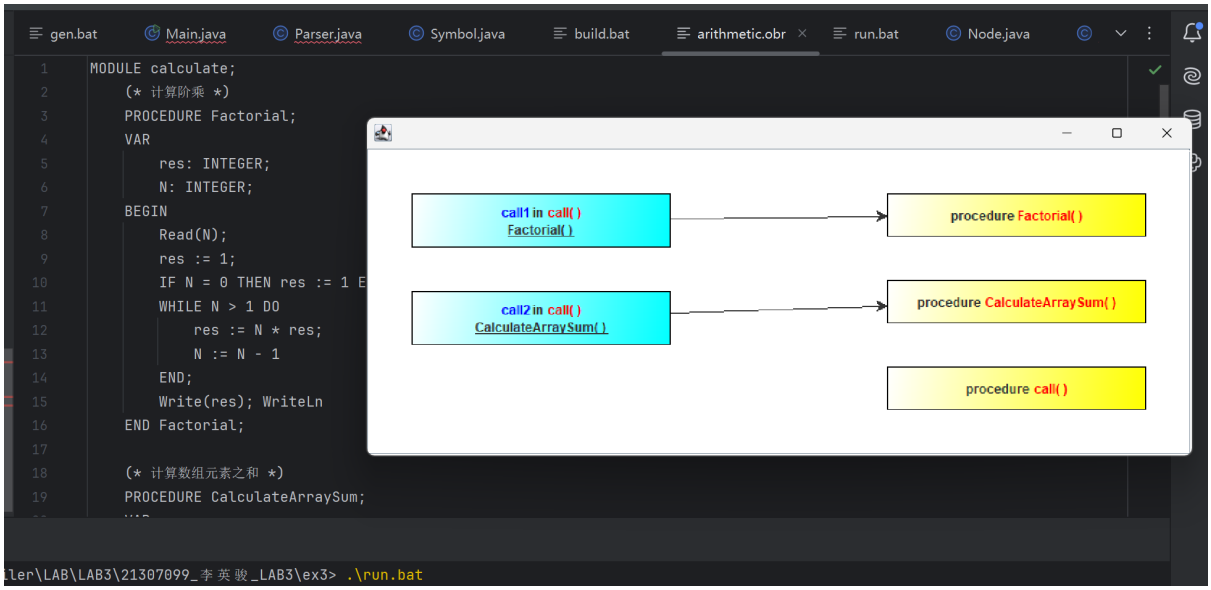

图 1: Arithmetic.obr

对错误样例:

```
# Error occurs at LINE 28 COL 7 java.lang.Exception: Can't recover from previous error(s)

Illegal Octal number.
Illegal Octal number.

Running arithmetic.002
..\src\testcases\arithmetic.002:# Error occurs at LINE 28 COL 13 exceptions.IllegalIntegerRangeException: LexicalException :
Illegal IntegerRange: more than 12.
Illegal IntegerRange: more than 12.

Running arithmetic.003
Illegal Symbol.
Illegal Symbol.

Running arithmetic.005
..\src\testcases\arithmetic.005:# Error occurs at LINE 14 COL 17 exceptions.MissingOperatorException: Syntactic Exception: Missing Operator Exception.

Running arithmetic.006
..\src\testcases\arithmetic.006:Syntax error at character 13 of input
instead expected token classes are [NOT, LeftParenthesis, TRUE, FALSE, IDENTIFIER, NUMBER]
Couldn't repair and continue parse at character 13 of input
# Error occurs at LINE 14 COL 12 java.lang.Exception: Can't recover from previous error(s)

Running arithmetic.007
..\src\testcases\arithmetic.007:# Error occurs at LINE 9 COL 9 exceptions.MissingRightParenthesisException: Syntactic Exception: Missing Right Parenthesis Exception.

Running arithmetic.008
..\src\testcases\arithmetic.008:# Error occurs at LINE 9 COL 10 exceptions.MissingLeftParenthesisException: Syntactic Exception: Missing LeftParenthesis Exception.

Running arithmetic.009
..\src\testcases\arithmetic.009:# Error occurs at LINE 29 COL 14 exceptions.TypeMismatchedException: Semantic Exception: Type Mismatched Exception.

Running arithmetic.010
..\src\testcases\arithmetic.010:# Error occurs at LINE 11 COL 11 exceptions.TypeMismatchedException: Semantic Exception: Type Mismatched Exception.
```

图 2: Exceptions

检测左括号缺失的部分有 bug, 会额外造成 javalang 的异常. 时间有限不做处理.

3 附录: 语法制导翻译模式

- **declarations:** 该规则定义了常量声明、类型声明、变量声明和过程声明的顺序。每个部分都可以为空。

```
declarations ::=  const_declarations type_declarations var_declarations procedure_declaration  {:  :}
                ;
```

- **procedure_declaration:** 定义了过程声明的规则。每个过程声明包括过程头和过程体，并且在语义动作中检查头部和体部的一致性，重置当前过程的标识和调用编号。

```
procedure_declaration ::= {: parser.callee = 1; :}
                        |  procedure_declaration procedure_heading:head SEMICOLON procedure_body:body SEMICOLON
                           {:
                              if (!head.equals(body))
                                  throw new SemanticException();
                              parser.currentProcedure = "";
                              parser.callee = 1;
                            :}
                        ;
```

- **procedure_heading:** 定义了过程头的规则，包含过程名和形式参数列表。在语义动作中，将过程名和参数添加到过程列表和调用图中，并初始化当前过程和调用编号。

```
procedure_heading ::=  PROCEDURE IDENTIFIER:procedureName format_parameters:formalParameters
                       {:
                           String newProcedure = new String(procedureName + "=" + "( " + formalParameters + " )");
                           parser.procedure.addElement(newProcedure);
```

```
                        parser.graph.addProcedure(procedureName, procedureName + "( " + formalParameters + " )");
                        parser.currentProcedure = procedureName;
                        parser.callee = 1;
                        RESULT = procedureName;
                    :}
                ;
```

- **format_parameters:** 定义了形式参数的规则，可以为空或包含一个或多个参数列表。处理缺少括号的异常。

```
format_parameters ::=  {: RESULT = ""; :}
                    |  LeftParenthesis RightParenthesis {: RESULT = ""; :}
                    |  LeftParenthesis fp_section:fp_sectionReturnString RightParenthesis {: RESULT = fp_sectionReturnString; :}
                    |  fp_section:fp_sectionReturnString RightParenthesis {: if (true) throw new MissingLeftParenthesisException(); :}
                    ;
```

- **fp_section:** 定义了参数段的规则，包含参数名列表和类型。在语义动作中，将参数类型和名称拼接形成参数列表字符串。

```
fp_section ::= fp_section:fp SEMICOLON identifier_list: identifierListReturnParamentNum COLON special_type: typeStringReturnFromspecial_type
                {:
                    for (int i = 0; i < identifierListReturnParamentNum.size(); i++) {
                        RESULT = fp + ", " + typeStringReturnFromspecial_type.type;
                    }
                :}
             | fp_section:fp SEMICOLON VAR identifier_list: identifierListReturnParamentNum COLON special_type: typeStringReturnFromspecial_type
                {:
                    for (int i = 0; i < identifierListReturnParamentNum.size(); i++) {
                        RESULT = fp + ", " + typeStringReturnFromspecial_type.type;
                    }
                :}
             | identifier_list : identifierListReturnParamentNum COLON special_type: typeStringReturnFromspecial_type
                {:
                    RESULT = "";
                    for (int i = 0; i < identifierListReturnParamentNum.size(); i++) {
                        RESULT += " " + typeStringReturnFromspecial_type.type;
                    }
                :}
             | VAR identifier_list: identifierListReturnParamentNum COLON special_type : typeStringReturnFromspecial_type
                {:
                    RESULT = "";
                    for (int i = 0; i < identifierListReturnParamentNum.size(); i++) {
                        RESULT += " " + typeStringReturnFromspecial_type.type;
                    }
                :}
             ;
```

- **special_type:** 定义了特殊类型的规则，包括标识符、整数、布尔、记录类型和数组类型。在语义动作中，检查标识符是否存在并返回相应的类型信息。

```
special_type ::=  IDENTIFIER:identifier
                  {:
                      RESULT = new Node();
                      boolean found = false;
                      for (int i = 0; i < parser.symbols.size(); i++) {
                          if (parser.symbols.elementAt(i).name.equals(identifier)) {
                              RESULT.type = parser.symbols.elementAt(i).type;
                              found= true;
                          }
                      }
                      if (!found)
                          throw new SemanticException();
                  :}
                |  INTEGER
                  {:
                     RESULT = new Node();
                     RESULT.type = "INTEGER";
                  :}
                |  BOOLEAN
                  {:
                      RESULT = new Node();
                      RESULT.type = "BOOLEAN";
                  :}
                |  record_type: recordToken
```

3 附录：语法制导翻译模式

```
                {:
                    RESULT = new Node(recordToken);
                :}
              |  array_type: arrayToken
                {:
                    RESULT = new Node(arrayToken);
                :}
              ;
```

- **array_type:** 定义了数组类型的规则，包含表达式和基本类型。在语义动作中，生成数组类型的节点。

```
array_type ::= ARRAY expression : expressionToken OF special_type :typeToken
               {:
                   RESULT = new Node();
                   RESULT.type= expressionToken.name  +"[" + typeToken.type +"]";
               :}
            ;
```

- **record_type:** 定义了记录类型的规则，包含字段列表。在语义动作中，生成记录类型的节点。

```
record_type ::= RECORD field_list:field_listTokne field_list_list: field_list_listToken END
                {:
                   RESULT = new Node();
                   RESULT.type = field_listTokne.type + field_list_listToken.type;
                :}
             ;
```

- **field_list:** 定义了字段列表的规则，可以为空或包含标识符列表和类型。在语义动作中，生成字段列表的节点。

```
field_list ::= {:RESULT = new Node();:}
            | identifier_list:identifierList COLON special_type: specialTypeToken
              {:
                  RESULT = new Node();
                  for (int i = 0; i < identifierList.size(); i++) {
                      RESULT.type += specialTypeToken.type;
                      RESULT.name += identifierList.elementAt(i).name;
                  }
              :}
            ;
```

- **field_list_list:** 定义了字段列表的规则，可以为空或包含多个字段列表。在语义动作中，将多个字段列表连接起来。

```
field_list_list ::= {: RESULT = new Node(); :}
                  | field_list_list : field_list_listToken SEMICOLON field_list : field_listToken
                    {:
                        RESULT = new Node();
                        RESULT.name = field_list_listToken.name + field_listToken.name;
                        RESULT.type = field_list_listToken.type + field_listToken.type;
                    :}
                  ;
```

- **identifier_list:** 定义了标识符列表的规则，可以为空或包含多个标识符。在语义动作中，将标识符添加到列表中。

```
identifier_list ::= IDENTIFIER : identifierName
                    {:
                        RESULT = new Vector<Node>();
                        RESULT.addElement(new Node(identifierName, ""));
                    :}
                  | identifier_list : previerIndentifierListParamentNum COMMA IDENTIFIER: identifierName
                    {:
                        RESULT = previerIndentifierListParamentNum;
                        RESULT.addElement(new Node(identifierName, ""));
                    :}
                  ;
```

- **procedure_body:** 定义了过程体的规则，包括声明部分和过程开始部分。在语义动作中，返回过程名。

3 附录：语法制导翻译模式

```
procedure_body ::= declarations procedure_begin END IDENTIFIER: procedreName
                   {:
                        RESULT = procedreName;
                   :}
                 ;
```

- **procedure_begin:** 定义了过程开始的规则，可以为空或包含语句序列。

```
procedure_begin ::=
                 | BEGIN statement_sequence
                 ;
```

- **statement_sequence:** 定义了语句序列的规则，可以包含多个语句。

```
statement_sequence ::= statement
                     | statement_sequence SEMICOLON statement
                     ;
```

- **statement:** 定义了语句的规则，可以是赋值语句、过程调用语句、条件语句、循环语句或读写语句。

```
statement ::= assignment
            | procedure_call
            | if_statement
            | while_statement
            | readwritestatement
            |
            ;
```

- **procedure_call:** 定义了过程调用的规则，包含过程名和实际参数列表。在语义动作中，更新调用图和过程调用信息。

```
procedure_call ::= IDENTIFIER: procedureName actual_parameters: actualParameters
                   {:
                       if (parser.callee == 1 && parser.currentProcedure.isEmpty()) {
                           parser.currentProcedure = "Main";
                           parser.graph.addProcedure("Main", "Main( )");
                           String newProcedure = new String(parser.currentProcedure + "=" + "( "  + " )");
                           parser.procedure.addElement(newProcedure);
                       }
                       parser.callSites.addElement(new String(parser.currentProcedure + parser.callee + "=" + parser.currentProcedure + "( )" + "=" + proce

                       System.out.println("WYT : " + procedureName);

                       parser.callEdges.addElement(new String(parser.currentProcedure + parser.callee + "=" + procedureName));
                       parser.callee++;
                   :}
                 ;
```

- **actual_parameters:** 定义了实际参数的规则，可以为空或包含一个或多个表达式。在语义动作中，生成参数节点。

```
actual_parameters ::= {: RESULT = new Node(); :}
                    | LeftParenthesis expression_list:parameters RightParenthesis {: RESULT = new Node(parameters); :}
                    ;
```

- **expression_list:** 定义了表达式列表的规则，可以为空或包含一个或多个表达式。在语义动作中，将表达式添加到列表中。

```
expression_list ::= {: RESULT = new Node(); :}
                  | expression:expressionToken {: RESULT =new Node(expressionToken); :}
                  | expression_list: expressionNameList COMMA expression:expressionToken
                    {:
                        RESULT = new Node();
                        RESULT.name = expressionNameList.name + "," + expressionToken.name;
                        RESULT.type = expressionNameList.type + " " + expressionToken.type;
                    :}
                  ;
```

- **expression:** 定义了表达式的规则，可以是简单表达式或包含比较运算符的复合表达式。在语义动作中，生成表达式节点，并进行类型检查。

3 附录：语法制导翻译模式

```
expression ::= simple_expression :simpleExpressionToken
               {:
                   RESULT = new Node();
                   RESULT.name = simpleExpressionToken.name;
                   RESULT.type = simpleExpressionToken.type;
               :}
             | simple_expression :simpleExpressionToken simple_expression_list :simpleExpressionListToken
               {:
                   if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpressionToken.t
                       throw new TypeMismatchedException();
                   }
                   RESULT = new Node();
                   RESULT.name = simpleExpressionToken.name + simpleExpressionListToken.name;
                   RESULT.type = "BOOLEAN";
               :}
             ;
```

- **simple_expression_list:** 定义了简单表达式列表的规则，可以包含比较运算符和简单表达式。在语义动作中，进行类型检查和生成相应节点。

```
simple_expression_list ::= EQUAL simple_expression : simpleExpressionToken
                           {:
                               if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                                   throw new TypeMismatchedException();
                               }
                               RESULT = new Node();
                               RESULT.name = "=" + simpleExpressionToken.name;
                               RESULT.type = "BOOLEAN";
                           :}
                         | EQUAL
                           {:
                               if (true) throw new MissingOperandException();
                           :}
                         | NOTEQUAL simple_expression :simpleExpressionToken
                           {:
                               if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                                   throw new TypeMismatchedException();
                               }
                               RESULT = new Node();
                               RESULT.name = "#" + simpleExpressionToken.name;
                               RESULT.type = "BOOLEAN";
                           :}
                         | NOTEQUAL
                           {:
                               if (true) throw new MissingOperandException();
                           :}
                         | LessThan simple_expression :simpleExpressionToken
                           {:
                               if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                                   throw new TypeMismatchedException();
                               }
                               RESULT = new Node();
                               RESULT.name = "<" + simpleExpressionToken.name;
                               RESULT.type = "BOOLEAN";
                            :}
                         | LessThan
                           {:
                               if (true) throw new MissingOperandException();
                           :}
                         | LessThanOrEqual  simple_expression :simpleExpressionToken
                           {:
                               if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                                   throw new TypeMismatchedException();
                               }
                               RESULT = new Node();
                               RESULT.name = "<=" + simpleExpressionToken.name;
                               RESULT.type = "BOOLEAN";
                           :}
                         | LessThanOrEqual
                           {:
                               if (true) throw new MissingOperandException();
                           :}
                         | GreatThan simple_expression :simpleExpressionToken
                           {:
                               if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                                   throw new TypeMismatchedException();
```

```
                    }
                    RESULT = new Node();
                    RESULT.name = ">" + simpleExpressionToken.name;
                    RESULT.type = "BOOLEAN";
                :}
            | GreatThan
                {:
                    if (true) throw new MissingOperandException();
                :}
            | GreatThanOrEqual  simple_expression :simpleExpressionToken
                {:
                    if ((simpleExpressionToken.type.indexOf("INTEGER") == -1 && simpleExpressionToken.type.indexOf("ARRAY") == -1 && simpleExpre
                        throw new TypeMismatchedException();
                    }
                    RESULT = new Node();
                    RESULT.name = ">=" + simpleExpressionToken.name;
                    RESULT.type = "BOOLEAN";
                :}
            | GreatThanOrEqual
                {:
                    if (true) throw new MissingOperandException();
                :}
            ;
```

- **simple_expression:** 定义了简单表达式的规则，可以是一个项或包含加减运算符和项。在语义动作中，生成表达式节点，并进行类型检查。

```
simple_expression ::= term : termToken  term_list : termListToken
                    {:
                        RESULT = new Node();
                        RESULT.name = termToken.name + termListToken.name;
                        RESULT.type = termToken.type;
                    :}
                | PLUS term : termToken term_list : termListToken
                    {:
                        if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                            || (termListToken.type.indexOf("INTEGER") == -1 && termListToken.type.indexOf("ARRAY") == -1 && termListToken.type.indexOf("
                            throw new TypeMismatchedException();
                        }
                        RESULT = new Node();
                        RESULT.name = "+" + termToken.name + termListToken.name;
                        RESULT.type = termToken.type;
                    :}
                | MINUS term : termToken term_list: termListToken
                    {:
                        if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                            || (termListToken.type.indexOf("INTEGER") == -1 && termListToken.type.indexOf("ARRAY") == -1 && termListToken.type.indexOf("
                            throw new TypeMismatchedException();
                        }
                        RESULT = new Node();
                        RESULT.name = "-" + termToken.name + termListToken.name;
                        RESULT.type = termToken.type;
                    :}
                ;
```

- **term:** 定义了项的规则，可以是一个因子或包含乘除模运算符和因子。在语义动作中，生成项的节点，并进行类型检查。

```
        | term: termToken DIV factor: factorToken
            {:
                if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                    || (factorToken.type.indexOf("INTEGER") == -1 && factorToken.type.indexOf("ARRAY") == -1 && factorToken.type.indexOf("RECORD") == -1)) {
                    throw new TypeMismatchedException();
                }
                RESULT = new Node();
                RESULT.name = termToken.name + "/" + factorToken.name;
                RESULT.type = "INTEGER";
            :}
        | term: termToken MOD factor: factorToken
            {:
                if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                    || (factorToken.type.indexOf("INTEGER") == -1 && factorToken.type.indexOf("ARRAY") == -1 && factorToken.type.indexOf("RECORD") == -1)) {
                    throw new TypeMismatchedException();
                }
                RESULT = new Node();
```

3 附录：语法制导翻译模式

```
            RESULT.name = termToken.name + "MOD" + factorToken.name;
            RESULT.type = "INTEGER";
        :}
    | term: termToken AND factor: factorToken
        {:
            if ((termToken.type.indexOf("BOOLEAN") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                || (factorToken.type.indexOf("BOOLEAN") == -1 && factorToken.type.indexOf("ARRAY") == -1 && factorToken.type.indexOf("RECORD") == -1)) {
                throw new TypeMismatchedException();
            }
            RESULT = new Node();
            RESULT.name = termToken.name + "AND" + factorToken.name;
            RESULT.type = "BOOLEAN";
        :}
    | TRUE
        {:
            RESULT = new Node("", "BOOLEAN");
        :}
    | FALSE
        {:
            RESULT = new Node("", "BOOLEAN");
        :}
     ;
```

- **term__list:** 定义了项列表的规则，可以包含多个加减运算符和项。在语义动作中，进行类型检查和生成相应节点。

```
term_list ::= {: RESULT = new Node(); :}
        | term_list: termListToken PLUS term: termToken
            {:
                if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                    || (termListToken.type.indexOf("INTEGER") == -1 && termListToken.type.indexOf("ARRAY") == -1 && termListToken.type.indexOf("RECORD")
                    throw new TypeMismatchedException();
                }
                RESULT = new Node();
                RESULT.name = termListToken.name + "+" + termToken.name;
                RESULT.type = termToken.type;
            :}
        | term_list: termListToken PLUS END
            {:
                if (true) throw new MissingOperandException();
            :}
        | term_list: termListToken MINUS term: termToken
            {:
                if ((termToken.type.indexOf("INTEGER") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                    || (termListToken.type.indexOf("INTEGER") == -1 && termListToken.type.indexOf("ARRAY") == -1 && termListToken.type.indexOf("RECORD")
                    throw new TypeMismatchedException();
                }
                RESULT = new Node();
                RESULT.name = termListToken.name + "-" + termToken.name;
                RESULT.type = termToken.type;
            :}
        | term_list: termListToken OR term: termToken
            {:
                if ((termToken.type.indexOf("BOOLEAN") == -1 && termToken.type.indexOf("ARRAY") == -1 && termToken.type.indexOf("RECORD") == -1)
                    || (termListToken.type.indexOf("BOOLEAN") == -1 && termListToken.type.indexOf("ARRAY") == -1 && termListToken.type.indexOf("RECORD")
                    throw new TypeMismatchedException();
                }
                RESULT = new Node();
                RESULT.name = termListToken.name + "OR" + termToken.name;
                RESULT.type = termToken.type;
            :}
        ;
```

- **factor:** 定义了因子的规则，可以是标识符、数字或包含括号的表达式。在语义动作中，生成因子的节点，并进行类型检查。

```
factor ::= IDENTIFIER: identifier selector: selectorValue
        {:
            RESULT = new Node();
            if (selectorValue.type == "") {
                RESULT.name = identifier;
                RESULT.type = "INTEGER";
            } else {
                RESULT.name = identifier + selectorValue.name;
                RESULT.type = selectorValue.type;
```

3 附录：语法制导翻译模式

```
            }
        :}
    | NUMBER: number
        {:
            RESULT = new Node(number, "INTEGER");
        :}
    | LeftParenthesis expression: expressionToken RightParenthesis
        {:
            RESULT = new Node();
            RESULT.name = "(" + expressionToken.name + ")";
            RESULT.type = expressionToken.type;
        :}
    | LeftParenthesis expression: expressionToken END
        {:
            if (true) throw new MissingRightParenthesisException();
        :}
    | NOT factor: factorToken
        {:
            if (factorToken.type != "BOOLEAN")
                throw new TypeMismatchedException();
            RESULT = new Node();
            RESULT.name = "~" + factorToken.name;
            RESULT.type = factorToken.type;
        :}
    | NUMBER NUMBER
        {:
            if (true) throw new MissingOperatorException();
        :}
    | IDENTIFIER IDENTIFIER
        {:
            if (true) throw new MissingOperatorException();
        :}
    ;
```

- **selector:** 定义了选择器的规则，可以是字段选择或数组选择。在语义动作中，生成选择器的节点，并进行类型检查。

```
selector ::= selector: previerSelectorName POINT IDENTIFIER: selectorIndetifier
             {:
                 RESULT = new Node();
                 RESULT.name += previerSelectorName.name + "." + selectorIndetifier;
                 RESULT.type = "RECORD";
             :}
           | selector: previerSelectorName LeftBracket expression: expressionToken RightBracket
             {:
                 RESULT = new Node();
                 RESULT.name += previerSelectorName.name + "[" + expressionToken.name + "]";
                 RESULT.type = "ARRAY";
             :}
           | {: RESULT = new Node("", ""); :}
           ;
```

- **const_declarations:** 定义了常量声明的规则，可以为空或包含多个常量声明。

```
const_declarations ::=
                    | CONST const_list
                    ;
```

- **const_list:** 定义了常量列表的规则，可以包含多个常量。在语义动作中，将常量添加到符号表中。

```
const_list ::= const_list IDENTIFIER: identifierName EQUAL expression: expressionToken SEMICOLON
               {:
                   parser.symbols.addElement(new Node(identifierName, expressionToken.type));
               :}
             |
             ;
```

- **type_declarations:** 定义了类型声明的规则，可以为空或包含多个类型声明。

```
type_declarations ::=
                   | TYPE type_list
                   ;
```

3 附录：语法制导翻译模式

- **type_list:** 定义了类型列表的规则，可以包含多个类型声明。在语义动作中，将类型添加到符号表中。

```
type_list ::= type_list IDENTIFIER: identifier EQUAL special_type: typeToken SEMICOLON
              {:
                  parser.symbols.addElement(new Node(identifier, typeToken.type));
              :}
            |
            ;
```

- **var_declarations:** 定义了变量声明的规则，可以为空或包含多个变量声明。

```
var_declarations ::= VAR var_list
                   |
                   ;
```

- **var_list:** 定义了变量列表的规则，可以包含多个变量声明。在语义动作中，将变量添加到符号表中。

```
var_list ::= var_list identifier_list: identifierList COLON special_type: typeToken SEMICOLON
             {:
                 for (int i = 0; i < identifierList.size(); i++)
                     parser.symbols.addElement(new Node(identifierList.elementAt(i).name, typeToken.type));
             :}
           |
           ;
```

- **module_body:** 定义了模块体的规则，可以为空或包含语句序列。

```
module_body ::= BEGIN statement_sequence
              |
              ;
```

- **readwritestatement:** 定义了读写语句的规则，包含读和写操作。在语义动作中，处理缺少操作数和括号的异常。

```
readwritestatement ::= READ LeftParenthesis RightParenthesis {: if (true) throw new MissingOperandException(); :}
                     | READ {: if (true) throw new MissingLeftParenthesisException(); :}
                     | READ LeftParenthesis expression: expressionToken RightParenthesis
                     | READ IDENTIFIER RightParenthesis {: if (true) throw new MissingLeftParenthesisException(); :}
                     | READ LeftParenthesis IDENTIFIER {: if (true) throw new MissingRightParenthesisException(); :}
                     | WRITE {: if (true) throw new MissingLeftParenthesisException(); :}
                     | WRITE IDENTIFIER RightParenthesis {: if (true) throw new MissingLeftParenthesisException(); :}
                     | WRITE LeftParenthesis IDENTIFIER {: if (true) throw new MissingRightParenthesisException(); :}
                     | WRITE LeftParenthesis RightParenthesis {: if (true) throw new MissingOperandException(); :}
                     | WRITE LeftParenthesis expression: expressionToken RightParenthesis
                     | WRITELN
                     ;
```

- **if_statement:** 定义了条件语句的规则，包含条件表达式、then 分支、elsif 分支和 else 分支。

```
if_statement ::= IF expression: expressionToken THEN statement_sequence elsif_statement else_statement END
               ;
```

- **elsif_statement:** 定义了 elsif 分支的规则，可以包含多个 elsif 分支。

```
elsif_statement ::=
                  | elsif_statement ELSIF expression: expressionToken THEN statement_sequence
                  ;
```

- **else_statement:** 定义了 else 分支的规则，可以为空或包含语句序列。

```
else_statement ::=
                 | ELSE statement_sequence
                 ;
```

3 附录：语法制导翻译模式

- **while_statement:** 定义了循环语句的规则，包含条件表达式和循环体。

```
while_statement ::= WHILE expression: expressionToken DO statement_sequence END
                  ;
```

- **assignment:** 定义了赋值语句的规则，包含标识符、选择器和表达式。在语义动作中，进行类型检查。

```
assignment ::= IDENTIFIER: identifier selector: selectorName ASSIGNMENT expression: expressionToken
                {:
                    boolean found = false;
                    for (int i = 0; i < parser.symbols.size(); i++) {
                        if (parser.symbols.elementAt(i).name.equals(identifier))
                            if (parser.symbols.elementAt(i).type != expressionToken.type)
                                throw new TypeMismatchedException();
                    }
                :}
             ;
```