# 编译课程项目报告

——语法分析器

靳帅祥

16307130023@fudan.edu.cn

2019 年 1 月 2 日

## 目录

# 1 实验目的

- 学习 LALR 文法和 Bison 语法分析生成器的用法
- 熟悉 PCAT 的语法
- 使用 bison 为 PCAT 编写语法生成器
- 利用上述过程构建抽象语法树并输出

# 2 概述

语法分析器是 PCAT 的编译器中非常重要的一部分。通常一个标准的编译器前端需要包含词法分析器、语法分析器、中间代码生成这几个部分，其中语法分析器的主要任务是读入词法单元流、判断输入程序是否匹配程序设计语言的语法规范，并在匹配规范的情况下构建起输入程序的静态结构。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。本实验中构建的是抽象语法树，将传递给后续的中间码生成步骤。

PCAT 的语法可以使用 LALR 文法描述，对于 LALR 文法，可以采用成熟的语法分析器生成器，输入 LALR 文法，并生成词文法对应的语法分析器代码。本实验采用 Bison 来生成语法分析器，输入 PCAT 语言的 BNF 文法描述文件，生成语法分析器，并把生成的语法分析器与上一次实验中的词法分析器组合，完成了可以对源代码进行分析，输出抽象语法树的程序。

# 3 实现过程

## 3.1 语法的整理

1. 阅读《PCAT 语言参考指南》, 整理出 ebnf 范式
2. 把 ebnf 转化为 Bison 可以接受的 bnf 语法，主要是闭包、可选项的转化，发现一些重复的部分, 重复的部分虽然形式相同，但其语义可能不同，不能随意合并。
   (a) varDeclType0 & procedureDeclType0 重复
   (b) varDeclIDArray & fpSectionIDArray 重复
3. 将 fixed 的 PCAT 语法加入前面整理的 bnf 语法中，fixed 部分的主要作用是消除语法的二义性，使各个运算符号能够正确的被计算
4. 继续整理 bnf-fixed
   (a) varDeclIDClosure 和 fpSectionIDClosure 合并为 IDClosure
   (b) 把可选项后缀由 0 改成 Option
   (c) 添加 constant
   (d) varDeclType0 和 procedureDeclType0 合并为 typenameOption

   如何转换闭包和可选项将在后面介绍。

## 3.2 重写词法分析器

第一个 PJ 中的词法分析器已经不适合于这个 PJ，所以对其进行重写，对于每个匹配项返回预定义的 Token，方便语法分析其进行处理。

### 3.2.1 %option noyywrap

**yywrap()** 这一函数在文件（或输入）的末尾调用。如果函数的返回值是 1，就停止解析。因此它可以用来解析多个文件。代码可以写在第三段，这就能够解析多个文件。

由于本次试验只需要解析一个单独的文件，所以可以使用 noyywrap 在读完单个文件后，直接结束整个解析过程。

### 3.2.2 %option yylineno

Flex 提供的计算行数的工具

### 3.2.3 YY_USER_ACTION

**Info:**

The YY_USER_ACTION macro is "called" before each of your token actions and updates yylloc.

使用这个属性来计算每个 Token 的位置。

为了更方便的在 Bison 中获取信息，使用了**%locations** 这个选项提供的 **YYLTYPE** 这个结构体，其定义如下：

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

在 Flex 使用 yylloc 来设置对应 Token 的位置信息，在每次匹配成功之后，调用下面函数

```
void yylexUpdateLocation() {
  yylloc.first_line = yylineno;
  yylloc.last_line = yylineno;
  yylloc.first_column = yycolumn;
  yylloc.last_column = yycolumn + yyleng - 1;
  yycolumn += yyleng;
}
```

之后在 Bison 中就可以使用 **@$** 或者用 **@n** 来获取 Token 的位置信息。

### 3.2.4 test locations

由于现在还没有实现 Bison 部分，所以在助教提供的 demo 中来测试上述方法是否可以使用。

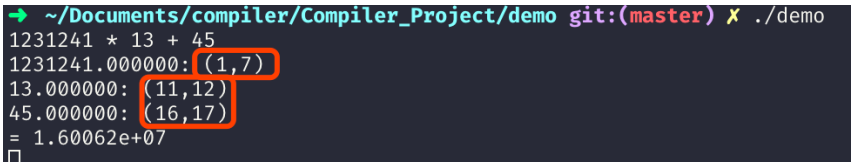


图 1: 位置信息

代码在**./lextests/demo** 中。

#### 3.2.5 test scaner

在**./lextests** 文件中编写了测试文件来进行测试，测试结果在**./lextests/out** 中。

Command Line

```
$ chmod +x lextest.sh
$ ./lextest.sh
```

### 3.3 语法分析器

#### 3.3.1 Bison 介绍

Bison 是一个 GNU 系列项目中的语法分析器生生成工具，它接受一个 LALR 文法，并生成一份与之对应的语法分析器源码。Bison 接受的 LALR 文法以 BNF 范式进行描述，生成语法分析器的代码。

#### 3.3.2 语法描述文件

Bison 接受 BNF 范式描述的 LALR 文法，描述包含在文法描述文件中。
文件的结构和 Flex 类似，也是分为三个部分。

```
/* 定 义 部 分 */
%%
/* 规 则 部 分 */
%%
/* 用 户 函 数 部 分 */
```

#### 3.3.3 定义部分

声明部分主要负责引用头文文件，设置 Bison 的参数，定义 Token 等工作。在文件的顶部可以引用头文件，目的是允许后续语义动作中调用其他源码文件中的函数，这一部分使用一对“%”与“%”符号来标识，其内部可以写入 C 语言 #include 语句、函数或者变量的定义。

此后设置 Bison 参数，定义 Token 和非终结符，这类设置通常以“%”开头。在 Bison 中，每一个 Token 或非终结符都对应一个数据类型，这就允许在进行语法分析的过程中，利用每个符号所对应的信息来构造抽象语法树。在此实验中构造语法树的方法也是如此，每一个符号都是语法树的一个节点，在分析过程中利用语义动作来构造语法树。

“%token”指令用用于定义 Token 及其类型，与词法分析器中输出的 Token 类型相对应。

“%type”指令用用于定义非非终结符的类型，通常可以定义为抽象语法树的节点。

此外，“%left”、“%right”等指令定义运算符的结合顺序，用用于消除冲突。由于修改过的语法已经考虑了运算符号的优先级，所以本次试验中未使用到这些指令。

### 3.3.4 规则部分

规则部分是整个描述文件中最重要的部分，

规则部分是由 BNF 语法与语义动作构成的，每一条文法规则都可以使用 BNF 描述，又可以为规则附加一条语义规则。Bison 接受无右递归的非二义性 LALR 文法。在表达式产生冲突时，可以使用算符优先级解决冲突。

语义动作是当采用一条规则归约后就会执行的一段代码，在这段代码中可以访问所归约语句的所有信息，包括获取终结符与非终结符的值，以及设置产生生式左部非终结符的值。在本实验中，所有非终结符都保存抽象语法树节点类型的值，在语义动作中，根据不同的语句来构造不同的子树。

### 3.3.5 用户函数部分

尾部在文件中是并不是必要的。有时会希望在语法分析器中写入函数等代码，则可以把它们写在这一部分。例如语义动作中调用的函数就可以在此处实现；也可以把用到的函数写在其他的 C 或 CPP 文件之中，在规则文件中仅引用其头文件，在编译时链接后编译成一个完整的可执行文件。

## 3.4 PCAT 文法的实现

在前面语法的整理部分遗留的两个问题是闭包和可选项的 ebnf 语法转化为 bnf 语法。

闭包的转化

闭包是指允许某一个符号重复出现任意次数的形式。需要将其转化为 bnf 语法，转化规则如下：

```
ebnf: {A}
bnf:  AClosure: %empty | AClosure A;
```

由此引入了一类新的非终结符，他们表示语法树种的一组节点；除此之外，其他的非终结符则表示一个普通的语法树节点。因此，对于任何改写后的闭包 (名称后缀种有 Closure)，都采用了 astList 类型；对于其他的非终结符，都采用 ast 类型。

可选项的转化

可选项是指一个符号出现 1 次或者 0 次。

```
ebnf: [A]
bnf:  AOption: %empty | A;
```

完成这一部分的整理之后，就完成了一大部分的工作。需要对编写的语法规则进行测试。

### 3.4.1 正确性测试

在测试中，由于还没有编写任何的语义动作，会使用 Bison 默认的语义动作 { $$ = $1 } 来进行。

此时，可以将所以得非终结符都设置为相同类型，方便后面的测试。

在调试过程中发现自己编写的语法出现错误，但不知道问题出在哪里，所以需要 Bison 更加详细的调试信息。下面是几个用到的选项:

- %error-verbose： 可以让报错更加详细

```
*** "syntax error" (line: 5, token:‘i’)

*** "syntax error, unexpected IDENTIFIER, expecting BEGINT or VAR or TYPE or PROCEDURE"
```

- -report=state -v 以及 -g： 可以生成关于语法的详细信息。

  -report=state -v 可以将 Bison 生成的 LALR 分析表输出出来

  -g 可以将整个表输出位一个.dot 文件，使用 Graphviz 软件可以将图像画出来，便于查错

但由于语法很复杂，通过上面的输出并不容易找到错位在哪里，还可以使用 Bison 提供的 debug 模式，他会把每一个移进/规约操作都输出出来。开启 debug 模式的方式是使用-t 或者–debug 命令，并在主程序中设置把 yydebug 这个变量设置为 1，这是 Bison 提供的一个变量。

经过调试，发现是自己的语法规则中少写了一条，导致后面的所有语法都无效。

```
andOperand: andOperand AND relationship
| relationship // 少了这一行
;
```

修复这个错误之后就可以判断程序是否符合 PCAT 的语法，但现在还不能输出语法树。

上述测试在**./bisontests** 文件夹中。

Command Line

```
$ chmod +x build.sh
$ ./build.sh
```

测试结果在**./bisontests/out** 中，19 有语法错误，少了一个分号；20 是词法错误；其余文件都正确。

## 3.5 抽象语法树的实现

词法分析器使用用一个全局变量 yylval 来传递 Token 的值，此值是一个联合体，在 Bison 语法规则文件的定义部分定义。本实验中定义如下:

```
%union {
  char*       Tstring;
  int         Tint;
  double      Treal;
  ast*        Tast;
  astList*    TastList;
}
```

图 2: 类型定义

在以上的定义之下，可以使用 yylval.Tstring 以字符串指针的类型来设置 Token 值，同理也可以使用 yylval.Tint 以整数类型来设置 Token 值。在词法分析器的分析动作中，对字符串、整数、实数，以及标识符几种类型的 Token 添加代码，逐一对 yylval 赋值。

与 Token 关联的内容，除了值以外还有源码位置信息。在 Bison 规则文件的声明部分加入%locations 参数，就可以开启词法分析器的位置跟踪。词法分析器中，位置跟踪也是通过一个全局变量 yylloc 传递给语法分析器的。具体介绍和使用方法见 **3.2.4**。

之后在 Bison 的语法规则文件中就可以引用 Token 的位置了。Bison 文件的语义动作中，每一个终结符或非终结符都可以通过“@”符号来引用其位置信息。对于终结符，位置信息就是 Token 的位置，对于非终结符，默认会根据它所包含的第一个 Token 与最后一个 Token 来确定它的位置信息。

抽象语法树的生成是通过语义动作来完成的，对于终结符，其语义动作中利用 yylval，调用"ast.h" 中定义的语法树相关函数来创建节点。

```
ast* makeInt(const void* loc, const int x);
ast* makeReal(const void* loc, const double x);
ast* makeVar(const void* loc, const char* x);
ast* makeStr(const void* loc, const char* x);
ast* makeNode(const void* loc, const astKind, astList* args);
ast* makeNodeR(const void* locl,
               const void* locr,
               const astKind tag,
               astList* args);

ast* makeUnaryExp(const void* loc, const astKind tag, ast* val);

ast* makeBinExp(const void* loc, const astKind tag, ast* val1, ast* val2);

ast* makeStatBlock(const void* loc, astList* revstat);

ast* makeIdList(const void* locl,
                const void* locr,
                ast* firstid,
                astList* revlst);
```

图 3: ast 相关函数

对于终结符节点创建，由于只有 4 种节点，所以直接写成了 4 个函数，函数接受位置信息和终结符的值, 返回一个 ast 指针类型的树节点。以 number 为例子来说明如何创建。

语义动作中，"@$" 表示归约结果的位置信息，传递给 makeInt 函数用于构造语法树; 而"$$" 表示归约结果的值，把构造出的语法树赋予归约结果，以便后续分析进一步使用。

除了 makeInt 函数，"ast.h" 中还定义了一系列构造语法树的函数。这一类函数几乎都接受一个位置信息的指针，用于在构造语法树时保存位置数据。此外，不同的函数接受不同的其他参数。对于终结符，通常需要接受一个具体的 Token 值，例如前文提到的整数节点需要接受一个整数值，而

对于字符串节点与标识符节点，则需要接受一个字符串值; 对于非终结符，则需要接受一组 ast 指针类型的语法树节点，例如 makeBinExp 用于构造二元运算表达式。

所有的常规非终结符都是 ast 指针类型的抽象语法树节点。这类节点可以包含具体的值 (如整数、实数、字符串等)，或是一颗子树。结构体 ast 中定义了一一个 astTag 枚举字段，用于唯一确定此节点的类型，通过这个类型就可以确定节点是否为子树，以及子树的具体结构。对于每一种语法树节点都创建一个枚举值。makeInt、makeReal、makeVar、makeStr 函数为非终结符创建一个单独的节点，而 makeNode 函数可以创建一个通用的子树，还定义了一些函数来简化构造过程。

在拥有抽象语法树后，把树的内容打印出来，就可以十分方方便的检验语法分析结果是否正确。在"ast.c" 中实现了语法树的打印函数 printAst，在文法的开始符号中，创建一个打印整棵语法树的语义动作，就可以在整个代码被接受时打印出来语法树了。

在打印的过程中，根据根据当前节点的位置信息来决定是否换行缩进: 如果当前节点与上一节点不在同一行行，则换行并添加缩进，反之则不换行缩进。

**Info:**

在编译的过程用遇到了循环引用的问题，在 ast.h 中引入了 pcat.h，在 pcat.y 中引入了 ast.h; 在 ast.h 中引入了 pcat.h 是为了使用 YYLTYPE 这个类型，所以定义一个一样的类型来解决循环引用的问题。

在 mac 下使用 clang 编译也有一些问题，所以需要使用 gcc 套件进行编译。

Command Line

```
$ chmod +x build.sh
$ ./build.sh
```

输出结果在**./out** 中，对于语法错误，可以输出所在的行号以及期望的符号。

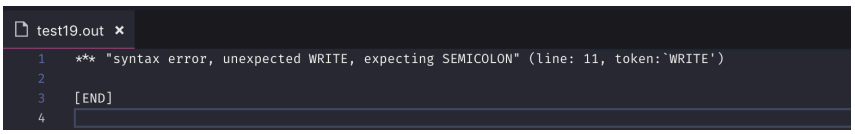


图 4: 错误提示

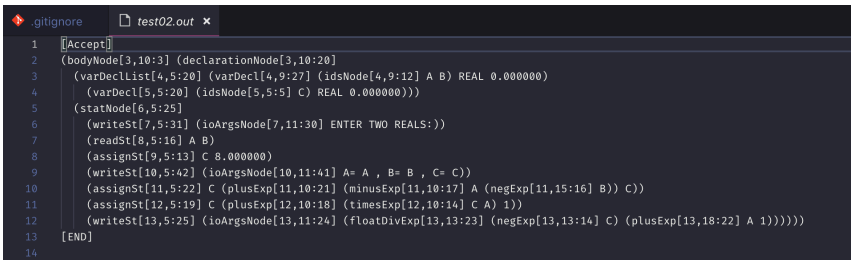


图 5: 输出结果

# 4 附录

## 4.1 EBNF

```
program          -> PROGRAM IS body ';'
body             -> {declaration} BEGIN {statement} END
declaration      -> VAR {var-decl}
                 -> TYPE {type-decl}
                 -> PROCEDURE {procedure-decl}
var-decl         -> ID { ',' ID } [ ':' typename ] ':=' expression ';'
type-decl        -> ID IS type ';'
procedure-decl   -> ID formal-params [':' typename] IS body ';'
typename         -> ID
type             -> ARRAY OF typename
                 -> RECORD component {component} END
component        -> ID ':' typename ';'
formal-params    -> '(' fp-section {';' fp-section } ')'
                 -> '(' ')'
fp-section       -> ID {',' ID} ':' typename
statement        -> lvalue ':=' expression ';'
                 -> ID actual-params ';'
                 -> READ '(' lvalue {',' lvalue} ')' ';'
                 -> WRITE write-params ';'
                 -> IF expression THEN {statement}
                    {ELSIF expression THEN {statement}}
                    [ELSE {statement}] END ';'
                 -> WHILE expression DO {statement} END ';'
                 -> LOOP {statement} END ';'
                 -> FOR ID ':=' expression TO expression [ BY expression ] DO {statement} END ';'
                 -> EXIT ';'
                 -> RETURN [expression] ';'
write-params     -> '(' write-expr {',' write-expr } ')'
                 -> '(' ')'
write-expr       -> STRING
                 -> expression
expression       -> number
                 -> lvalue
                 -> '(' expression ')'
                 -> unary-op expression
                 -> expression binary-op expression
                 -> ID actual-params // Procedure call
                 -> ID record-inits
                 -> ID array-inits
lvalue           -> ID
                 -> lvalue '[' expression ']'
                 -> lvalue '.' ID
actual-params    -> '(' expression {',' expression} ')'
                 -> '(' ')'
record-inits     -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
array-inits      -> '[<' array-init { ',' array-init } '>]'
array-init       -> [ expression OF ] expression
number           -> INTEGER | REAL
unary-op         -> '+' | '-' | NOT
binary-op        -> '+' | '-' |  '*' | '/' | DIV | MOD | OR | AND
                 -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

## 4.2 BNF

```
program                     -> PROGRAM IS body ';'
body                        -> declarationClosure BEGIN statementClosure END
declarationClosure          -> declarationClosure declaration
                            ->
statementClosure            -> statementClosure statement
                            ->
declaration                 -> VAR varDeclClosure
                            -> TYPE typeDeclClosure
                            -> PROCEDURE procedureDeclClosure
varDeclClosure              -> varDeclClosure varDecl
                            ->
typeDeclClosure             -> typeDeclClosure typeDecl
                            ->
procedureDeclClosure        -> procedureDeclClosure procedureDecl
                            ->
varDecl                     -> identifier varDeclIDClosure varDeclType0 ':=' expression ';'
varDeclIDClosure            -> varDeclIDClosure ',' identifier
                            ->
varDeclType0                -> ':' typename
                            ->
typeDecl                    -> identifier IS type ';'
procedureDecl               -> identifier formalParams procedureDeclType0 IS body ';'
procedureDeclType0          -> ':' typename
                            ->
typename                    -> identifier
type                        -> ARRAY OF typename
                            -> RECORD component componentClosure END
componentClosure            -> componentClosure component
                            ->
component                   -> identifier ':' typename ';'
formalParams                -> '(' fpSection fpSectionClosure ')'
                            -> '(' ')'
fpSectionClosure            -> fpSectionClosure ';' fpSection
                            ->
fpSection                   -> identifier fpSectionIDClosure ':' typename
fpSectionIDClosure          -> fpSectionIDClosure ',' identifier
                            ->
statement                   -> lvalue ':=' expression ';'
                            -> identifier actualParams ';'
                            -> READ '(' lvalue statementLvalueClosure ')' ';'
                            -> WRITE writeParams ';'
                            -> IF expression THEN statementClosure
                               statementElsifClosure
                               statementElse0 END ';'
                            -> WHILE expression DO statementClosure END ';'
                            -> LOOP statementClosure END ';'
                            -> FOR identifier ':=' expression TO expression statementBy0
                               DO statementClosure END ';'
                            -> EXIT ';'
                            -> RETURN expression0 ';'
statementLvalueClosure      -> statementLvalueClosure ',' lvalue
                            ->
statementElsifClosure       -> statementElsifClosure ELSIF expression THEN statementClosure
                            ->
statementElse0              -> ELSE statementClosure
                            ->
statementBy0                -> BY expression
                            ->
writeParams                 -> '(' writeExpr writeParamsExprClosure ')'
                            -> '(' ')'
writeParamsExprClosure      -> writeParamsExprClosure ',' writeExpr
                            ->
writeExpr                   -> string
                            -> expression
expression0                 -> expression
                            ->
expression                  -> number
                            -> lvalue
                            -> '(' expression ')'
                            -> unaryOp expression
                            -> expression binaryOp expression
                            -> identifier actualParams // Procedure call
                            -> identifier recordInits
```

```
                          -> identifier arrayInits
lvalue                    -> identifier
                          -> lvalue '[' expression ']'
                          -> lvalue '.' identifier
actualParams              -> '(' expression actualParamsExprClosure ')'
                          -> '(' ')'
actualParamsExprClosure   -> actualParamsExprClosure ',' expression
                          ->
recordInits               -> '{' identifier ':=' expression recordInitsPairClosure '}'
recordInitsPairClosure    -> recordInitsPairClosure ';' identifier ':=' expression
                          ->
arrayInits                -> '[<' arrayInit arrayInitClosure '>]'
arrayInitClosure          -> arrayInitClosure ',' arrayInit
                          ->
arrayInit                 -> expression
                          -> expression of expression
unaryOp                   -> '+' | '-' | NOT
binaryOp                  -> '+' | '-' |  '*' | '/' | DIV | MOD | OR | AND
                          -> '>' | '<' | '=' | '>=' | '<=' | '<>'
number                    -> INTEGER | REAL
string                    -> STRING
identifier                -> ID
```

## 4.3 BNF-fixed

```
program                   -> PROGRAM IS body ';'
body                      -> declarationClosure BEGIN statementClosure END
declarationClosure        -> declarationClosure declaration
                          ->
statementClosure          -> statementClosure statement
                          ->
declaration               -> VAR varDeclClosure
                          -> TYPE typeDeclClosure
                          -> PROCEDURE procedureDeclClosure
varDeclClosure            -> varDeclClosure varDecl
                          ->
typeDeclClosure           -> typeDeclClosure typeDecl
                          ->
procedureDeclClosure      -> procedureDeclClosure procedureDecl
                          ->
varDecl                   -> identifier IDClosure typenameOption ':=' expression ';'
IDClosure                 -> IDClosure ',' identifier
                          ->
typenameOption            -> ':' typename
                          ->
typeDecl                  -> identifier IS type ';'
procedureDecl             -> identifier formalParams typenameOption IS body ';'
typename                  -> identifier
type                      -> ARRAY OF typename
                          -> RECORD component componentClosure END
componentClosure          -> componentClosure component
                          ->
component                 -> identifier ':' typename ';'
formalParams              -> '(' fpSection fpSectionClosure ')'
                          -> '(' ')'
fpSectionClosure          -> fpSectionClosure ';' fpSection
                          ->
fpSection                 -> identifier IDClosure ':' typename
statement                 -> lvalue ':=' expression ';'
                          -> identifier actualParams ';'
                          -> READ '(' lvalue statementLvalueClosure ')' ';'
                          -> WRITE writeParams ';'
                          -> IF expression THEN statementClosure
                             statementElsifClosure
                             statementElseOption END ';'
                          -> WHILE expression DO statementClosure END ';'
                          -> LOOP statementClosure END ';'
                          -> FOR identifier ':=' expression TO expression statementByOption
                             DO statementClosure END ';'
                          -> EXIT ';'
                          -> RETURN expressionOption ';'
statementLvalueClosure    -> statementLvalueClosure ',' lvalue
                          ->
```

```
statementElsifClosure       -> statementElsifClosure ELSIF expression THEN statementClosure
                            ->
statementElseOption         -> ELSE statementClosure
                            ->
statementByOption           -> BY expression
                            ->
writeParams                 -> '(' writeExpr writeParamsExprClosure ')'
                            -> '(' ')'
writeParamsExprClosure      -> writeParamsExprClosure ',' writeExpr
                            ->
writeExpr                   -> string
                            -> expression
expressionOption            -> expression
                            ->
expression                  -> orOperand
orOperand                   -> orOperand OR andOperand
                            -> andOperand
andOperand                  -> andOperand AND relationship
relationship                -> summand '>' summand
                            -> summand '<' summand
                            -> summand '=' summand
                            -> summand '>=' summand
                            -> summand '<=' summand
                            -> summand '<>' summand
                            -> summand
summand                     -> summand '+' factor
                            -> summand '-' factor
                            -> factor
factor                      -> factor '*' unary
                            -> factor '/' unary
                            -> factor DIV unary
                            -> factor MOD unary
                            -> unary
unary                       -> '+' unary
                            -> '-' unary
                            -> NOT unary
                            -> term
term                        -> number
                            -> lvalue
                            -> identifier actualParams // Procedure call
                            -> identifier recordInits
                            -> identifier arrayInits
                            -> '(' expression ')'
                            -> constant
lvalue                      -> identifier
                            -> lvalue '[' expression ']'
                            -> lvalue '.' identifier
actualParams                -> '(' expression actualParamsExprClosure ')'
                            -> '(' ')'
actualParamsExprClosure     -> actualParamsExprClosure ',' expression
                            ->
recordInits                 -> '{' identifier ':=' expression recordInitsPairClosure '}'
recordInitsPairClosure      -> recordInitsPairClosure ';' identifier ':=' expression
                            ->
arrayInits                  -> '[<' arrayInit arrayInitClosure '>]'
arrayInitClosure            -> arrayInitClosure ',' arrayInit
                            ->
arrayInit                   -> expression
                            -> expression OF expression
number                      -> INTEGER | REAL
string                      -> STRING
identifier                  -> ID
constant                    -> TRUE
                            -> FALSE
                            -> NIL
```