

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 姓 名： | 王泽杰，张启锐，叶金韬 |
| 学 院： | 计算机科学与技术学院 |
| 系： | / |
| 专 业： | 软件工程 |
| 学 号： | 3150103504, 3150103501, 3150102210 |
| 指导教师： | 陈纯，冯雁 |

编译原理小组实验报告

1. **序言**

本实验中我们编写出了一个编译器，命名为XXCompiler。该编译器是类PASCAL的编译器，语法是PASCAL语法的子集（砍掉了很多功能），使用C++编写（词法分析使用Lex，语法分析使用Yacc，并使用Flex和Bison翻译成了C++），程序执行流程为：代码预处理，词法分析，语法分析，语法树IR生成，符号表生成，类型检查和目标代码生成。目标代码是MIPS汇编代码。

源代码有多个源文件，如下所述：

* lexer.l: 词法分析的Lex源码；
* parser.y: 语法分析与生成AST的Yacc源码；
* lexer.h与lexer.cpp: 词法分析Lex生成的代码；
* parser.h与parser.cpp: 语法分析Yacc生成的代码；
* comment.h与comment.cpp: 预处理的源代码，主要对注释进行处理；
* tree.h与tree.cpp: 语法树的定义，包括输出AST以及构建符号表、类型检查、生成代码的功能；
* optimizer.h与optimizer.cpp: 语法树优化的源码，因为时间原因，没有上优化功能，只是空实现；
* symbol.h与symbol.cpp: 符号表的定义；
* codegen.h与codegen.cpp: 目标代码生成模块；
* parser.output: 不是源代码文件，但是它说明了本编译器语法的LALR(1)的状态转换信息。

若要编译可执行程序，必须安装Make, CMake，以及g++ 5以上版本。使用2行脚本即可：

* cmake .
* make

小组内分工如下：

* 张启锐：负责中间层，编写AST，并与前端语法书制导和后端代码生成、类型检查的对接；
* 王泽杰：负责代码生成，符号表的编写；
* 叶金韬：负责代码预处理、词法分析、语法分析以及生成语法树及其输出。

1. **代码预处理**

源代码编译的第一个步骤就是代码预处理。像C语言，代码预处理主要是宏展开和注释处理，本编译器的代码预处理主要是处理掉注释，即删除注释。

注释删除功能完全实现了C语言的注释功能，即包含：

* /\*\*/ 多行注释
* // 单行注释
* // ..\ 折行注释
* 引号内的注释符合被忽略
* 符号“\”进行转义

在Python中，这个功能可以直接被一个正则表达式解决的，即引入re包，使用如下代码：

|  |
| --- |
| re.sub(r'((?<=\n)|^)[ \t]\*\/\\*.\*?\\*\/\n?|\/\\*.\*?\\*\/|((?<=\n)|^)[ \t]\*\/\/[^\n]\*\n|\/\/[^\n]\*','',code\_content) |

而C++ 11并没有原生的正则处理，所以我们使用自动机思想，将过滤注释的正则串转成有限状态自动机。

为了实现上述的功能，我们的自动机有10个状态，如下所示：

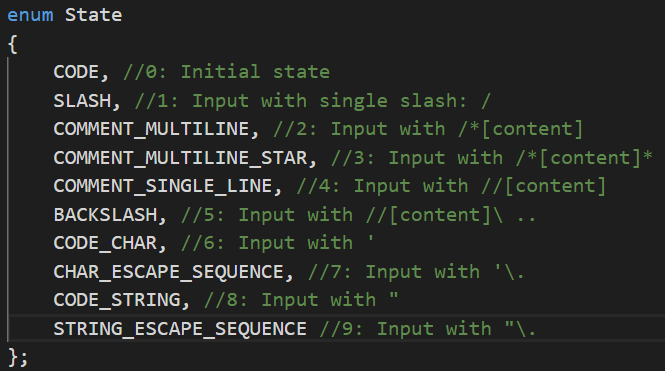


图1，注释自动机的状态

下面说明这些状态的转移，初始状态为CODE：

* 状态CODE若遇到/，说明可能遇到注释，进入状态SLASH
* 状态SLASH若遇到\*，说明进入多行注释部分，进入状态COMMENT\_MULTI\_LINE
* 状态SLASH若遇到/，说明进入单行注释部分，进入状态COMMENT\_SINGLE\_LINE
* 状态SLASH若没遇到\*或者/，说明不是注释，恢复状态CODE
* 状态COMMENT\_MULTILINE遇到了\*，说明多行注释可能要结束，进入COMMENT\_MULTILINE\_STAR
* 状态COMMENT\_MULTILINE没遇到了\*，说明多行注释继续，状态不变
* 状态COMMENT\_MULTILINE\_START若遇到\*，状态不变
* 状态COMMENT\_MULTILINE\_START若没遇到/或\*，恢复状态COMMENT\_MULTILINE
* 状态COMMENT\_SINGLE\_LINE若遇到\，说明要有折行注释，进入状态BACKSLASH
* 状态BACKSLASH若依旧遇到\，状态不变，若为其它字符，恢复COMMENT\_SINGLE\_LINE
* 状态COMMENT\_SINGLE\_LINE遇到回车，恢复CODE状态
* 状态CODE遇到’，说明可能是字符，进入CODE\_CHAR
* 状态CODE\_CHAR遇到转义符\，进入CHAR\_ESCAPE\_SEQUENCE
* 状态CHAR\_ESCAPE\_SEQUENCE遇到任何字符，都恢复状态CODE\_CHAR
* 状态CODE\_CHAR遇到’说明字符结束，恢复CODE状态
* 状态CODE\_STRING和STRING\_ESCAPE\_SEQUENCE转换的流程和CODE\_CHAR类似

状态CODE, CODE\_CHAR, CHAR\_ESCAPE\_SEQUENCE, CODE\_STRING, STRING\_ESCAPE\_SEQUENCE都需要输出当前字符，转移到上述4个状态的时候也要输出相应的字符，而其他的状态会有不同的动作，比如不输出字符，直接忽略。

状态转移图如下，为方便，就以图1注释的编号说明：

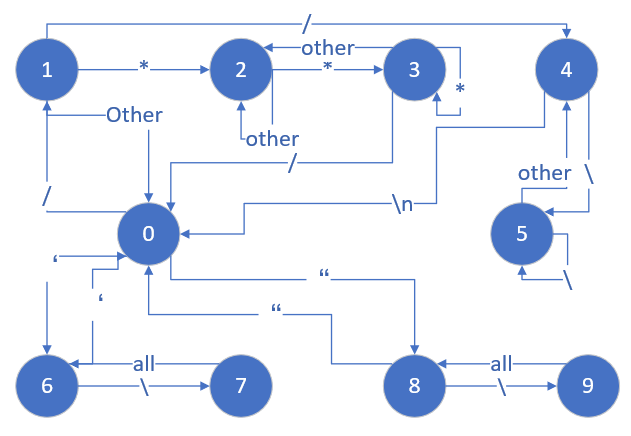


图2，注释自动机转换图

有了转换自动机图，就很方便处理并删除代码中的注释了。

1. **词法分析**

词法分析的功能是由Lex完成的，我们只需要编写Lex代码即可。我们可以通过正则表达式定义Token，然后结果Lex的转换，可以生成有限状态自动机，当文本流输入后，状态会根据自动机的定义流动，当识别成一个Token后，Lex会返回一个数字给语法分析器，告诉它这是一个什么Token，而这个数字的定义是在语法分析的Yacc源码文件中定义的（下文再叙述）。

对于我们的类PASCAL语言，我们需要对一些类型进行一些定义，如下所示，我们对数字，数字串，字母，整数，实数，字符串常量，字符常量，换行符，空白符和标识符进行了定义。其中，实数必须定义成{digits}’.’{digits}，即小数点两边都有数字，而标识符可以定义成字母和下划线开头，字母、下划线和数字的组合字符串。具体定义如下所示：

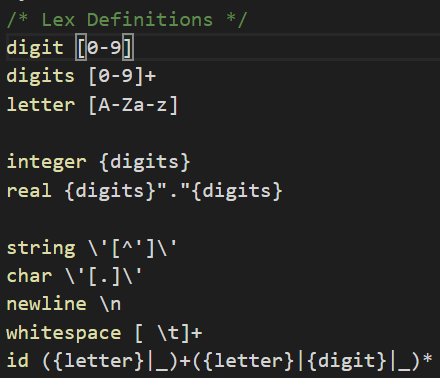


图3，正则表达式的定义

我们的类PASCAL定义了若干关键字，由于关键字很多，所以放上如下的截图。需要说明的是，我们只定义了4个原生的类型：整数、单精度浮点数、字符、字符串，另外我们只定义了2个原生过程（procedure），即读取变量的read和打印变量的write。

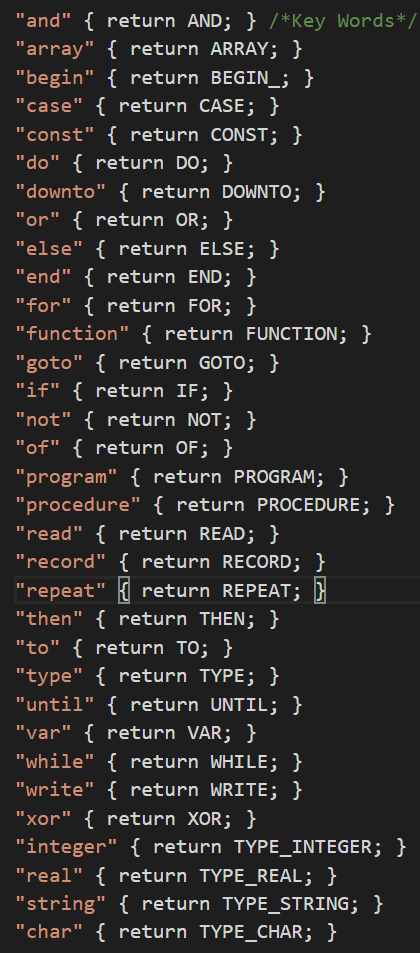


图4，关键字的定义

对于一元和二元运算符的定义如下：

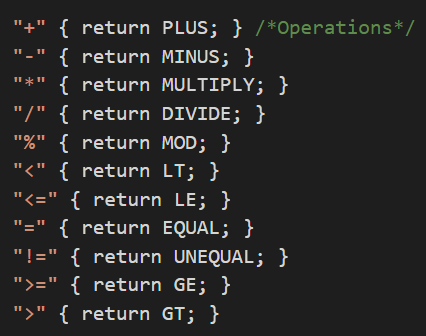


图5，运算符定义

然后是其它的符号的定义，如下所示。这里可以发现，ID这个Token不仅仅是返回这个Token ID，还会返回一个TreeNode，这个树节点记录的这个ID的值，但是一旦获取了这个值，这个TreeNode会被立即清除。

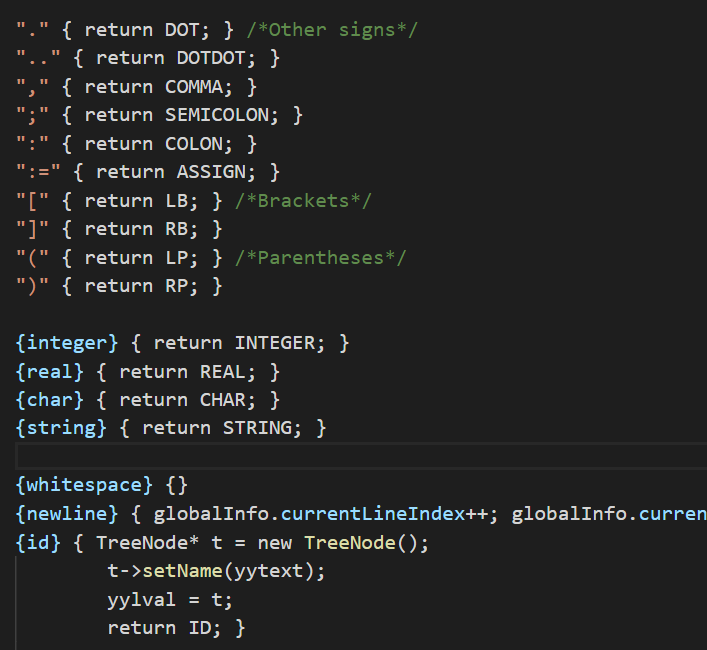


图6，其它符合的定义

由于Lex在词法分析时，其追踪分析流程的功能较为薄弱，所以我们定义了一个数据结构GlobalInfo，用于追踪数据流分析的位置，比如当前的行号，当前的Token文本以及当前Token属于当前行的第几个。数据结构定义如下：

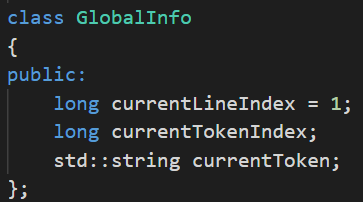


图7，跟踪词法分析的数据结构

我们还对yylex()函数进行了包装，使其能更新上述的数据结构的信息（原有的yylex()只会分析当前输入的文本，若与某个Token匹配，则返回这个Token的ID），并且能将分析的结果输出到指定的文本中去，如下所示：



图8，对yylex()进行包装

这样，词法分析部分就算是基本完成了。词法分析器可以将输入的文本转换成一个一个Token ID，传给语法分析器，同时，由于我们构造了追踪的数据结构，只要将其声明为全局变量，那么追踪信息也能被语法分析器共享，可以用于输出语法分析（尤其是规约动作）时程序分析的位置，或者输出语法错误的所在位置，对调试来说非常方便。

1. **语法分析**

本编译器使用Yacc作为语法分析的工具。Yacc输入是巴克斯范式的语法规则和语法规约的处理代码，输出的是表驱动的编译器，我们这里是输出一颗语法树。而且Yacc是采用LALR(1)的语法分析方法，是从底向上分析，使用移入和规约的动作，并且当出现SR冲突的时候，默认执行Shift（下文会涉及到）。

本编译器采用的语法是PASCAL的子集，不过删去了很多功能，所以是比较简单的。当然首先是定义Token，这样语法分析器就能识别出词法分析器提供给它是什么类型的Token。

程序的结构是：program <program\_name>; <block> .。主要的重点在于block的语法定义。

**4.1. block的结构**

程序的主体主要由常量声明，类型声明，变量声明，函数和过程声明以及程序执行体。前4个是可选的，但是它们的相对位置是不能改变的，否则会出现语法错误。

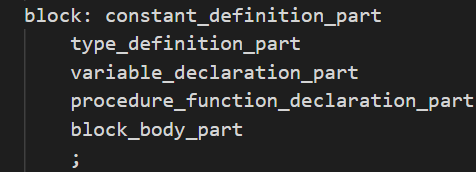


图9，block的结构说明

**4.2. 常量声明**

常量声明这里只支持原生的4种类型的常量声明（即整数，浮点数，字符，字符串），语法类似与下面的写法，每个语句后面都必须加分号，而常量类型是可以通过词法分析器来自动推导出来的：

|  |
| --- |
| const  id1 = const\_val1;  id2 = const\_val2;  … |

在Yacc中，声明也变得非常简单，如下所示，注意常量声明是可选的：

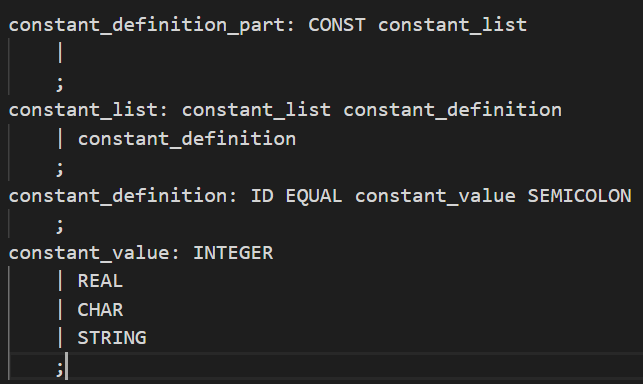


图10，常量声明

**4.3. 类型声明**

类型声明的格式和常量声明的格式类似。不过这里类型声明的种类就可以多样化了。我们把类型分为了4个大类（在Yacc代码里，我们把它们归结为type\_denoter）：

* 简单类型，包含4类原生类型，自定义的类型（用ID表示），以及枚举类型（类似于( id1, id2, … )）
* 范围类型，上下确界都是用常量表示，格式为const\_value1..const\_value2
* Record类型，类似于C语言的struct，其文法定义如下所示，注意Record的最后一个field句子末尾是不能加分号的，否则会出现语法错误：

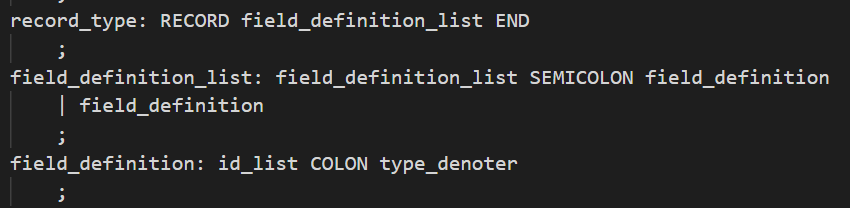


图11，Record类型的定义

* 数组类型，其文法定义如下所示：

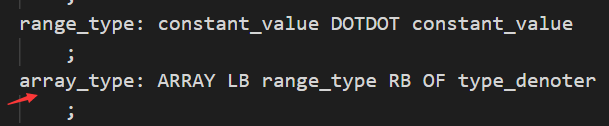


图12，数组的文法定义

最后类型声明列表的文法定义如下，它也是一种列表的形式。该区域是可选的，但是必须出现在常量声明之后。

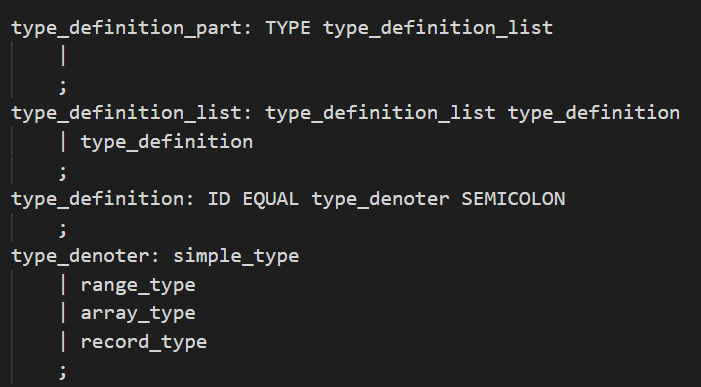


图13，类型声明区域的文法描述

**4.4. 变量声明**

变量声明的语法格式也是类似的，可以声明的类型是原生类型以及类型定义部分已经定义的类型，如下面的样例代码：

|  |
| --- |
| var  id1 : type;  id2 : type;  … |

其Yacc文法说明也是非常的简单，如下所示，这里不再详细描述：

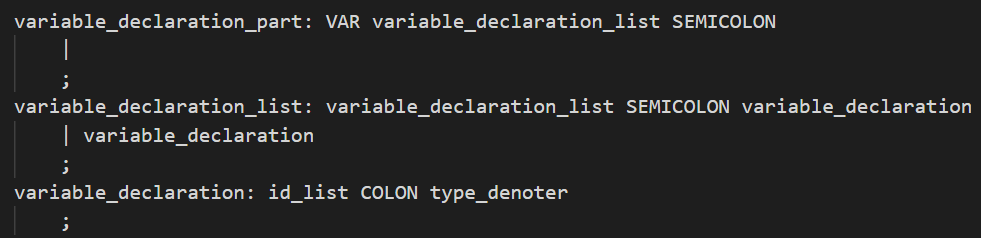


图14，变量声明部分文法描述

**4.5. 过程与函数声明**

在本编译器中，过程（procedure）是没有返回值的方法，而函数（function）是有返回值的方法。在过程和函数声明区域里，过程和函数没有声明的顺序规定，但是过程和函数并不支持嵌套，尽管在进行语法分析以及AST生成的时候可能受支持，但是在语义分析中，这个功能并不支持。

此外，函数和过程的参数类型以及函数的返回类型（返回值由最后一个语句决定），只支持上述的简单类型（见4.3小节），因为当我们定义参数的语法时，只支持了简单类型，而不支持其它的类型（比如数组）。函数的参数必须一个一个声明，不能像PASCAL一样可以用一个列表的形式声明。

这个区域的声明由procedure\_function\_declaration的列表组成，而它又可以是procedure\_declaration或者function\_declaration。它们的文法定义如下图所示：

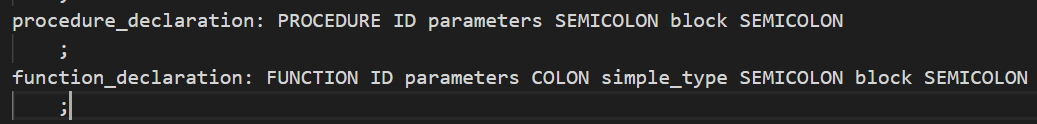


图15，函数与过程的文法定义

所以一个函数和过程的样例代码可以是如下的：

|  |
| --- |
| procedure proc(num1:integer; num2:integer);  begin  // statement list  end;  function func(num1:integer; num2:integer):char  begin  // statement list  end |

**4.6. 执行体(body)**

执行体是以begin关键字打头，end关键字作为结尾，中间是一系列statement组成的语句块。而statement是支持标签跳转的（标签是个字符串），即statement是可以由label或没有label的，所以我们抽出了一个新的因子stmt，它就是没有label的。

而stmt会被分为以下几种类型：赋值语句、调用过程语句、if语句、repeat语句、for语句、while语句、case语句、goto语句以及复合语句。其中最后的复合语句（compound\_statement）和执行体（body）的结构是一样的。

* 赋值语句主要是定义左值的访问方式，这里左值可以是一个变量、数组的元素（以id[index]形式访问）以及record的一个field（以id.id的形式访问）。
* if语句的格式是if <cond\_expr> then <statement> [else <statement>]，其中else字句是可选的。假如不做任何处理，这里会出现一个SR冲突，可以在Yacc声明代码中添加代码%right THEN ELSE强制其进行移入，从而解决冲突（不处理其实也是可以的，因为默认走移入）。
* 调用过程语句格式为proc\_name([args])，本语言可以使用的内置过程只有read()和write()，分别是读入一个变量和打印一个变量的值。
* 循环调用语句分为repeat，for和while，第一个和第三个是由一个表达式控制状态，而for可以指定条件变量是递增还是递减（即指明to还是downto），其文法描述如下：

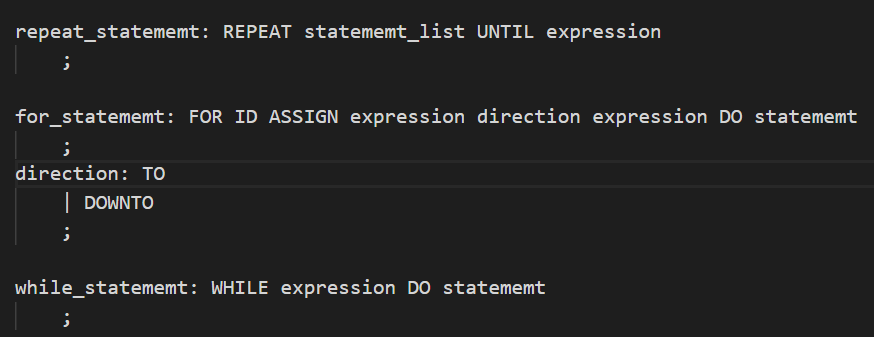


图16，循环语句文法

* case语句文法如下所示，注意的是，每个case\_item只支持常量值，且最后一个case\_item最后不能加分号，否则语法错误。

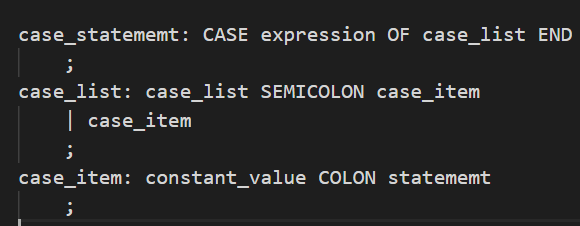


图17，case语句文法定义

- goto语句很简单，只需goto <label>即可。label是要在整个程序中有定义的，这会在语义分析中检查。

**4.7. 表达式**

表达式主要处理的问题是优先级（即二义性问题）。Yacc可以有2种不同的方式，一种是在开头用%left和%right声明操作符的结合性和优先级，另一种是构造新的类型结点。我们采用了后者，因为用这种写法构造语法树会更加清晰。这里我们借鉴了《编译原理》（第2版，龙书，中文184页）的写法，原本只定义了expression，现在多添加了3类结点（expr， term， factor），将运算符分为4个优先级，优先级从高到底如下：

* not, -（一元运算）
* \*, /, %, and
* +, -, or, xor
* <, <=, =, >=, >, !=

根据龙书的写法，我们仿照后得到下面的文法描述，它就很好地解决了优先级和文法二义性的问题。

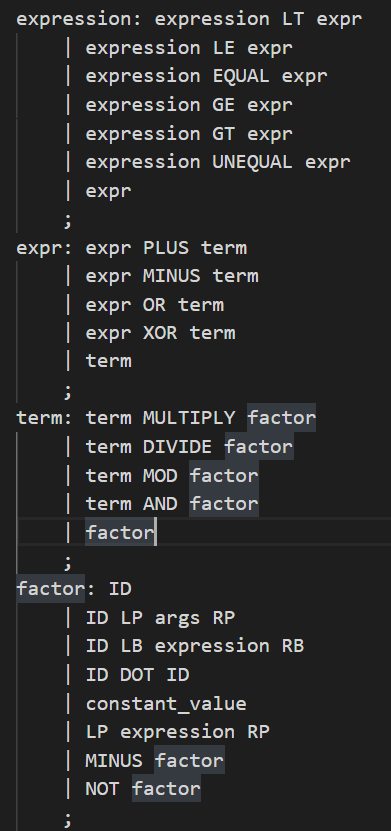


图18，解决表达式操作符的优先级

上述基本完成了整个编译器的语法分析（定义）部分。根据Yacc分析的结果，我们定义的文法会有2个SR冲突，但是在报告文件里，我们可以看到，它默认走了移入，这和我们期望的动作是一致的，所以不需要进一步改动了。

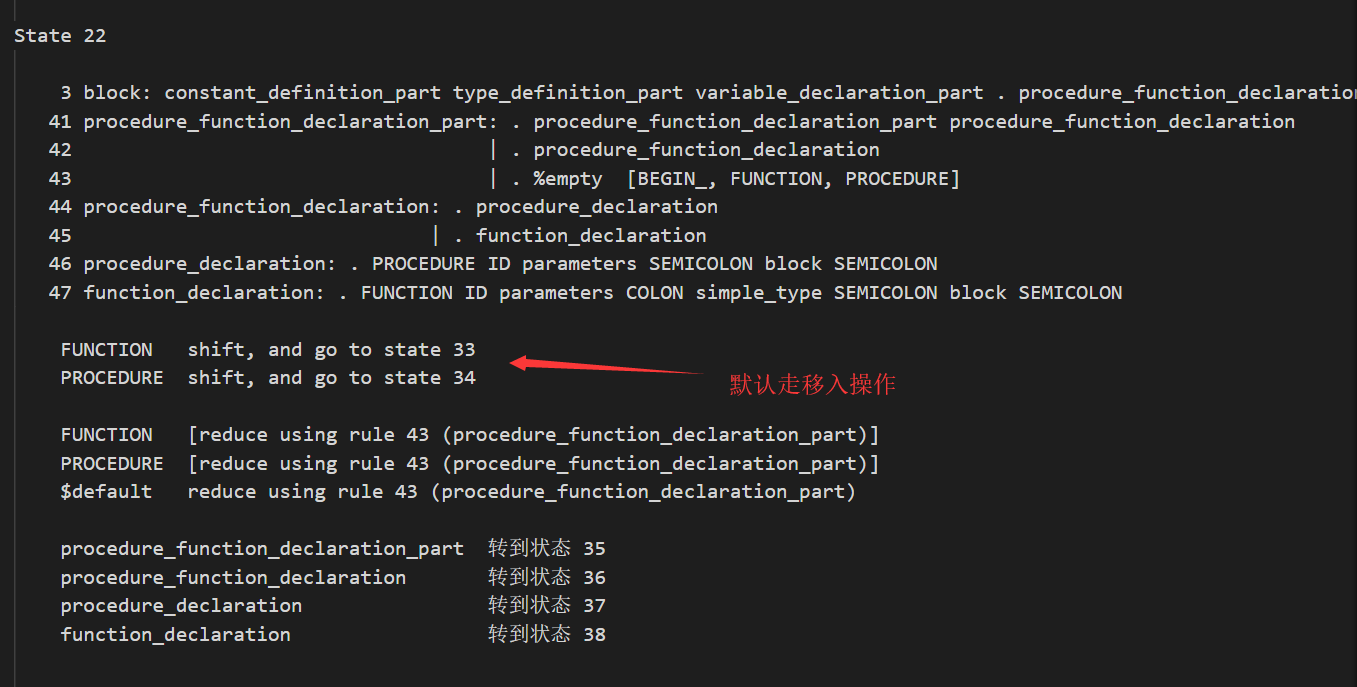


图19，SR冲突的处理

1. **AST生成（语法制导翻译）**

由于我们使用C++，所以我们可以很好地利用其面向对象的特性。所以我们定义了一个很庞大的语法树结点的继承结构。当语法树生成后，我们可以判断某个结点是哪个类的实例，在符号表生成，语义分析等阶段执行特定的分析和生成动作。

我们将树的结点分为以下几类，具体的继承关系可看下图：

* ProgramBodyTreeNode: 程序主结点，它作为语法树的根
* ExprTreeNode: 表达式的结点
* FuncOrProcTreeNode: 函数或者过程的结点
* IDTreeNode: 一个变量/常量元素的结点
* ListTreeNode: 列表结点
* StatementTreeNode: 语句的结点
* TypeTreeNode: 类型的结点

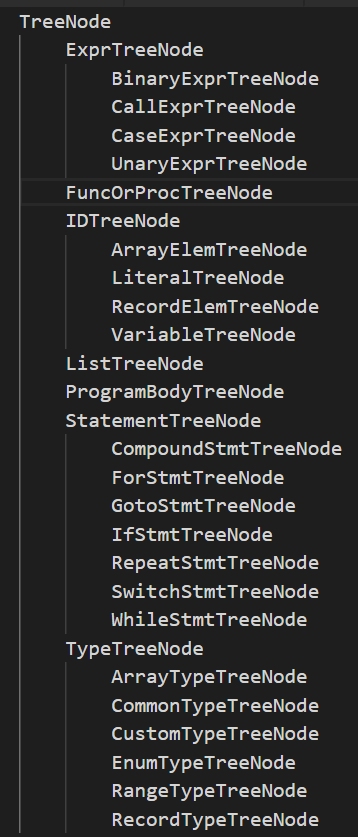


图20，语法树结点的继承结构

语法分析需要构建一棵语法树，所以

由于Yacc使用的是LALR(1)文法，所以我们自底向上说明各个结点在语法分析中的使用。

**5.1. 表达式结点**

表达式结点有4类：

* BinaryExprTreeNode：二元表达式的结点，它接受2个树结点作为左值和右值，并接受一个操作符，因此可以递归地计算这个结点的值。制导翻译时，在使用二元操作符的时候应用这个结点，可以这样操作：

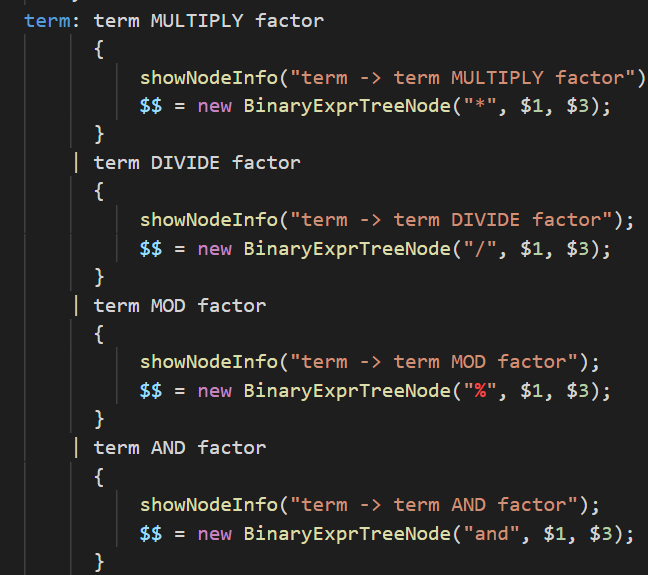


图21，二元操作符语法制导翻译

* CaseExprTreeNode：在制导case\_item的时候使用它。它接受一个case常量的树节点，任何接受一个StatementTreeNode结点，表面后面执行的动作，可以这么使用：

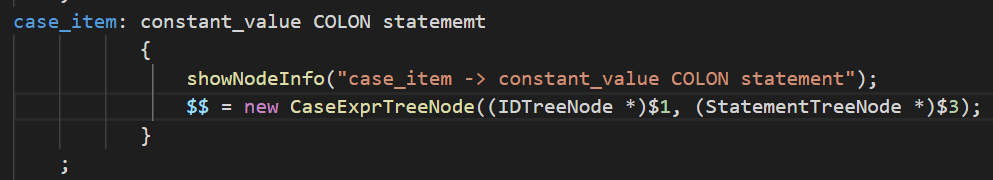


图22，制导翻译case\_item

* CallExprTreeNode：调用函数时应用的结点，调用函数会返回值，所以可以当作一个表达式看待。它接受一个函数名，参数列表（树节点列表），并表明它是否为函数；
* UnaryExprTreeNode：使用一元操作符的时候使用，它相比BinaryExprTreeNode少接受一个操作数而已。

后两个的制导可以这么使用，以生成对应的ExprTreeNode：

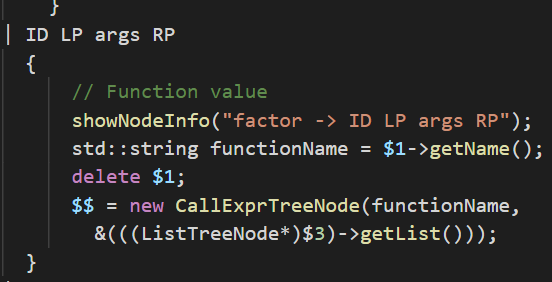
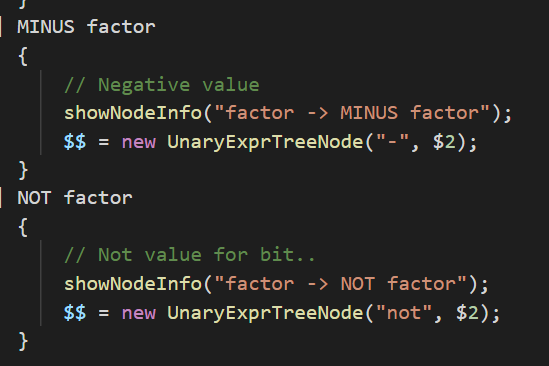
 

图23，一元操作符和函数调用表达式的制导翻译

**5.2. 变量/常量等元素**

这里表示IDTreeNode，可以有4类：

* VariableTreeNode: 表示一个变量的树节点。比如我们定义了一个变量var a : integer，那么变量a会被记录成这类结点。它接受一个变量名，一个类型结点TypeTreeNode以记录变量的类型，当它为空时，我们会做类型检查，此外还要声明它是否被赋值成常量（这会在constant\_definition\_part中声明常量时使用）；
* LiteralTreeNode：表示一个字面量的树节点，它仅仅表示字面量，和变量和常量没什么关系。它仅仅接受一个字面量值和类型，默认用字符串保存。我们可以根据类型将其转成整数或者浮点数；它只在表名一个字面量的时候使用，如下所示：

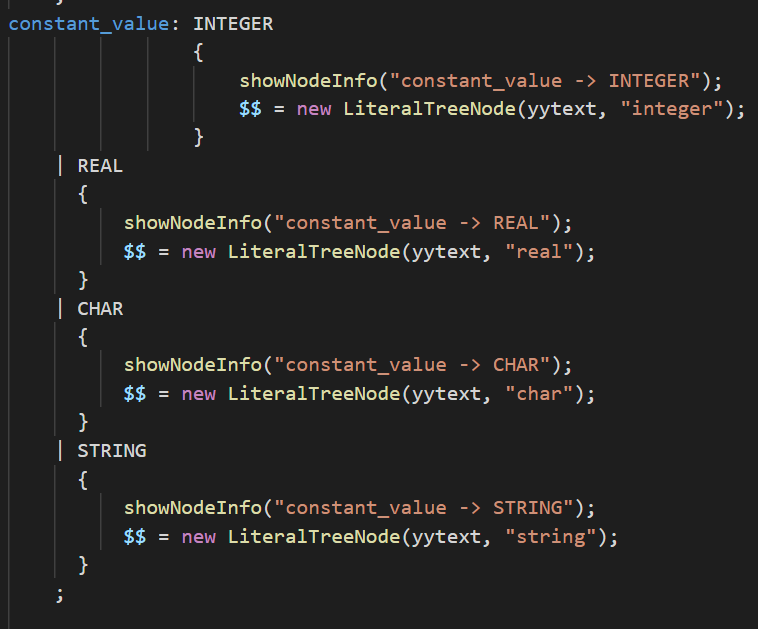


图24，字面量的制导翻译

* ArrayElemTreeNode：表示一个数组元素的树节点，它接受一个数组名，任何再接受一个ExprTreeNode作为其下表，下表从0开始计数；
* RecordElemTreeNode：表示一个Record的一个field元素，它只接受一个record名字和它下面的field名字，在后面这些内容都会做检查，保证其有效。

除了字面量的制导，其它3类结点使用可以再下面的例子中体现出来，制导还是非常方便的。

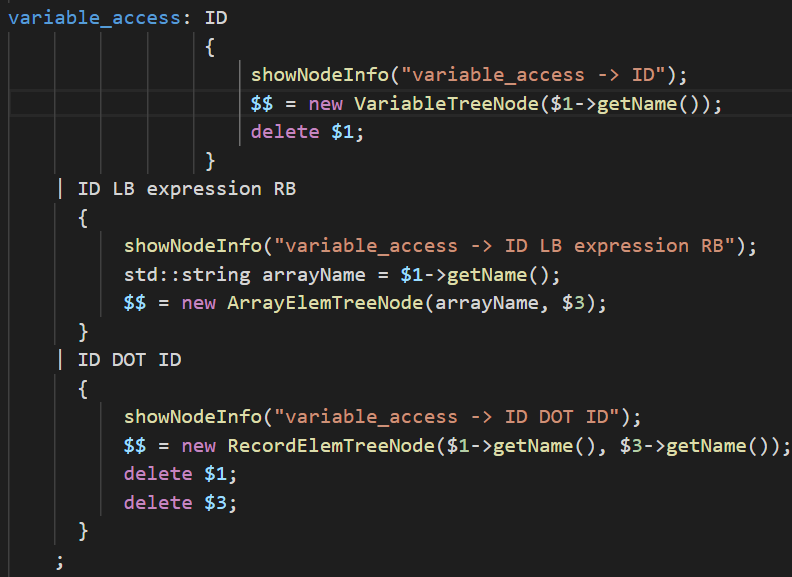


图25，其它3类结点的制导翻译

**5.4. 类型结点**

根据我们语法的定义，我们总共有6类结点，用于声明类型：

* CommonTypeTreeNode: 表示系统自带的原生四种类型的结点，只需接受一个类型名即可；
* RangeTypeTreeNode: 表示范围的类型结点，这个结点并不直接对外充当一个真正的类型，而只是表达一个范围，它只接受2个树节点，分别为上界和下界；
* ArrayTypeTreeNode：表示数组类型的结点。这里接受2个结点，一个是上面的RangeTypeTreeNode表面数组下标的范围，另一个是数组类型的结点，是CommonTypeTreeNode，即只支持创建系统自带原生4种类型的数组；
* RecordTypeTreeNode：表示Record的数据结构类型的结点，它只需要一个参数就是一个有当前record所有字段结点的列表；
* EnumTypeTreeNode：表示枚举类型的结点，它需要一个枚举类型名，还需要一些枚举，被存在ListTreeNode中，这个结点会保存一个结点列表，列表中包含了枚举的结点信息；
* CustomTypeTreeNode：表示用户自定的类型，它需要一个类型名，如果从原生类型衍生而来的，还会有TypeTreeNode结点。在制导的时候，若声明一个自定的类型且和原生的类型没有关系，那么TypeTreeNode字段可能为空，需要后面进行类型检查。

对于这些类型结点的制导翻译也和上面类似，只要在Yacc中创建这些树结点，并给构造函数必要的信息，这些信息都是可以从自底向上语法分析中得到的。如下是几个类型制导翻译的例子，具体还是请看Yacc代码文件：

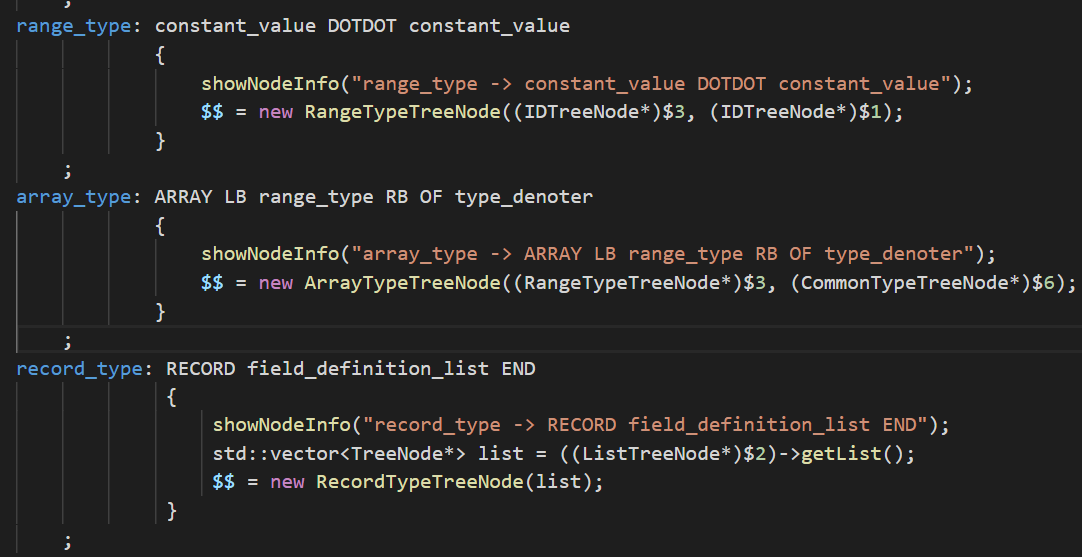


图26，类型结点语法制导

**5.5. 语句结点**

根据上面的定义，我们配套有7个语句类型结点，分别为：

* CompoundStmtTreeNode: 复合语句结点，里面就只有一个ListTreeNode，存了一个包含许多statement列表，当然这些statement也可以是复合语句。这就类似设计模式中的组合模式，而组合模式的构建也可以看成一颗树，语句执行的顺序只要前序遍历这棵树就可以了；
* IfStmtTreeNode：if语句结点，它接受一个ExprTreeNode作为判断条件，然后接受一个StatementTreeNode作为其满足条件的执行语句，可选另一个StatementTreeNode作为else字句的执行语句；
* RepeatStmtTreeNode：repeate语句的结点，同样它接受一个ExprTreeNode作为判断条件，然后接受一个StatementTreeNode作为其满足条件的循环体；
* WhileStmtTreeNode: while语句的结点，和repeat一样，同样它接受一个ExprTreeNode作为判断条件，然后接受一个StatementTreeNode作为其满足条件的循环体；
* SwitchStmtTreeNode：switch语句的结点，它首先接受一个ExprTreeNode作为判断的条件变量，然后再提供一个ListNode作为case items，里面存放了一系列的CaseExprTreeNode
* ForStmtTreeNode: for语句的结点，该结点构成略微复杂，首先需要一个ExprTreeNode作为循环初始值，通常是一个赋值语句，然后还需要一个TreeNode作为循环的结束值，此外还需要接受一个方向（可以是to也可以是downto），最后是一个StatementTreeNode作为循环体；
* GotoStmtTreeNode: goto语句结点，只需要给goto的标签label即可。

语法制导翻译这些语句的时候，也是类似的，只要往构造函数填入对应的参数即可，而参数都是可以从自底而上分析得到的。比如for语句的语法制导翻译，只需要把分析来的初始值结点，方向，结束值结点和循环体结点注入到构造函数，然后返回即可，如下所示，是非常简单的：

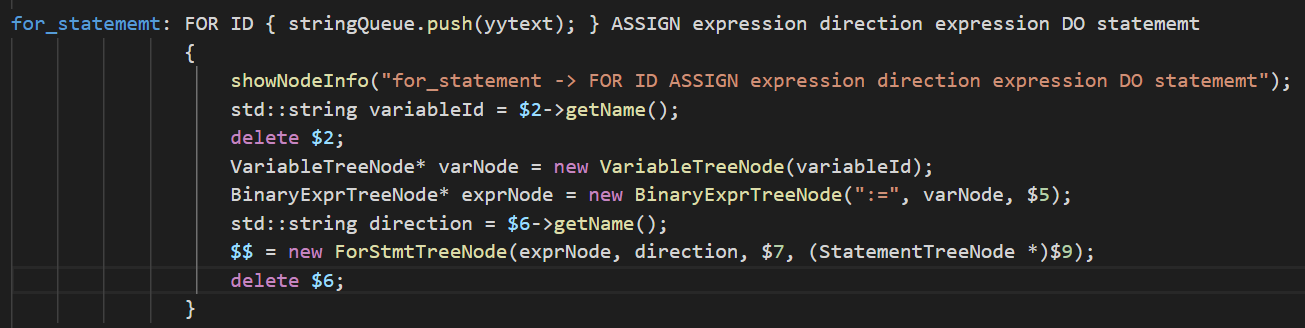


图27，for语句的制导翻译

**5.6. 构建完整的语法树**

我们把ProgramBodyTreeNode作为整个语法树的根结点，而它又有5个ListTreeNode组成，分别代表了常量声明列表，类型声明列表，变量声明列表，函数过程声明列表以及执行体。而这些元素都可以用列表的形式表示，即树结点的列表。

上述的每个部分。都会返回一个ListTreeNode，分别存储了一系列的常量结点、类型声明结点、变量声明结点、函数过程声明结点和执行语句结点。然后我们把这5个结点都注入到ProgramBodyTreeNode中去，然后把它赋值到全局变量root中，供下面的语义分析使用。

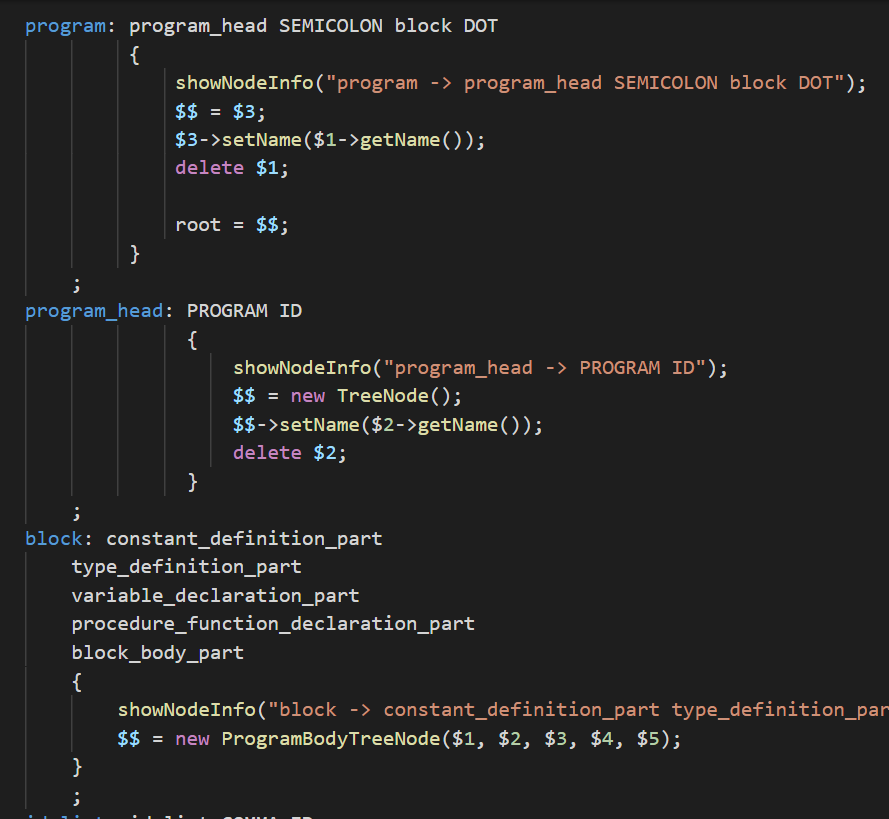


图28，最后的大语法树的生成

由于我们定义了很多种类的结点，所以制导翻译的时候，很好地和语法的定义结合在了一起，不需要额外的操作，只需在构造方法中提供对应的信息即可。此外，由于利用了OO特性，后面进行符号表的生成、类型检查以及目标代码生成的时候，也会比较方便维护管理。

而最后是关于整个语法树的输出。我们定义了一个虚方法printSelf()，然后子类去重写这个方法，打印自己结点的信息，并递归打印其下面的儿子信息。每个结点都以其内存地址的十六进制表示作为自己的ID。为了保证递归不陷入死循环，我们定义了一个集合，当结点被打印后，结点的地址会放入这个集合里去，防止重复打印。我们提供了14个样例供测试，结果都是我们期望的。

最后输出的结果样例如下所示，打印结果默认输出到文件中去：

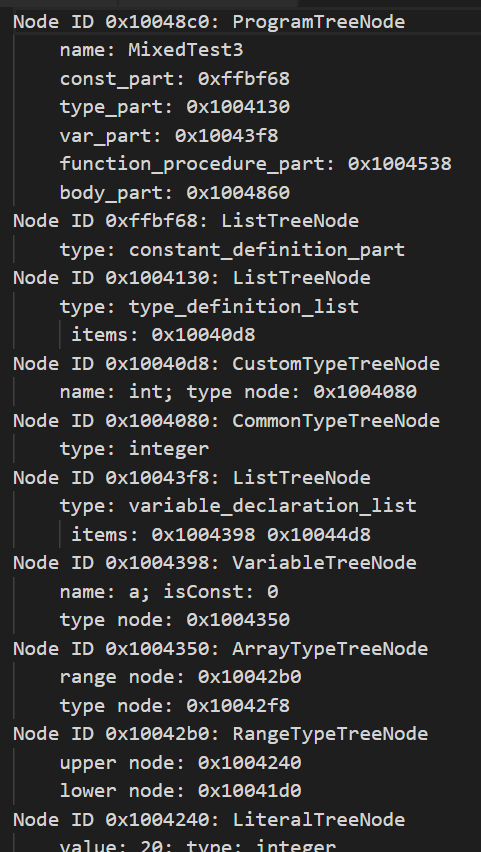


图29，一个AST输出的样例

1. **符号表生成**
2. **类型检查（语义分析）**
3. **代码生成**
4. **测试案例**