[自定义语言编译器前端 1](#_Toc1130916604)

[设计文法 1](#_Toc1134849847)

[文法的定义如下： 1](#_Toc31725801)

[词法分析 2](#_Toc135131057)

[语法分析 3](#_Toc1035201033)

[FIRST集 4](#_Toc898863708)

[状态转移图 5](#_Toc2039428957)

[Action表 9](#_Toc209269048)

[语法制导翻译和中间代码生成 10](#_Toc1682610928)

[对numeric -> INT | REAL | STRING | CHAR的翻译 10](#_Toc160263187)

[对expr -> numeric的翻译 11](#_Toc467901490)

[对expr -> ident = expr的翻译 11](#_Toc261148888)

[对于expr -> expr operation expr的翻译 12](#_Toc694270213)

[对logical\_stmt -> logical\_stmt lop M logical\_stmt的翻译 12](#_Toc2090925933)

[对if\_stmt -> if ( logical\_stmt ) M block N else M block的翻译 14](#_Toc483123010)

[对while\_stmt -> while M ( logical\_stmt ) M block的翻译 15](#_Toc591246629)

[测试程序 15](#_Toc573886938)

[实验总结 17](#_Toc1513275089)

自定义语言编译器前端

* 词法分析
* 语法分析
* 语法制导翻译
* 中间代码生成

设计文法

文法规则代表着一个语言的标准，使用者遵循该规则编写程序代码，编译器根据规则解析用户程序，进而翻译成可执行文件

一些基本的文法有

* 变量的声明定义
* 控制流语句if else for while swtch...
* 函数的声明定义

目前阶段仅仅对变量的声明与定义，if else，while提供了支持。另外不需要显示指出变量类型，在解析的过程中会自动进行类型推导

文法的定义如下：

program -> stmts ;

var\_decls -> var\_decls var\_decl | @ ; 变量的声明和定义 如 a = 10

var\_decl -> ident | ident = expr ;

expr -> ident = expr | numeric | ident | expr operation expr ; 表达式，包括赋值，比较等

operation -> + | - | \* | / ; 二元运算符

comparsion -> > | < | == | >= | <= | != ; 二元比较符

numeric -> INT | REAL | STRING | CHAR ; 类型，用于推导使用，如10会被规约成INT, "hello"会被规约成STRING

block -> { stmts } | { } ; 语句块，用在if, else和while后面

stmts -> stmt | stmts stmt ;

stmt -> var\_decl | expr | if\_stmt | logical\_stmt | while\_stmt ; 语句

M -> @ ;

N -> @ ;

logical\_stmt -> logical\_stmt lop M logical\_stmt | ! logical\_stmt | expr comparsion expr | ( logical\_stmt ) ; 逻辑表达式

lop -> && | || ;

if\_stmt -> if ( logical\_stmt ) M block | if ( logical\_stmt ) M block N else M block ; 条件表达式

while\_stmt -> while M ( logical\_stmt ) M block ; 循环表达式

词法分析

词法分析阶段主要用于将程序代码解析成一个个单词，为后续流程提供token。

比如a = 10会被解析成a, =, 10这三个单词。在解析的同时也需要根据读到的单词判断它的类型，比如

对于关键字（如if, while, else等）保留其符号，token中存储的就是关键字

对于变量（如上述的a），需要将其转换成文法中表示变量的符号（ident），变量名。同时将该变量添加到符号表中

对于分界符（如, ;, (, ), {, }等），同样保留符号

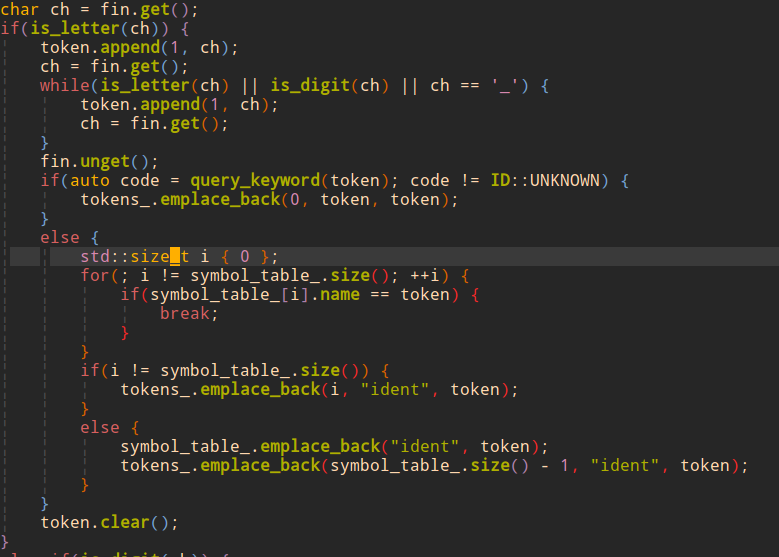
对于常量字面值（如10， 2.5，“hello”等），将其转换成文法中表示常量的符号（INT, DOUBLE, STRING），变量名。

词法分析的程序采用一个字符一个字符读取的方式实现（状态机），每读到一个字符，根据文法中定义的规则判断是否继续读取以构成完成的符号，还是重新读取。比如

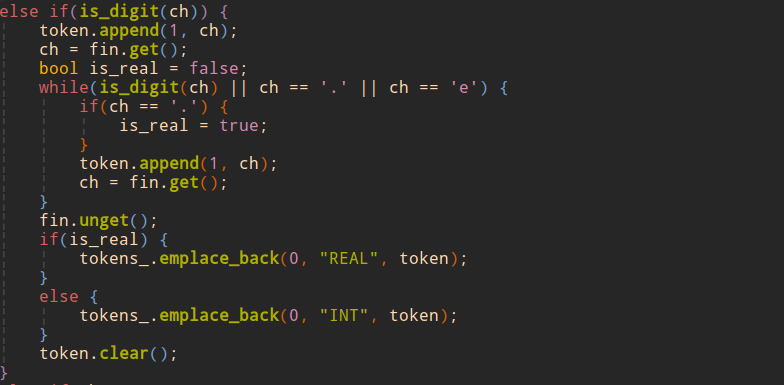
对于变量的定义，方法是字母数字下划线，所以解析时也是按照这种方式解析

对于关键字和变量的解析，首先判断读到的字符是否是字母，如果是，则执行

继续读取字母或数字或下划线，读完之后组装成一个token



对于字面值数字常量，又分为整型数和浮点数，所以需要判断是否存在小数点



对于其它符号的解析方法都相同，不再一一列举

语法分析

语法分析的目标是判断程序代码是否符合语法规则，实现的方法有自顶向下和自底向上两种。对于自顶向下常用的有LL分析，即递归下降分析法

对于自底向上常用的有LR分析，即移进和规约

实验中采用自底向上分析中的LR(1)分析法实现语法分析。

FIRST集

在LR(1)分析法中，首先需要求出每个非终结符的FIRST集，实现方法如下（大写字母代表非终结符，小写字母代表终结符，特殊字符代码由若干终结符或非终结符组成的串）

* 如果A -> Bα，那么需要将FIRST(B)中的所有元素添加到FIRST(A)中
* 如果A -> aβ，那么需要将a添加到FIRST(A)中
* 如果A -> BCθ，且FIRST(B)中包含空字符，那么同时也需要将FIRST(C)中的所有元素添加到FIRST(A)中，一直进行下去直到不再出现空字符

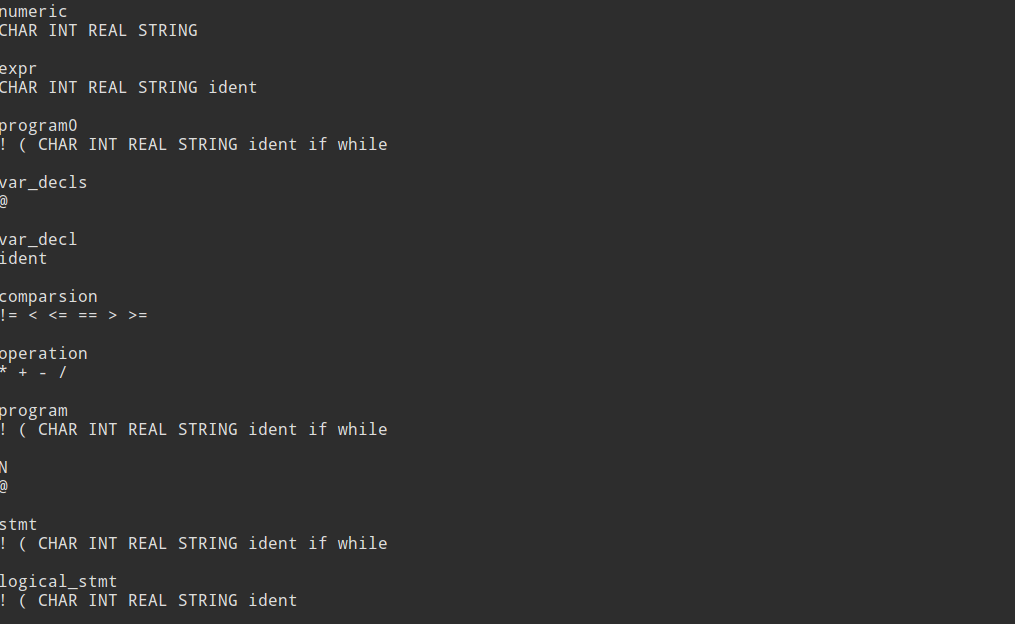
实现方案就是按照上述三步不断进行，直到所有非终结符的FIRST集不再更新

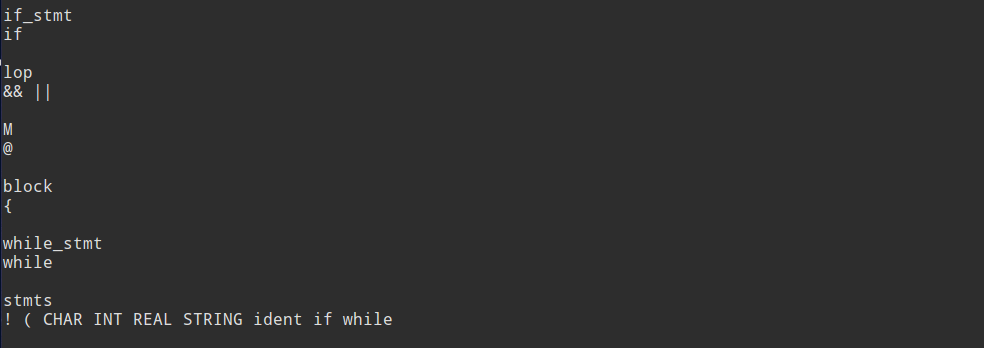






对于开篇给出的文法，每个非终结符的FIRST集如下（其中第一行是非终结符，第二行是它的FIRST集）





状态转移图

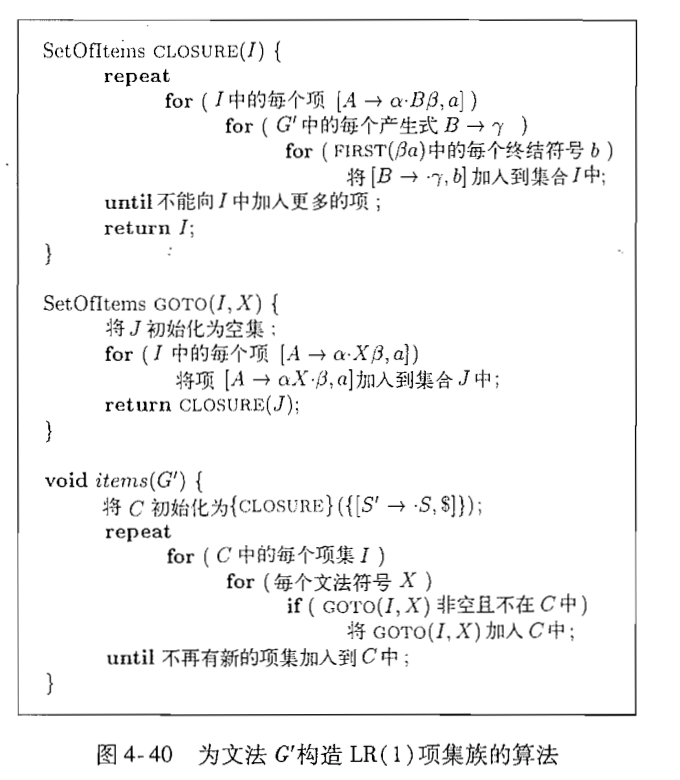
LR(1)分析法的第二步是构造状态转移图，实际上LR将整个分析的过程看做一个个状态，当状态转移图构造完成后，只需要判断当前状态下遇到某个单词时应该转换到的下一个状态即可，这个过程称为移进

当移进到某些状态后无法继续移进时，会进行适当的返回操作，即将之前读到的某几个单词合并成一个新的单词，这个过程称为规约

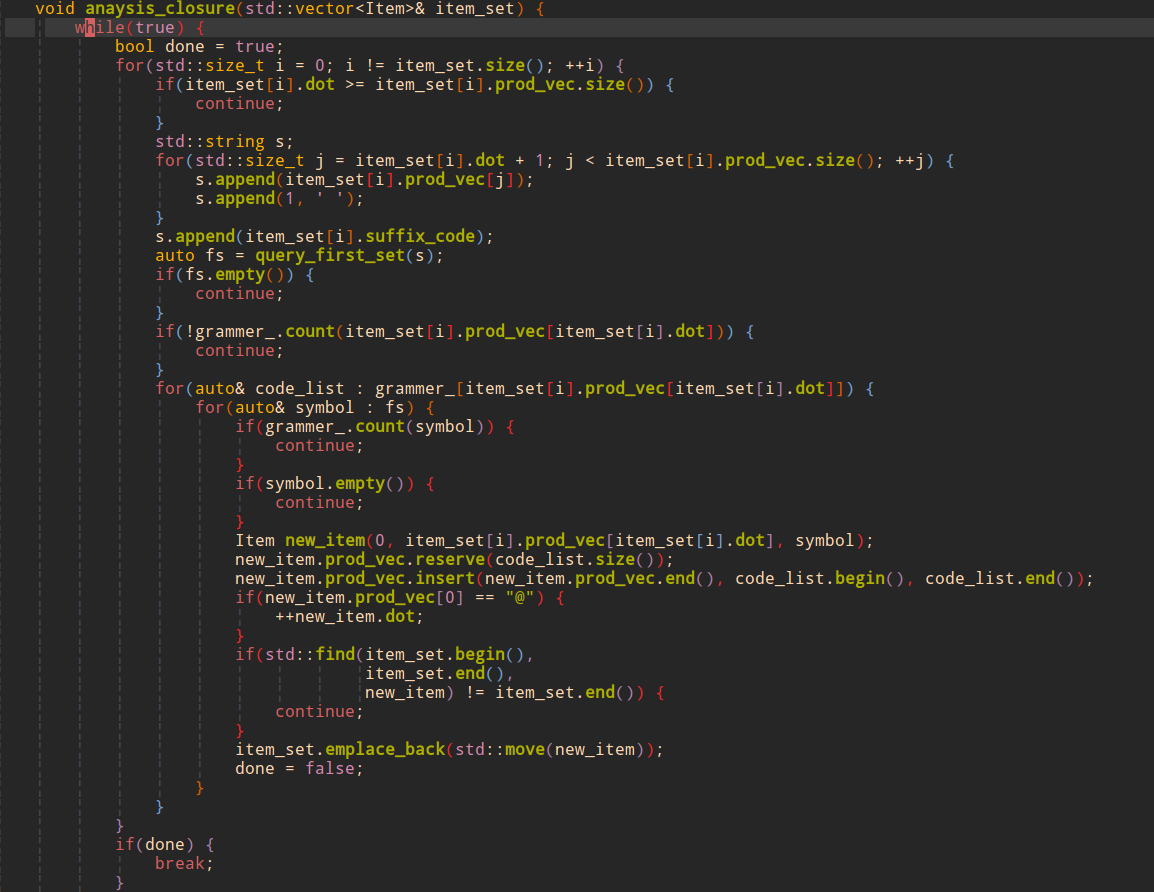
另外为了解决LR中的移进规约冲突，LR(1)采用向前看一个单词的方法解决

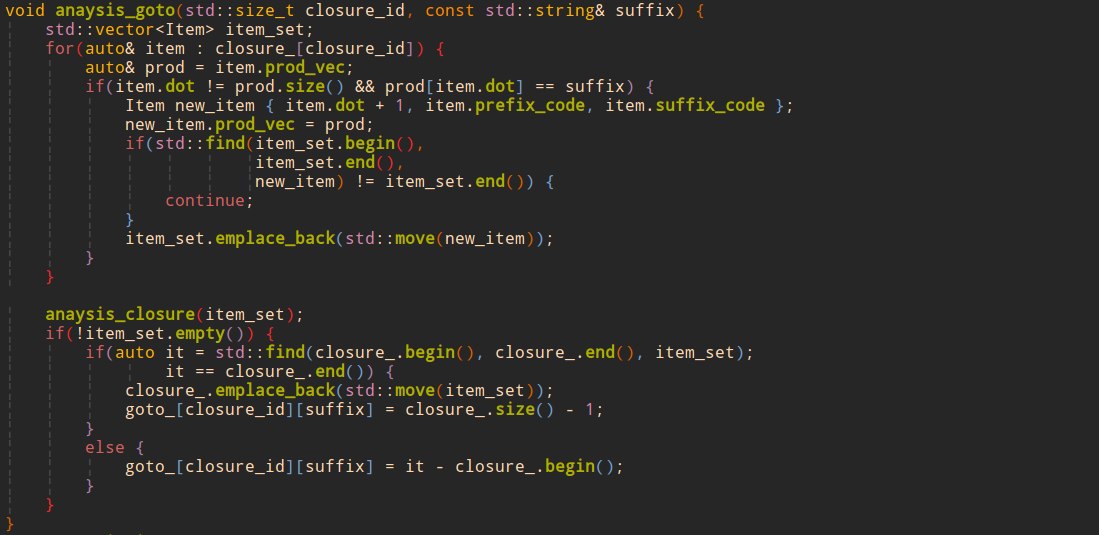
构造状态转移图的步骤如下

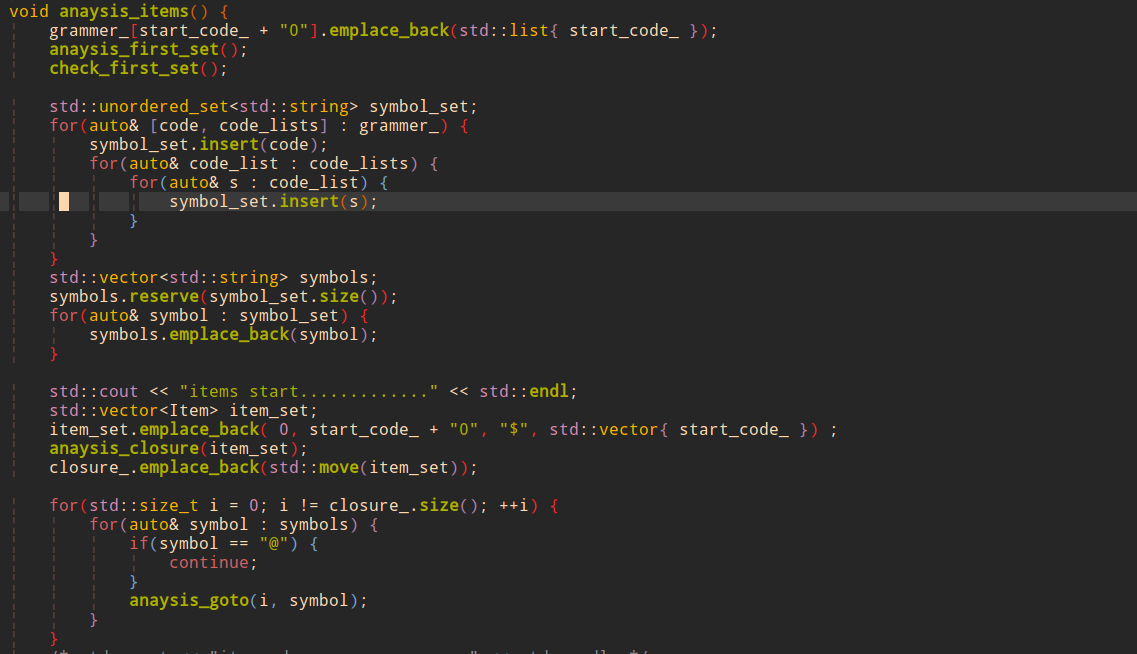
* 构造拓广文法
* 构造一个个状态闭包closure
* 根据闭包之间的生成情况构造goto表
* 根据状态转移图构造action表



分别对应下面三个函数

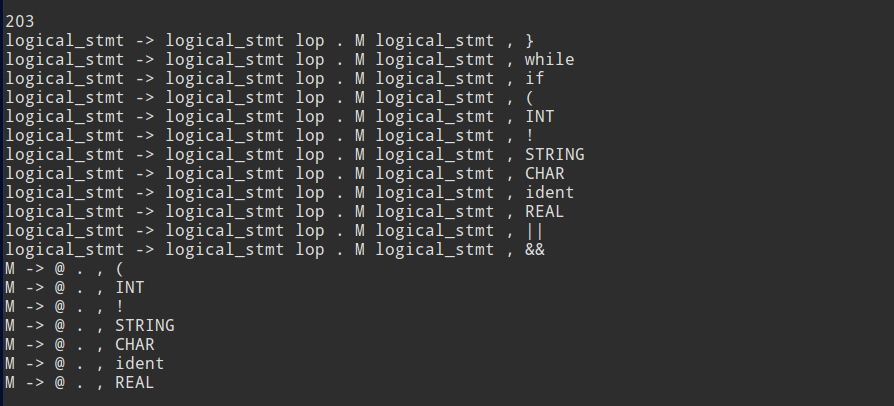


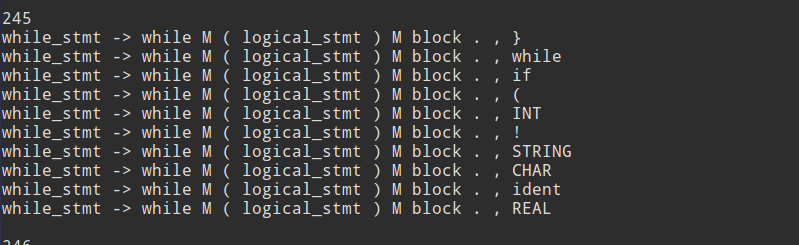


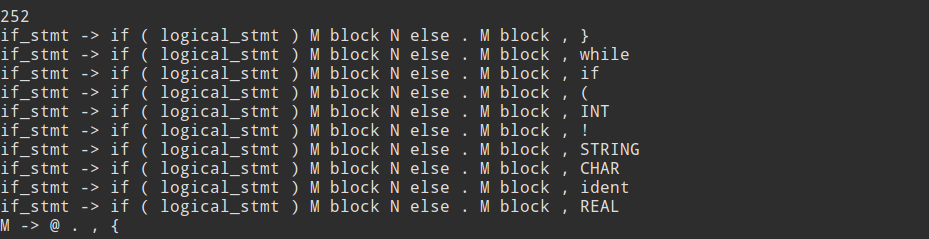


将结果输出，可以看到每个状态下的产生式（仅显示部分状态）



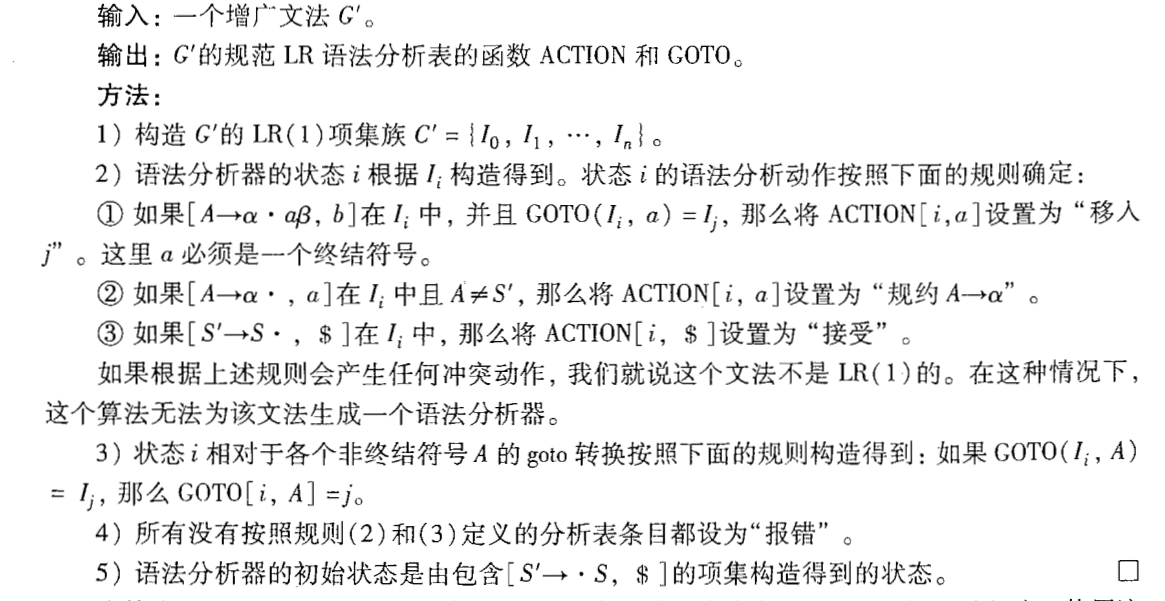




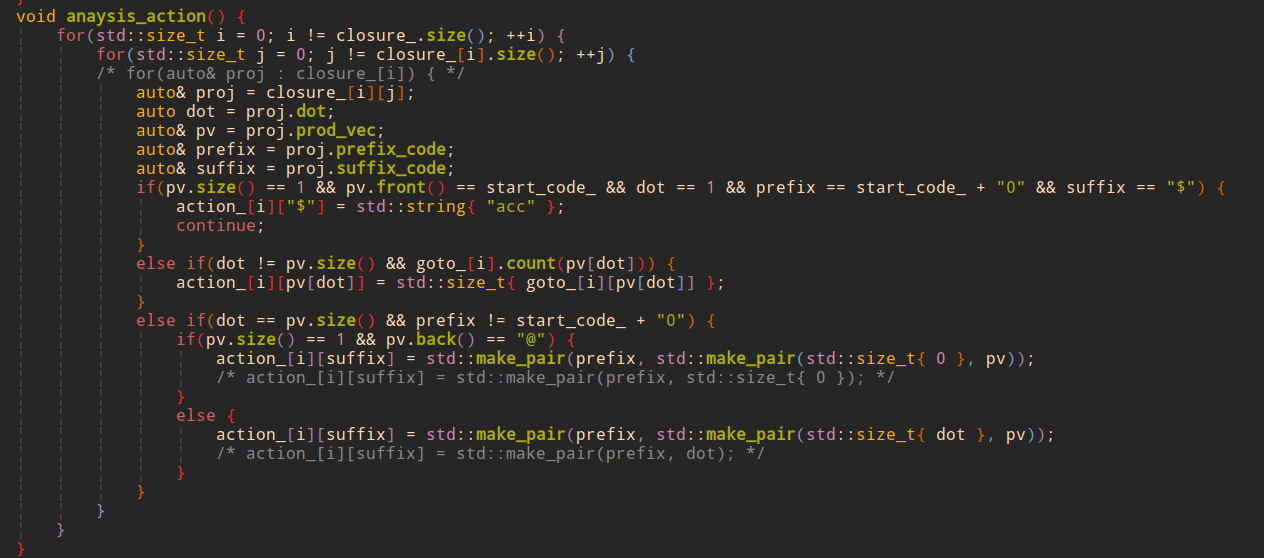


Action表

最后根据这个状态表和代码中生成的转移表构造action表（将goto和action合并）



实际上只需要记录当前状态遇到某个单词后是移进，规约还是acc即可，如果是规约，再记录需要从栈中弹出多少个状态，代码如下



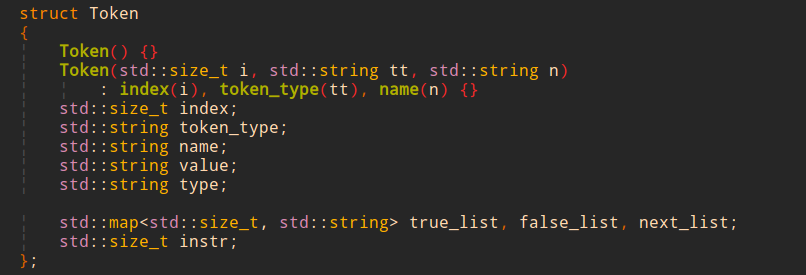
至此整个LR(1)分析器的构造已经完成，接下来的任务就是对源程序进行分析，因为上面已经构造出状态转移表，所以只需要一步步执行即可，如果顺利完成，说明程序没有语法问题，否则出错

语法制导翻译和中间代码生成

实验中采用三地址码的翻译方案将源程序翻译成中间代码，因为LR(1)规约的过程就相当于是对抽象语法树的后序遍历，所以只需要在规约的时候进行适当的翻译

为了实现上的方便，对词法分析中提到的Token进行扩充，包含以下信息

* 符号表中的下标
* 单词类型（文法中的符号）
* 名字（如果是变量，则作为变量名存储）
* 值（保存变量的值，常量的值）
* 类型（类型推导产生的类型，如int, double, string等）
* Instr, True\_list与False\_list（用于对布尔表达式的翻译）



此后在规约的过程中，完全可以使用每个Token记录当前规约的状态，便于后续的翻译

对numeric -> INT | REAL | STRING | CHAR的翻译

当检测到此时的规约是按照上述产生式执行时，进行简单的翻译操作，该操作不产生中间代码，仅仅用于类型推导，将推导出的类型保存在新生成的numeric\_token中。此时就相当于属性栈的栈顶是一个常量，它的类型和值都是已知的（保存在token中），比如存在一个定义语句a = 10，那么对于10这个常量就会进行numeric -> INT规约



对expr -> numeric的翻译

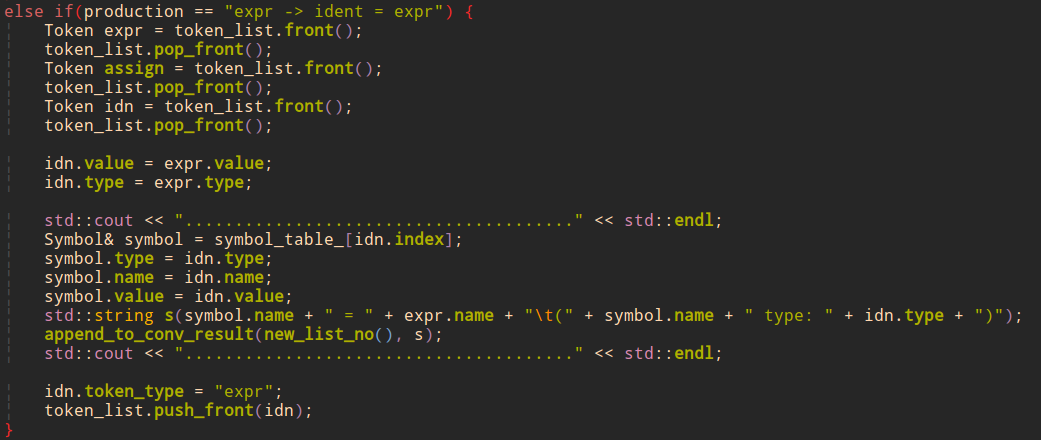
对于a = 10这个定义语句，需要定义一个临时变量t，使其保存numeric的值（这里是10），该操作会产生一个中间代码t1 = 10



对expr -> ident = expr的翻译

仍然是a = 10这条语句，此时右侧的expr是上面翻译expr -> numeric产生的临时变量t1。此时需要从Token栈中弹出三个token分别代表右侧的ident，=以及expr，然后创建新的Token new\_expr代表产生式左侧的expr进行规约，将expr的值保存到ident的符号表中，然后将new\_expr添加到Token栈中

此时产生中间代码a = t1（由此推导出a的类型是t1的类型，也就是int）



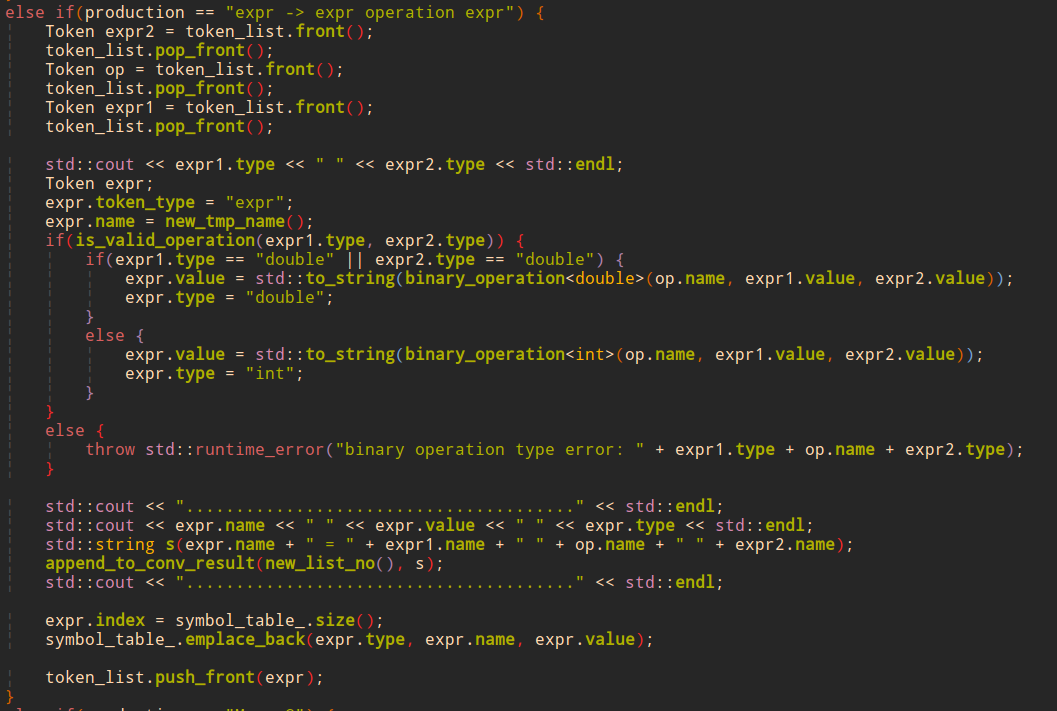
经过上述三步规约，就可以将a = 10翻译完成，结果如下



其中前两行是翻译结果，后两行是符号表中的内容，因为程序在一边翻译一边计算，所以符号表中可以直接填写变量a的值

对于expr -> expr operation expr的翻译

Operation是二元运算符，目前只支持加减乘除并带有类型检测，翻译时，只需要从右侧两个expr中取值，然后进行运算（由于Token中保存有类型，所以可以进行类型检查），最后将结果记录在左侧expr中并添加到Token栈中

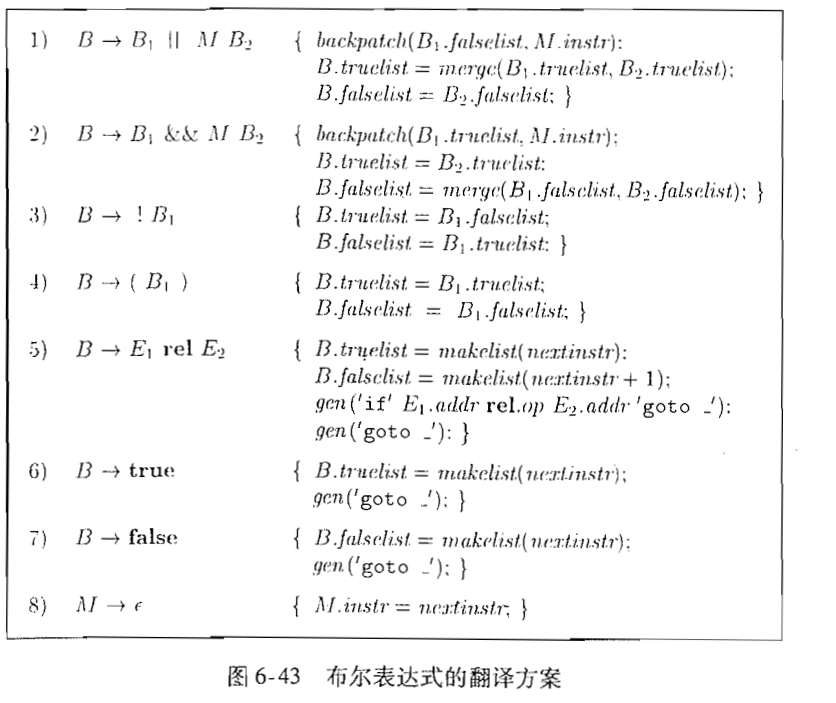


比如a = 10 + 2.5，在进行numeric -> INT | REAL，expr -> numeric，operation -> +的规约操作时，此时的状态就是a = expr + expr，翻译结果为



对logical\_stmt -> logical\_stmt lop M logical\_stmt的翻译

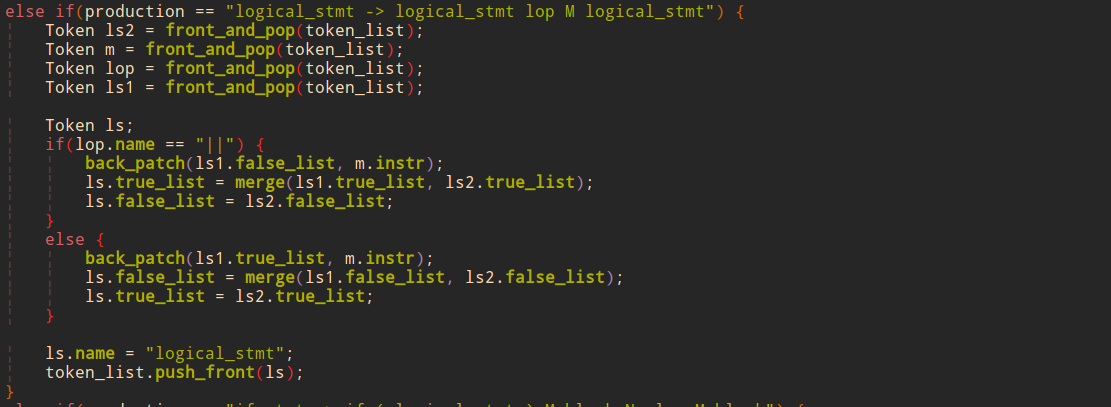
该语句是布尔表达式（逻辑语句），主要是将其翻译成if goto ... goto 的形式，由于在规约时无法确定后面语句的个数，所以goto的目标位置无法确定，实验中采用回填的方法（也就是上面Token中true\_list，false\_list以及instr的用处），翻译方案如下



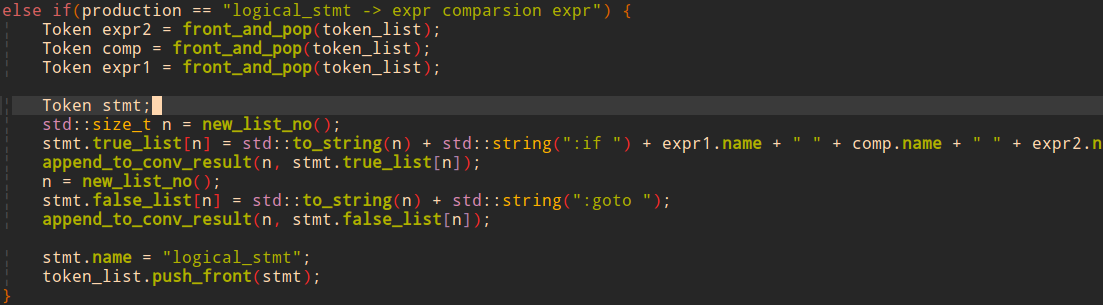
由于文法的限制，只实现了1, 2, 5的翻译，主要方法就是在对对应表达式进行规约时对true\_list和false\_list进行修改，最终输出到中间代码结果中

上述的1,2,5分别对应文法中的两条文法（&&和||合并成comparsion）

Logical\_stmt -> logical\_stmt lop M logical\_stmt



Logical\_stmt -> expr comparsion expr



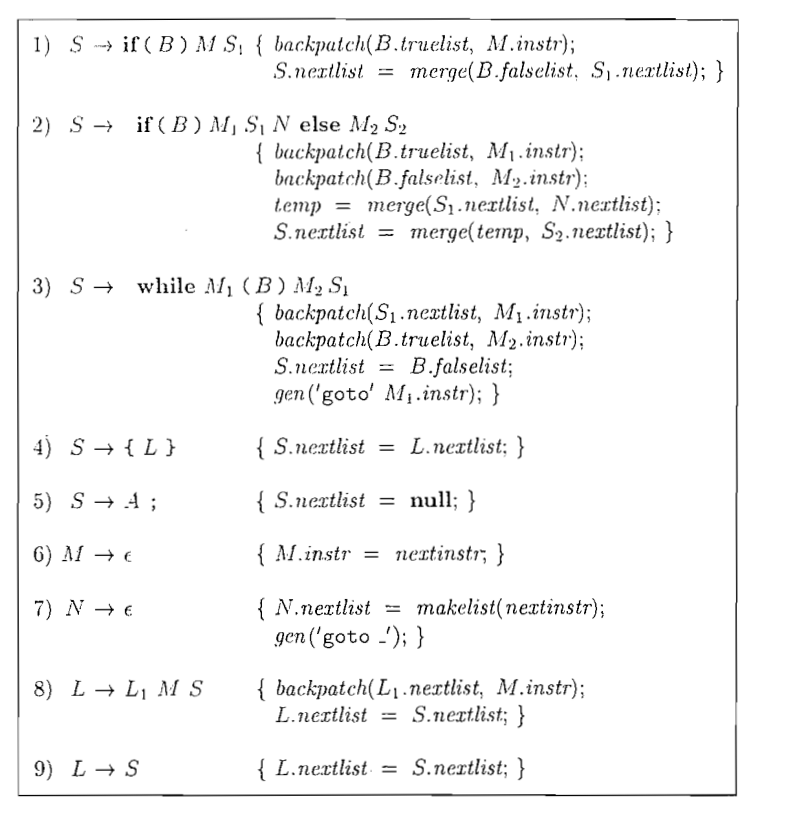
以a=10 b=a a>10&&b<5为例，翻译结果为

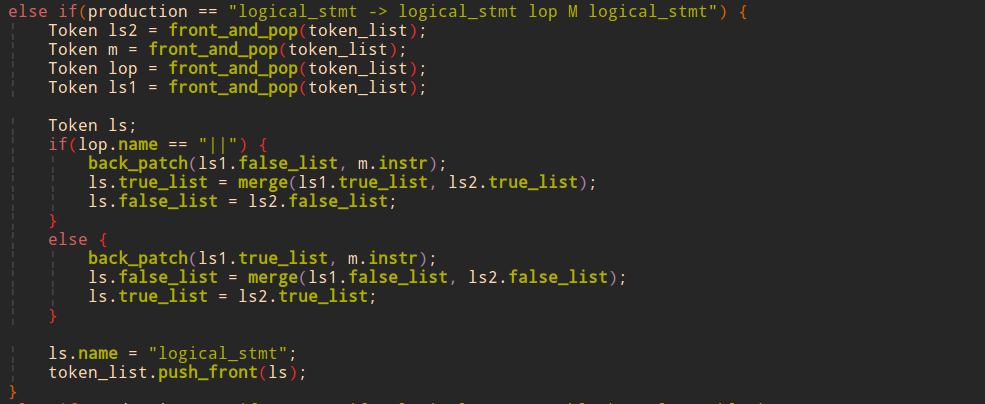


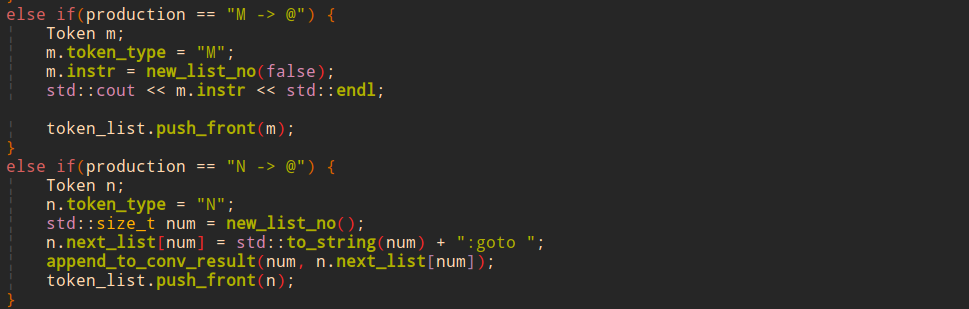
因为逻辑表达式的后面没有任何语句，所以仅仅回填了if t3 > t4 中的goto，事实上，如果将该表达式与if语句结合使用，就会将所有的goto都回填完成

对if\_stmt -> if ( logical\_stmt ) M block N else M block的翻译

当通过该语句进行规约时，说明正在翻译一个完整的if else语句，这里的M和N是用来获取下一个语句标号，方便在规约时进行回填，翻译方案如下（但仅仅需要考虑1, 2, 4, 6, 7）

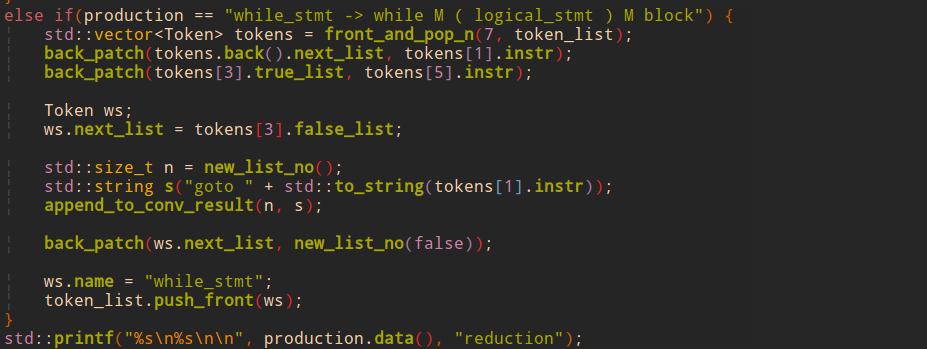






对while\_stmt -> while M ( logical\_stmt ) M block的翻译

翻译方案和上面相同



测试程序

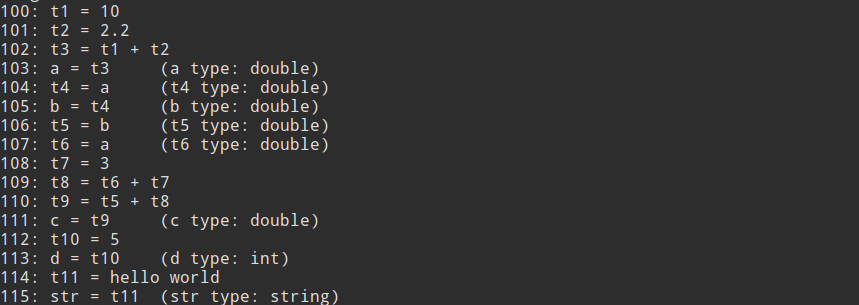
测试程序中包含了变量的定义，赋值，算数表达式，if，while

同时还实现了部分语义错误如类型操作不匹配，不能是用未定义的变量等

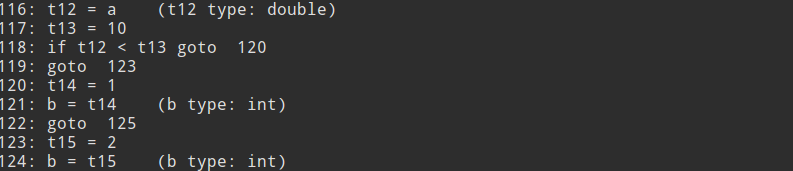


输出结果

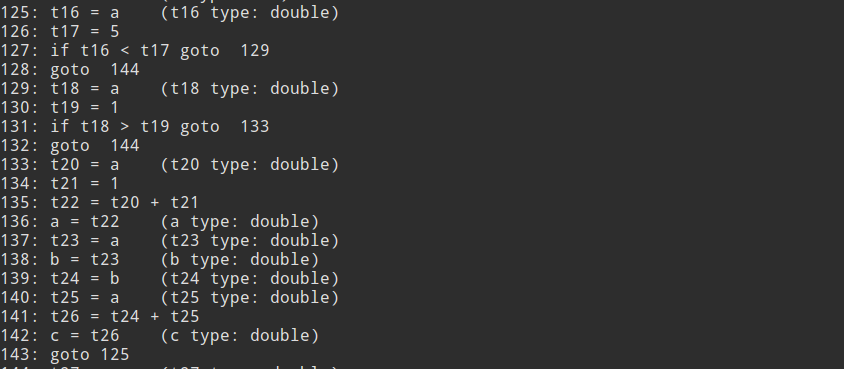
变量定义赋值



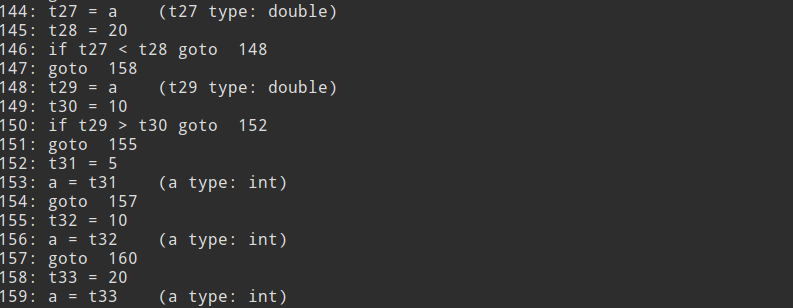
if



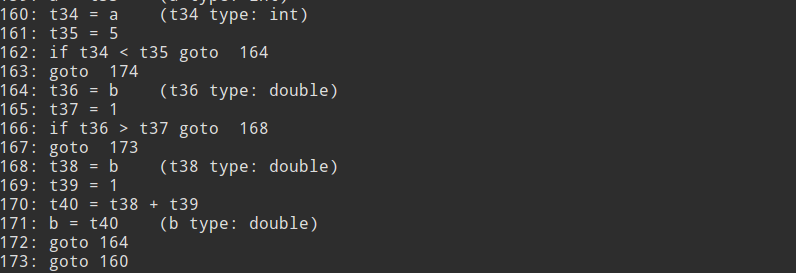
while



嵌套if



嵌套while



实验总结

通过本次实验，实现了词法分析，语法分析，语法制导翻译以及中间代码生成，初步完成了一个简单的解释器，理解了编译器的工作原理