



编译原理大作业

——C0编译器

软件学院

16211077

王启鹏

目录

[一、实验内容 3](#_Toc533518516)

[1.1、实验要求 3](#_Toc533518517)

[1.2、C0文法介绍 3](#_Toc533518518)

[二、编译器设计实现 4](#_Toc533518519)

[2.1、总体结构 4](#_Toc533518520)

[2.2、词法分析 6](#_Toc533518521)

[2.3、语法分析 7](#_Toc533518522)

[2.4、符号表管理 8](#_Toc533518523)

[2.5、错误处理 10](#_Toc533518524)

[2.6、Pcode生成 11](#_Toc533518525)

[2.6.1、修改Pcode 11](#_Toc533518526)

[2.6.2、生成Pcode 12](#_Toc533518527)

[三、拓展部分 13](#_Toc533518528)

[3.1、解释器 13](#_Toc533518529)

[3.2、GUI界面 14](#_Toc533518530)

[3.3、界面使用说明 15](#_Toc533518531)

[3.4、测试结果示例 16](#_Toc533518532)

[四、实验感想 17](#_Toc533518533)

# 一、实验内容

## 1.1、实验要求

1. 以个人为单位开发C0编译器
2. 文件输入：符合C0文法的源程序（自己要有5个测试用例，包含出错的情况，还要用老师提供的测试用例进行测试）
3. 输出：P-Code
4. 错误信息：C0部分错误信息参见教材第316页表14.4
5. P-Code指令集：参见教材第316页表14.5
6. 语法分析部分要求统一使用递归下降子程序法实现。
7. 加分项：友好的UI界面、解释程序、C0文法
8. 上交材料中不但要包括源代码（含注释）和可执行程序，还应有完整文档。

## 1.2、C0文法介绍

1.关键字: const, int, void, if, else, while, main, return, printf, scanf

＜加法运算符＞ ::= +｜－

＜乘法运算符＞ ::= \* ｜／

＜关系运算符＞ ::= ＜｜＜＝｜＞｜＞＝｜!＝｜＝＝

＜字符＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= 0｜＜非零数字＞

＜非零数字＞ ::= 1｜．．．｜9

＜专用符号＞ ::= (｜)｜{｜}｜，｜；｜＝

<字符串> ::= "｛<合法字符> }"

2.

＜程序＞ ::= 〔＜常量说明部分＞〕〔＜变量说明部分＞〕｛＜函数定义部分＞｝＜主函数＞

＜常量说明部分＞ ::= const ＜常量定义＞｛,＜常量定义＞｝;

＜常量定义＞ ::= ＜标识符＞＝＜整数＞

＜整数＞ ::= 〔＋｜－〕＜非零数字＞｛＜数字＞｝｜０

＜标识符＞ ::= ＜字符＞｛＜字符＞｜＜数字＞｝

＜声明头部＞ ::= int　＜标识符＞

＜变量说明部分＞ ::= ＜声明头部＞｛，＜标识符＞｝；

＜函数定义部分＞ ::= （＜声明头部＞｜void ＜标识符＞）＜参数＞＜复合语句＞

＜复合语句＞ ::= ‘{’〔＜常量说明部分＞〕〔＜变量说明部分＞〕＜语句序列＞‘}’

＜参数＞ ::= ‘(’＜参数表＞‘)’

＜参数表＞ ::= int ＜标识符＞｛，int ＜标识符＞} | 空

＜主函数＞ ::= ( void ｜int ) main ＜参数＞＜复合语句＞

＜表达式＞ ::= 〔＋｜－〕＜项＞｛＜加法运算符＞＜项＞｝

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞ ::= ＜标识符＞｜‘（’＜表达式＞‘）’｜＜整数＞｜＜函数调用语句＞

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜‘{’<语句序列>‘}’｜＜函数调用语句＞;｜＜赋值语句＞; | <返回语句>;｜＜读语句＞;｜＜写语句＞;｜＜空＞

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞

＜条件语句＞ ::= if‘（’＜条件＞‘）’＜语句＞〔else＜语句＞〕

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞

＜循环语句＞ ::= while‘（’＜条件＞‘）’＜语句＞

＜函数调用语句＞ ::= ＜标识符＞‘（’＜值参数表＞‘）’

＜值参数表＞ ::= ＜表达式＞｛，＜表达式＞｝｜＜空＞

＜语句序列＞ ::= ＜语句＞｛＜语句＞｝

＜读语句＞ ::= scanf‘(’＜标识符＞‘）’

＜写语句＞ ::= printf‘(’[<字符串>,][＜expression ＞]‘）’

＜返回语句＞ ::= return [ ‘(’＜表达式＞’)’ ]

# 二、编译器设计实现

## 2.1、总体结构

程序采用多遍扫描的方式，首先调用词法分析程序生成token集合，之后调用语法分析程序，利用生成的token集合分析给定代码。采用递归下降子程序实现语法分析程序，遇到错误时进行错误处理，正确时进行符号表管理和Pcode生成。程序流程图如图1所示。

本程序采用java语言实现，最终各个java文件功能如表1所示

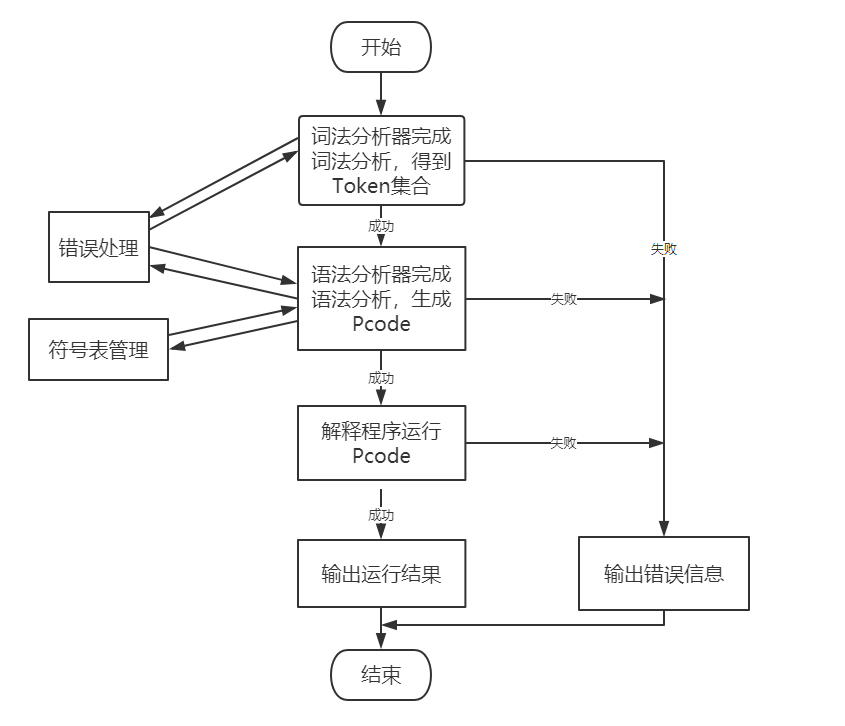


图1 C0编译流程图

表1 各个类说明

|  |  |
| --- | --- |
| 文件名（类） | 功能 |
| Main | 完成UI界面，调用词法分析器、语法分析器、解释器，显示错误信息/程序运行结果 |
| Lexer | 词法分析器，完成词法分析生成所有Token的集合 |
| Parser | 语法分析器，完成语法分析，生成Pcode，包含符号表管理和错误处理 |
| Interpreter | 解释器，运行Pcode |
| Error | 错误处理类，用于生成错误信息 |
| Token | 单个Token，包含Token的所有详细信息（名字，类型、值、行号） |
| Pcode | 单个Pcode，包含F、L、A，修改了部分指令含义，增加属性S方便输出字符串 |
| SymbolType | Token类型，用于一符一类表示 |
| SymbolTableItem | 符号表项，用于管理一条符号表记录 |
| JTextFieldHintListener | 绘制GUI的辅助类，用以显示输入提示 |
| LineNumberHeaderView | 绘制GUI的辅助类，用以显示行号 |

## 2.2、词法分析

C0文法的所有单词可分为表2中的4个大类，在本实验中要求一符一类，所以分界符和关键字的每种使用一个名字进行标记，记录在SymbolTable中

词法分析程序的逻辑是：每次读取一个字符，跳过所有空白字符，对于第一个非空白字符，判断他的类型，进入不同的if分支进行判断。

对于识别出来的标识符，首先和关键字进行比价，之后进行分类；

对于识别完的数字，需要判断后面紧挨着的是不是下划线或者字母，以保证标识符不以数字开头；

对于识别出来的 + 或者 - ，根据前面的Token和后面是否为数字判断是加减运算符还是正负号，如果前面是运算符、关系负、逗号、左小括号说明是正负号，否则不是；

对于识别出来的 / ，判断后面是不是注释，跳过所有的注释，注释由/\*\*/包含

最终程序的结构如图2所示。

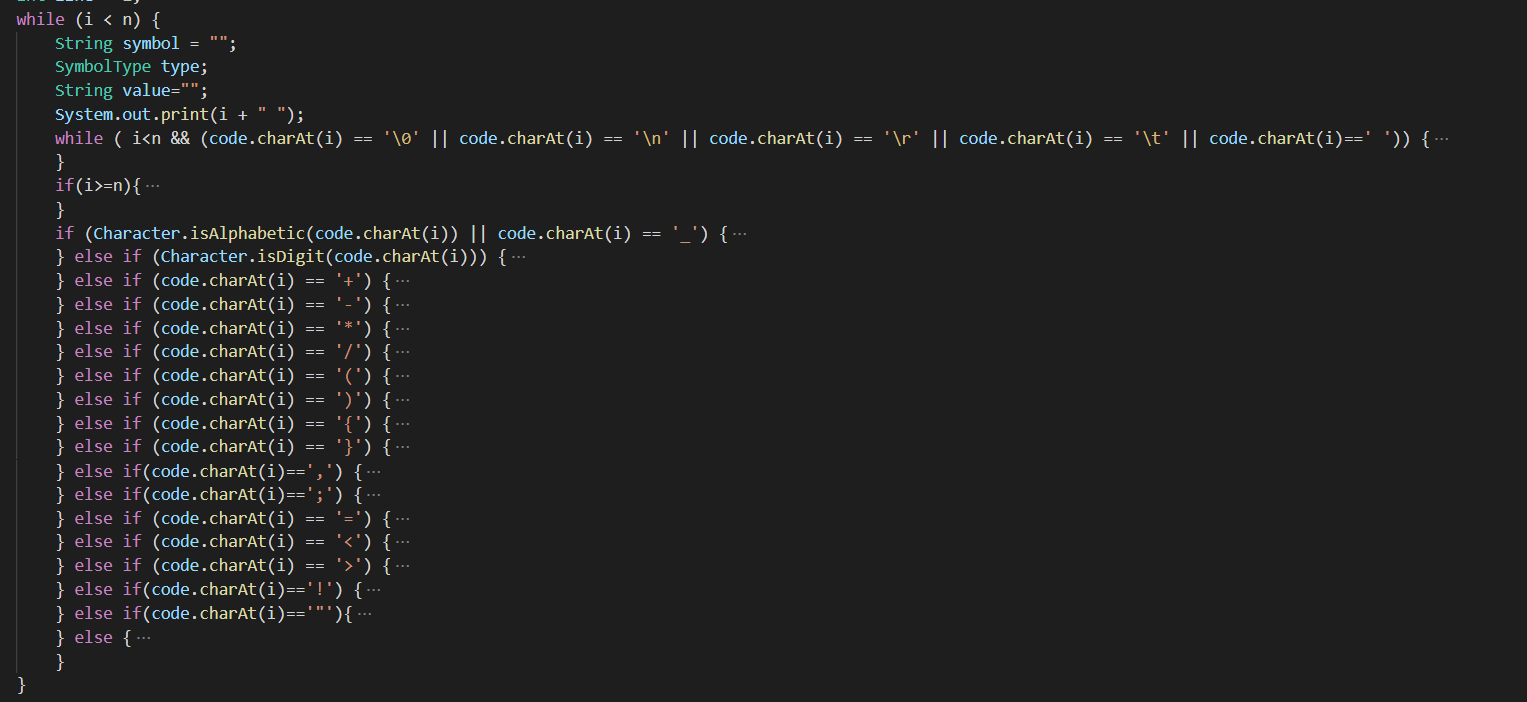


图2 词法分析程序结构

表2 C0的Token类型

|  |  |
| --- | --- |
| 关键字 | const, int, void, if, else, while, main, return, printf, scanf |
| 分界符 | + - \* / < <= > >= == != ( ) { } ， ； = |
| 常量 | 整数 |
| 标识符 | 字母下划线开头，后面跟字母数字下划线构成的串 |
| 字符串 | 双引号包含 |

## 2.3、语法分析

语法分析采用递归下降子程序实现，为每个Vn编写一个子程序，根据当前读取的符号进行相应子程序调用，对于递归子程序方法，对文法有两个要求：

* 1. 文法不能左递归
  2. 不能回溯，即同一个Vn候选式的First集合不能相交

对于给定的扩充BNF表示的C0文法满足无左递归，对于但是存在First相交的问题。

First相交问题一：文法中＜变量说明部分＞和＜函数定义部分＞都是以<声明头部>开始的，而且二者都是可选的，因此需要做判断。程序中没有提取公因子，而是选择预判，具体做法为：当读取到int的时候，就提前看看int后面第二个token，如果是（ 说明是函数定义，否则是变量声明，这点在多遍扫描程序中实现很简单。

First相交问题二：语句中的函数调用语句和赋值语句的first都是<标识符>，但是这个问题可以加入符号表管理之后就不存在，只需要判断对应符号表项中的type属性即可。

同时C0还存在典型的“悬挂else”问题，是个典型的造成文法二义性的问题，对于这个问题，解决方法是语义上规定else跟就近的if。

完成上述分析之后，设计各个模块。将<声明头部>带入到需要的其他模块中，将＜函数定义部分＞分解为＜int函数定义部分＞和＜void函数定义部分＞，将＜主函数＞融合到两个<函数定义部分>中，最终C0语法分析器各个递归下降子函数功能如下表3所示

表3 递归下降子函数表

|  |  |
| --- | --- |
| 函数 | 功能 |
| program | ＜程序＞ |
| constDeclaration | ＜常量说明部分＞ |
| constDefine | ＜常量定义＞ |
| variateDeclaration | ＜变量说明部分＞ |
| intFunctionDeclaration | ＜int函数定义部分＞ |
| voidFunctionDeclaration | ＜void函数定义部分＞ |
| compoundStatement | ＜复合语句＞ |
| parameter | ＜参数＞ |
| expression | ＜表达式＞ |
| term | ＜项＞ |
| factor | ＜因子＞ |
| statement | ＜语句＞ |
| assignStatement | ＜赋值语句＞ |
| ifStatement | ＜条件语句＞ |
| condition | ＜条件＞ |
| whileStatement | ＜循环语句＞ |
| callStatement | ＜函数调用语句＞ |
| valueParameter | ＜值参数表＞ |
| statementList | ＜语句序列＞ |
| scanfStatement | ＜读语句＞ |
| printfStatement | ＜写语句＞ |
| returnStatement | ＜返回语句＞ |

说明：在处理printf语句的时候，由于给定文法本身比较奇怪不符合C语言的习惯，于是编写了两个函数printfStatement和printfStatementA，分别是使用了原始文法的处理方式和使用了C语言的处理方式，在展示的时候使用的是printfStatement，如果需要的时候在statement函数里面更换相应的调用函数即可，具体如下图4所示。



图3 更换相应的printf子模块

## 2.4、符号表管理

符号表管理模块融合到语法分析部分一起完成，符号表项定义如表4所示。

符号表的管理设计三个主要操作：

1. 检查表：判断符号表中是否存在对应的符号表项，以判断是否可以插入
2. 查找表：查找对应表项，返回结果
3. 插入表：项符号表中插入对应项

表4 符号表项定义

|  |  |
| --- | --- |
| 属性 | 含义 |
| name | 名字：常量名、变量名or过程名 |
| value | 值 |
| type | 1：常量（只有int）、2：变量（只有int）、3：int函数、4：void函数、5参数、6字符串 |
| level | 1表示全局层；2表示函数内层 |
| address | 地址：常量、变量地址为相对于AR基址的偏移量；函数地址为第一条pcode所在位置 |
| parameterNumber | 参数个数，函数专用 |

这里将检查是否能插入表和查找表项分开，原因是标识符存在作用域，而局部的标识符会覆盖全局的标识符。经过尝试后发现，图3所示的代码可以正常编译运行，结合之前的知识不难得出结论：局部标识符（常量、变量）不能和局部标识符重名，但是可以和全局标识符重名，包括函数名；全局标识符只需要不互相重名即可。

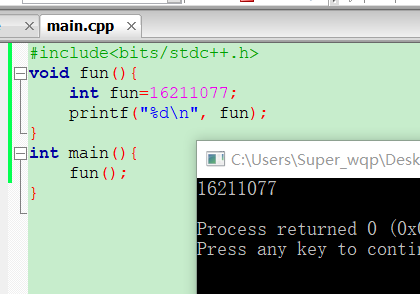


图4标识符作用域测试

注意到C0的标识符作用域只有两层，即全局层和函数内层。C0不能函数套函数，语句里面不能再定义变量，属于非分程序，因此采用书5.3.1节的分离符号表的策略，即设置全局符号表和局部符号表，全局符号表存储全局常量、变量、函数名，局部符号表存储函数参数、局部常量、变量，建立函数名到符号表的映射。

最终形成图4所示的符号表管理图。实现上述的三种操作：

1. 检查表：根据当前所在层次判断，如果是全局层，搜索管全局表，如果是局部层，就寻找最后一个全局表项对应的局部表，搜索对应局部表。根据表中是否存在返回boolean类型
2. 搜索表：根据当前所在层次判断，如果是全局层，搜索管全局表，如果是局部层，就先搜索对应局部表项，如果没有再搜索全局表。如果找到返回对应表项，失败返回null
3. 插入表：首先检查表判断能否插入，之后插入到对应表中

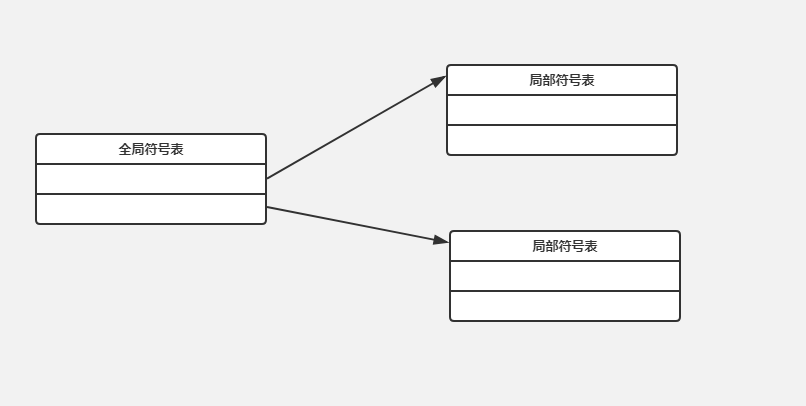


图5 符号表管理图

## 2.5、错误处理

由于C0和PL0文法的差异性，书上的一些错误处理不能直接使用，所以结合了书上的错误信息和C0本身的特性，总结出来如表5所示的错误信息。

错误处理模块中尽量准确的定位了各个符号的所在行号，同时尽量保证了程序鲁棒性，即两个语句中间出现了= <这种符号在处理时候应当能够过滤并提示。对于每条语句，如果发现括号、分界符等不匹配时，除了处理错误信息之外还应该继续处理，即跳转到下一个可能的开始部分的first集合中或者结束后的follow集合中。

比如printf中缺少了（，那么就应该跳转到一条语句的开始或者分号或者语句序列的follow集，即右花括号，之后继续分析处理。再比如return后面少了分号，那么跳转就应当是右花括号、函数开始部分（int、void）

错误处理的难点在于遇到错误时如何选择跳转的终点，在本程序中采用的是跳转到后面可能跟着的集合中。

表5 错误信息

|  |  |
| --- | --- |
| 编号 | 信息 |
| 0 | 标识符不能以数字开头 |
| 1 | 数字不能前导0 |
| 2 | 缺少 \*/ |
| 3 | '!'后面缺少‘=’ |
| 4 | \后面缺少字符 |
| 5 | 缺少 " |
| 6 | 非法字符（提示字符是什么） |
| 7 | 非法单词/符号 |
| 8 | 此处应该是 const |
| 9 | 缺少分号； |
| 10 | 此处应该为标识符 |
| 11 | 此处应该为 = |
| 12 | 此处应该为整数 |
| 13 | 此处应为 int |
| 14 | 缺少 main 函数 |
| 15 | 非法函数头部 |
| 16 | 此处应为（ |
| 17 | 此处应为 ） |
| 18 | 此处应为 { |
| 19 | 此处应为 } |
| 20 | 重复定义main函数 |
| 21 | 此处应为void |
| 22 | 缺少函数名 |
| 23 | int类型函数缺少返回值 |
| 24 | void类型函数不能有返回值 |
| 25 | printf缺少逗号'，' |
| 26 | 表达式错误 |
| 27 | 函数应定义在main之前 |
| 28 | 标识符重复定义（提示标识符是什么） |
| 29 | 常量不能作为语句的开始（提示常量是什么） |
| 30 | 标识符没有定义（提示标识符是什么） |
| 31 | 参数不匹配 |
| 32 | void函数不能参与运算 |
| 33 | scanf不能作用于常量/函数 |
| 34 | 输入不合法 |
| 35 | 计算错误：除数为0 |
| 36 | 计算错误：运行栈溢出 |

## 2.6、Pcode生成

### 2.6.1、修改Pcode

由于C0包含了函数调用，而这涉及到参数的传递和返回值的保存问题，同时printf可以书写字符串，所以按照书上的Pcode指令集不能满足上述的两个需求。由于C0的总体层次只有两层，因此层次差只能是1或者0，所以部分指令的L参数可以自定义，确保满足C0的需求。现对Pcode做如表6改动。

表6 修改的Pcode指令含义

|  |  |  |  |
| --- | --- | --- | --- |
| OPR | 0 | 0 | Void类型函数返回 |
| 1 | 0 | Int类型函数返回，将栈顶返回值保存在调用者栈顶 |
| LOD | 0 | a | 将全局变量相对偏移量a的数据保存在栈顶 |
| 1 | a | 将局部变量相对偏移量a的数据保存在栈顶 |
| 2 | a | 将相对调用者栈底偏移a的数据把存在栈顶 |
| CALA | 0 | a | 为要调用的函数开辟新的栈帧，需要a的空间以保存变量、常量、返回信息，但是不改变PC值 |
| CALB | 0 | a | 改变PC跳转到第a指令开始执行 |
| WRTI | 0 | 0 | 将栈顶整数内容打印 |
| WRTS | 0 | 0 | 将保存的字符串信息打印 |

修改Pcode的思路如下：

原始Pcode不包含函数参数的传递，而传参需要为新函数开辟空间，此时栈帧已经改变，但是仍然要运行＜值参数表＞以将参数压栈，也就是说此时仍在调用者的指令内运行。如果只利用一个CAL，那么如果放在压入参数之前，PC就必须跳转，那样参数就无法入栈；如果放在压入参数之后，那么新栈的栈底就找不到了，因此将CAL分解为CALA和CALB两个操作。

Pcode中不包含返回值，但是C0返回值可能参与运算，也就是返回值需要在栈顶。OPR的L始终是0，所以可以修改L改变OPR在a=0的时候的含义，分别作为int和void函数返回的标志

C0文法需要输出字符串和数字，所以将WRT分解为WRTI和WRTS分别表示write int和write string。

在上述函数调用过程中，当执行了CALA指令之后，栈帧已经发生了变化，如果仍然用相对当前栈帧的偏移量表示参数的地址就无法压入正确的参数，因此对LOD指令的L进行了修改，当L=2时，表示相对调用者的栈底偏移，在解释Pcode的时候只需要查找对应的栈底值寻找原栈底即可。

### 2.6.2、生成Pcode

修改了部分Pcode指令之后，即可生成Pcode指令。在语法分析的时候，当分析到一条语句的时候即可生成相应的Pcode。

下面介绍该项目中生成Pcode的两个核心点：生成逻辑和拉链回填

生成逻辑：由于C0需要从main的第一条指令开始执行，因此需要进行跳转，但是跳转到main函数需要为main开辟变量空间，之后跳转，类似于函数调用。而代码的第一条INT指令是为了为全局变量或者常量开辟地址空间，通过增加栈帧的方式实现。由于Pcode解释执行结束的标志是PC=0（参考书P380），所以main返回之后需要把PC置零，因此可以假象有一个“上帝函数”调用main函数，将该函数的指令放在最前面，就可以通过调用main找到main函数了，具体如下：

void god(){

//全局常量、变量

main();

return;

}

只需要开辟全局常量、变量空间，并god函数的保存的返回信息设置为0，那么返回后PC就自然是0了。这就是前四条指令总是INT、CALA、CALB、OPR的原因了

为什么不直接在main的栈帧中设置返回信息为0呢？原因则是调用main函数是通过和其他函数一样的两个CAL指令完成的，根本不知道是不是调用的main函数。如果为了main特地修改Pcode就显得不划算了。

拉链回填：用于＜条件语句＞和＜循环语句＞，用于指令的跳转。由于顺序扫描不知道接下来你跳转的位置，因此需要先记录，之后回填。两种语句的具体操作如下

＜条件语句＞🡪if‘（’＜条件＞‘）’@(JPC lab1)＜语句＞lab1〔else @(JMP lab2)＜语句＞lab2〕

＜循环语句＞ 🡪 while lab1 ‘（’＜条件＞‘）’ @(JPC lab2)＜语句＞@(JMP lab1) lab2

其中红色表示动作（一条Pcode），蓝色代表跳转的位置。因此拉链回填实现就比较简单了，只需要在所有标签处记录位置，之后在执行到生成Pcode的时候填写位置；或者生成Pcode时先填0，之后找到标签之后返回修改即可。

# 三、拓展部分

## 3.1、解释器

了解了Pcode每个指令的含义之后，可以很容易的实现每条指令。下面说明符号表管理中的具体细节。由于涉及解释器，所以最关键的是标识符的地址，这里定义为相对于栈底的偏移量，从3开始，偏移为0、1、2的地方分别保存原调用者栈底、返回的PC值和调用的函数的栈顶（int函数是返回值，void函数最后会忽略），因此对于当前层常量和变量的地址顺序累加即可。对于函数的地址，设置为函数的第一条指令的索引。

对于没有修改的Pcode指令，实现参考课本P378的interpreter，下面给出修改后的Pcode的解释程序的实现

表7 部分Pcode的解释执行

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OPR | 0 | 0 | Void；类型函数返回 | t=b; b=stack[t]; p=stack[t+1]; |
| 1 | 0 | Int类型函数返回，将栈顶返回值保存在调用者栈顶 | stack[b+2]=stack[t-1];  t=b;  b=stack[t];  p=stack[t+1];  stack[t]=stack[t+2];  t++; |
| LOD | 0 | a | 将全局变量相对偏移量a的数据保存在栈顶 | stack[t++]=stack[b\*pcode.L + pcode.A]; |
| 1 | a | 将局部变量相对偏移量a的数据保存在栈顶 | stack[t++]=stack[b\*pcode.L + pcode.A]; |
| 2 | a | 将相对调用者栈底偏移a的数据把存在栈顶 | stack[t++]=stack[stack[b] + pcode.A]; |
| CALA | 0 | a | 为要调用的函数开辟新的栈帧，需要a的空间以保存变量、常量、返回信息，但是不改变PC值 | stack[t]=b; b=t; t+=pcode.A; |
| CALB | 0 | a | 改变PC跳转到第a指令开始执行 | stack[b+1]=p; p=pcode.A; |
| WRTI | 0 | 0 | 将栈顶整数内容打印 | output.add(stack[t-1]+"\n"); |
| WRTS | 0 | 0 | 将保存的字符串信息打印 | output.add(escape(pcode.S)); |

：

说明：WRTI和WRTS中的output是定义的输出列表，用于存储输出信息返回Main刷新UI；pcode为当前指令，p指向即将执行的下一跳指令，b指向当前栈栈底，相对于b偏移0存储原栈底，偏移1存储返回指令地址，偏移2存储返回值（就是被调用函数的栈顶），t存储栈顶指针，指向下一个可以存放元素的位置；pcode.S存储需要打印的字符串，为WRTS专用。

运行栈定义的大小是1000000，如果溢出会有错误提示。运行出现除以0错误进行错误提示。

## 3.2、GUI界面

GUI界面沿用了之前词法分析程序的界面设计，使用java自带的图形界面库，添加了Pcode指令集的显示，添加了输入文本框，具体如图5所示

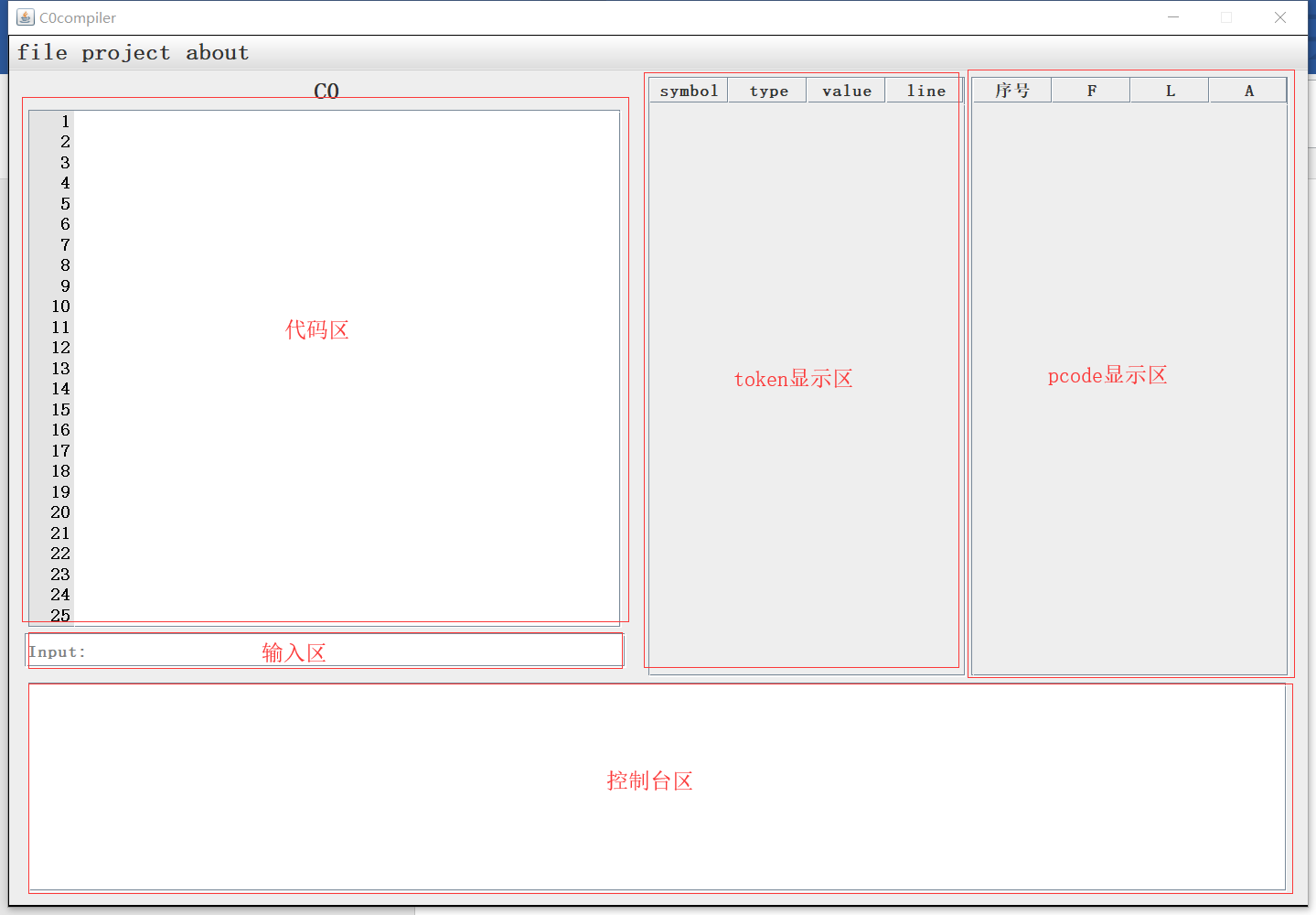


图6 GUI界面

## 3.3、界面使用说明

界面包含四大部分，分别是菜单栏，输入部分（包含代码和输入），表格显示（包含token和pcode）以及控制台信息提示部分。

用户可以选择在代码区自行输入或者选择file的open打开文件。File save用以保存文件，close用于清屏，exit用于退出程序。每个模块都有是否保存文件的相关提示信息。

用户点击project的compile选项编译代码，生成token和pcode列表，显示在表格区域。多遍扫描程序，如果某一个子模块出现错误，就不在进行下一个处理，**比如词法分析出错就不会进行语法分析处理，只有修改了词法分析的错误才可以。**点击run可以执行pcode，执行前需要预先进行输入，否则会报错。只有词法分析成功了才显示token，并进行语法分析；语法分析成功时显示pcode，否则不予以显示。

显示字符串时对于\n\t\r三个转义符做了判断，可以特殊处理，比如printf(“hello\nworld”,)会分行显示hello world（注意字符串后面需要接逗号）。

## 3.4、测试结果示例

下面给出正常显示样例和失败样例的测试结果。

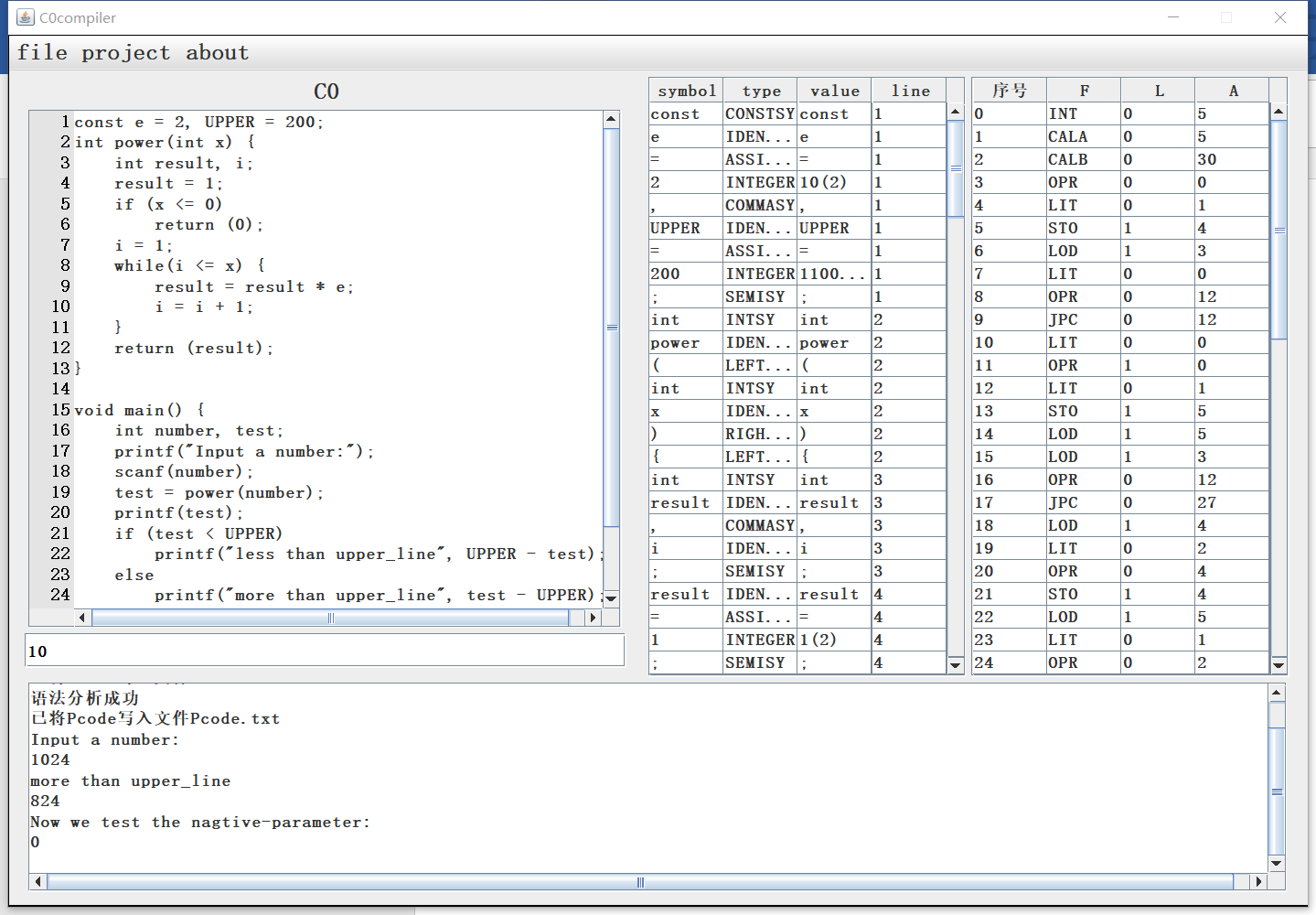


图7 正确测试用例

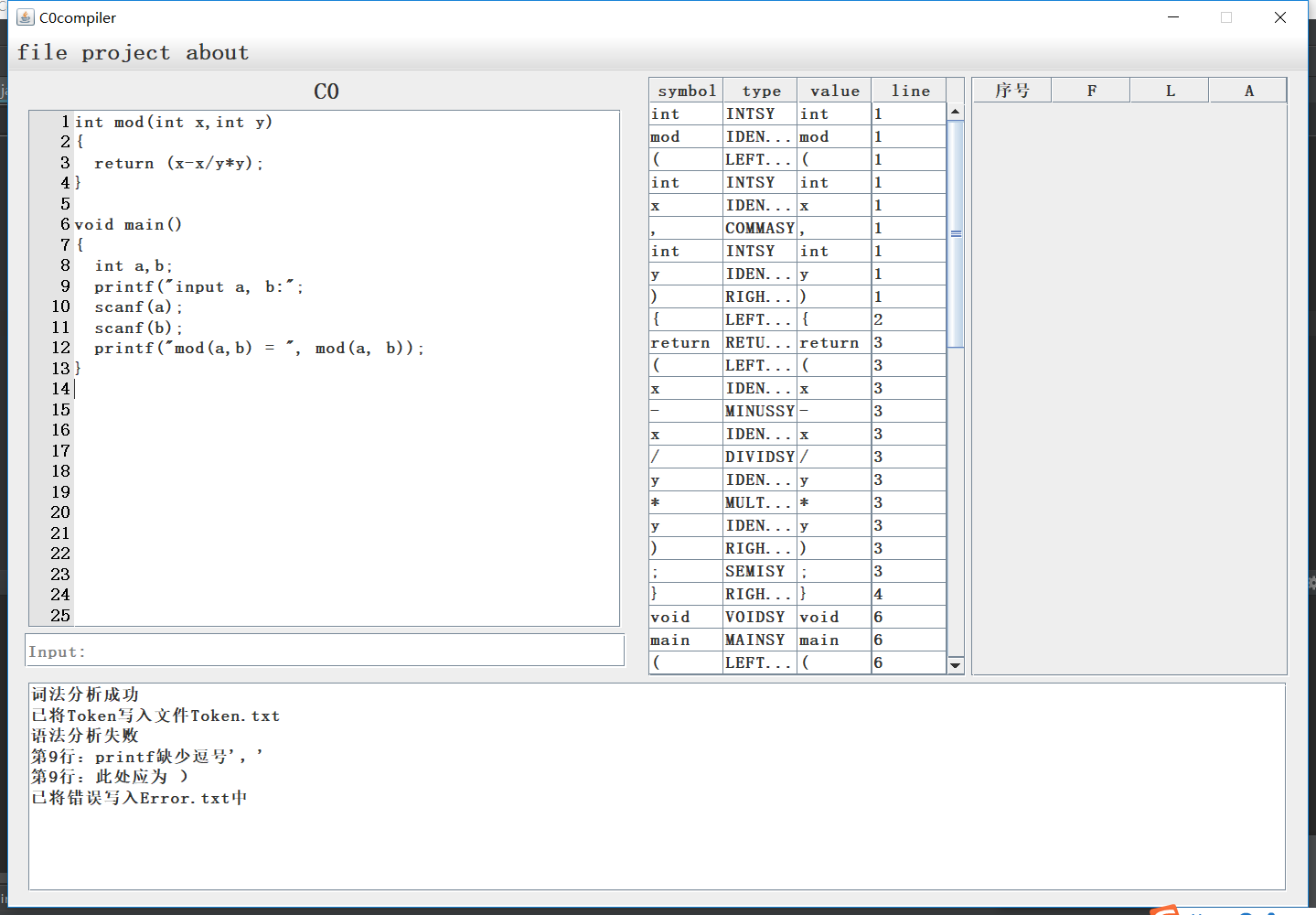


图8错误测试用例

# 四、实验感想

首先是两种文法的比较，C0文法的总体结构相对于PL/0要复杂，但是如果仅仅是词法和语法分析而言差距并不算是特别明显。主要的难点在于C0文法参数的传递以及返回值的处理，原始的Pcode不能处理这两种情况，所以必须在原始的基础上做一定的修改，而这个修改的过程是模仿了X86的函数调用中栈帧的处理过程。

C0中相对容易的一点是符号表管理，由于C0变量的作用域只有全局和局部两种，所以对符号表的管理非常简单，这也就暗示了Pcode中的L参数只能是0和1，这使得我们可以在部分指令中修改L的含义来修改原始的Pcode操作。

Pcode的生成是整个编译器最难的地方，首先是需要完全理解每条指令的含义以及“栈式计算机”的工作模式，如果不能理解，那么就无法生成Pcode，也就自然无法写出解释器了。

而理解了Pcode并按照自己的逻辑生成了正确的Pcode之后解释器就非常好写了。只需要按着每条指令的内容完成即可，而大部分内容在树上都有详细的说明，只需要注意一下自己修改的部分即可。

整个编译器第二个难点是错误处理，语法分析的时候识别到错误非常简单，但是难点是遇到错误之后该怎么办，应该是能够继续处理后面的部分，这里就涉及到了程序的跳转，即跳转到下一个合适的地方进行分析，而跳转的位置需要仔细考量，否则会产生报错不准的情况。同时考虑到鲁棒性，还应该考虑两个句子中间插入一个 <= 之类的符号，也应该跳过，否则后面的处理全部都会出错。总之错误处理的细节非常多。

再者就是写程序的时候一定要先搭好框架，否则程序就会非常乱，面向对象的思想和方法还是应该多多学习和应用。

最后就是GUI界面了，那各位大佬们的比起来还是有点简陋了，不过java自带的图形界面库的功能还是稍微有点局限了，不过还是自己能力有限，还是应该多学习才行。