****

**南京航空航天大学**

**编译原理I**

**课程设计报告**

|  |  |
| --- | --- |
| 姓 名 | 孙海龙 |
| 学 号 | 181910720 |
| 学 院 | 计算机科学与技术学院 |
| 专 业 | 计算机科学与技术 |
| 班 级 | 1619101 |
| 指导教师 | 谢强 |

二〇二一年十二月

目录

[1.设计任务 4](#_Toc91004527)

[1.1 PL/0语言的BNF描述（扩充的巴克斯范式表示法） 4](#_Toc91004528)

[1.2 假想目标机代码 5](#_Toc91004529)

[1.3假想机结构 5](#_Toc91004530)

[1.3.1 存储器 5](#_Toc91004531)

[1.3.2 寄存器 5](#_Toc91004532)

[1.3.3 单元运算 5](#_Toc91004533)

[1.4活动记录 6](#_Toc91004534)

[1.5设计要求 6](#_Toc91004535)

[1.5.1总体设计 6](#_Toc91004536)

[1.5.2.系统设计 7](#_Toc91004537)

[2.系统设计 9](#_Toc91004538)

[2.1词法分析器设计 9](#_Toc91004539)

[2.1.1设计思想 9](#_Toc91004540)

[2.1.2符号种别码 10](#_Toc91004541)

[2.1.3词法分析相关函数 11](#_Toc91004542)

[2.1.4词法分析错误处理 11](#_Toc91004543)

[2.2语法分析器 12](#_Toc91004544)

[2.2.1递归下降分析程序 12](#_Toc91004545)

[2.2.2语法分析器的功能 12](#_Toc91004546)

[2.2.3带有错误恢复的语法分析处理 16](#_Toc91004547)

[2.3语义分析 17](#_Toc91004548)

[2.3.1符号表管理 17](#_Toc91004549)

[2.3.2目标代码的产生分析 19](#_Toc91004550)

[2.4语法语义分析器 20](#_Toc91004551)

[2.4.1语法语义分析函数 20](#_Toc91004552)

[2.4.2错误处理 21](#_Toc91004553)

[2.5具体代码生成 22](#_Toc91004554)

[2.5.1针对CALL指令传递参数的指令设计 23](#_Toc91004555)

[2.5.2解释执行 23](#_Toc91004556)

[3. 功能完善 24](#_Toc91004557)

[3.1内存优化 24](#_Toc91004558)

[3.1.1内存优化算法分析 24](#_Toc91004559)

[3.1.2算法执行数据栈和符号表变化 25](#_Toc91004560)

[3.1.3可视化举例 26](#_Toc91004561)

[3.2语法分析采用同步符号集 27](#_Toc91004562)

[3.2.1同步符号集算法分析 27](#_Toc91004563)

[3.2.2语法分析错误举例 28](#_Toc91004564)

[3.3词法分析支持注释 29](#_Toc91004565)

[3.3.1支持注释算法分析与程序举例 29](#_Toc91004566)

[4.测试分析 30](#_Toc91004567)

[4.1测试递归 30](#_Toc91004568)

[4.2测试多参数传参数 31](#_Toc91004569)

[4.3测试循环函数 32](#_Toc91004570)

[4.4测试多函数调用 33](#_Toc91004571)

[4.5测试符号类型的错误使用 35](#_Toc91004572)

[4.6测试重复定义，调用函数参数不匹配，使用未定义符号 36](#_Toc91004573)

[4.7测试中间代码优化 38](#_Toc91004574)

[5.课设总结 39](#_Toc91004575)

[6．参考文献 40](#_Toc91004576)

# 1.设计任务

一个PASCAL语言子集（PL/0）编译器的设计与实现。

## 1.1 PL/0语言的BNF描述（扩充的巴克斯范式表示法）

<prog> → program <id>；<block>

<block> → [<condecl>][<vardecl>][<proc>]<body>

<condecl> → const <const>{,<const>};

<const> → <id>:=<integer>

<vardecl> → var <id>{,<id>};

<proc> → procedure <id>（[<id>{,<id>}]）;<block>{;<proc>}

<body> → begin <statement>{;<statement>}end

<statement> → <id> := <exp>

|if <lexp> then <statement>[else <statement>]

|while <lexp> do <statement>

|call <id>（[<exp>{,<exp>}]）

|<body>

|read (<id>{，<id>})

|write (<exp>{,<exp>})

<lexp> → <exp> <lop> <exp>|odd <exp>

<exp> → [+|-]<term>{<aop><term>}

<term> → <factor>{<mop><factor>}

<factor>→<id>|<integer>|(<exp>)

<lop> → =|<>|<|<=|>|>=

<aop> → +|-

<mop> → \*|/

<id> → l{l|d} （注：l表示字母）

<integer> → d{d}

注释：

<prog>：程序 ；<block>：块、程序体 ；<condecl>：常量说明 ；<const>：常量；<vardecl>：变量说明 ；<proc>：分程序 ； <body>：复合语句 ；<statement>：语句；<exp>：表达式 ；<lexp>：条件 ；<term>：项 ； <factor>：因子 ；<aop>：加法运算符；<mop>：乘法运算符； <lop>：关系运算符。

## 1.2 假想目标机代码

|  |  |
| --- | --- |
| **中间代码** | **代码解释** |
| LIT 0 ，a | 取常量a放入数据栈栈顶 |
| OPR 0 ，a | 执行运算，a表示执行某种运算 |
| LOD L ，a | 取变量（相对地址为a，层差为L）放到数据栈的栈顶 |
| STO L ，a | 将数据栈栈顶的内容存入变量（相对地址为a，层次差为L） |
| CAL L ，a | 调用过程（转子指令）（入口地址为a，层次差为L） |
| INT 0 ，a | 数据栈栈顶指针增加a |
| JMP 0 ，a | 无条件转移到地址为a的指令 |
| JPC 0 ，a | 条件转移指令，转移到地址为a的指令 |
| RED L ，a | 读数据并存入变量（相对地址为a，层次差为L） |
| WRT 0 ，0 | 将栈顶内容输出 |

## 1.3假想机结构

### 1.3.1 存储器

|  |  |
| --- | --- |
| **存储器** | **作用** |
| 存储器CODE | 存放当前要执行的代码 |
| 数据存储器STACK（栈） | 指向数据栈STACK的栈顶 |

### 1.3.2 寄存器

|  |  |
| --- | --- |
| **寄存器** | **作用** |
| 指令寄存器I | 存放当前运行过程的数据区在STACK中的起始地址 |
| 栈顶指示器寄存器T | 向数据栈STACK的栈顶 |
| 基地址寄存器B | 存放当前运行过程的数据区在STACK中的起始地址 |
| 程序地址寄存器P | 存放下一条要执行的指令地址 |

### 1.3.3 单元运算

该假想机没有供运算用的寄存器。所有运算都要在数据栈STACK的栈顶两个单元之间进行，并用运算结果取代原来的两个运算对象而保留在栈顶。

## 1.4活动记录

|  |  |
| --- | --- |
| **活动记录** | **存储信息** |
| RA | 返回地址 |
| DL | 调用者的活动记录首地址 |
| SL | 保存该过程直接外层的活动记录首地址 |

|  |
| --- |
|  |
| RA |
| DL |
| SL |

图1：数据栈活动记录

## 1.5设计要求

### 1.5.1总体设计

（1）PL/0语言可以看成PASCAL语言的子集，它的编译程序是一个编译解释执行系统。PL/0的目标程序为假想栈式计算机的汇编语言，与具体计算机无关。

PL/0的编译程序和目标程序的解释执行程序可以采用C、C++、Java等高级语言书写。

其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需要生成相应的目标代码时，则调用代码生成程序。

（2）根据所给的PL/0语言的BNF描述，该语言的组成单词包括以下元素：

关键字（程序保留字）{program，const，var，procedure，begin，end，if，else，then，call，while，do，read，write}

运算符：{+，-，\*，/，odd}

界符：{“，”，“;”，“（”，“）”}

关系运算符：{=，<,<=,>,>=,<>}

数字：只能为整型，且常量不可以是负数

标识符：由用户定义，以字母开头，由数字和字母组成

词法分析程序读取源文件，识别出上述关键字、界符、关系运算符、数字、标识符五种元素，输出到txt文件中，供后续的语法分析程序使用。其中，

(1)除标识符外，剩余的每一种字符可以用数字表示；

(2)而标识符则统一用一个数字表示其为标识符，再将标识符本身存储起来；

(3)数字的存储和标识符存储类似。

### 1.5.2.系统设计

编译器逐行扫描高级语言程序源程序，编译的过程如下：

1. 词法分析

识别关键字、字面量、标识符 (变量名、数据名)、运算符、注释行(给人看的，一般不处理)、特殊符号(续行、语句结束、数组)等六类符号，分别归类等待处理。

1. 语法分析

一个语句看作一串记号 (Token)流，由语法分析器进行处理。按照语言的文法检查判定是否是合乎语法的句子。如果是合法句子就以内部格式保存，否则报错。直至检查完整个程序。

1. 语义分析

语义分析器对各句子的语法做检查：运算符两边类型是否相兼容；该做哪些类型转换 (例如，实数向整数赋值要"取整")；控制转移是否到不该去的地方；是否有重名或者使语义含糊的记号，等等。如果有错误，则转出错处理，否则可以生成执行代码。

1. 中间代码生成

中间代码是向目标码过渡的一种编码，其形式尽可能和机器的汇编语言相似，以便下一步的代码生成。但中间码不涉及具体机器的操作码和地址码。采用中间码的好处是可以在中间码上做优化。

1. 优化

对中间码程序做局部优化和全局 (整个程序)优化，目的是使运行更快，占用空间最小。局部优化是合并冗余操作，简化计算，例如x:=0可用一条"清零"指令替换。全局优化包括改进循环、减少调用次数和快速地址算法等。

1. 代码生成

由代码生成器生成目标机器的目标码 (或汇编)程序，其中包括数据分段、选定寄存器等工作，然后生成机器可执行的代码。

词法分析程序

语法语义分析程序

代码生成程序

表格管理程序

出错处理程序

PL/0源程序

目标程序

图2：编译程序的结构

结构的相关作用：

|  |  |
| --- | --- |
| **程序** | **作用** |
| 词法分析程序 | 使用循环分支方法实现PL/0语言的词法分析器，能够读入使用PL/0语言书写的源程序 |
| 语法语义分析程序 | 对给出的PL/0语言进行分析，为block等主要模块设计子程序 |
| 代码生成程序 | 按照语义分析的结果生成目标代码 |
| 解释执行程序 | 当语法语义分析没有检测到错误时，运行生成的目标代码程序 |

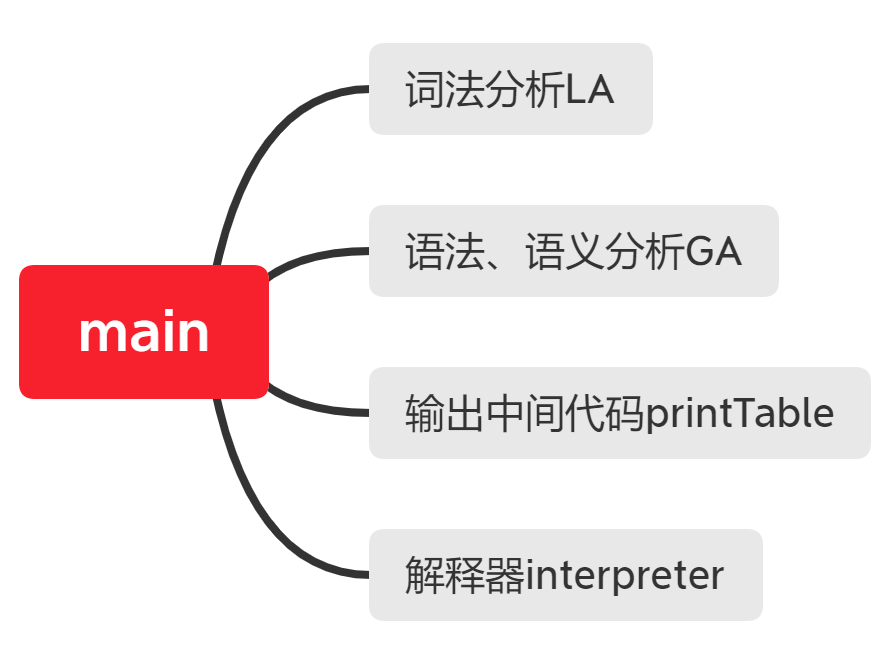


图3：编译器的函数构造

# 2.系统设计

## 2.1词法分析器设计

### 2.1.1设计思想

词法分析器获取源程序中的单词。使用循环分支方法实现PL/0语言的词法分析器，能够读入使用PL/0语言书写的源程序，输出单词符号串及其属性到中间文件中，具有一定的错误处理能力，给出词法错误，见图4，5。

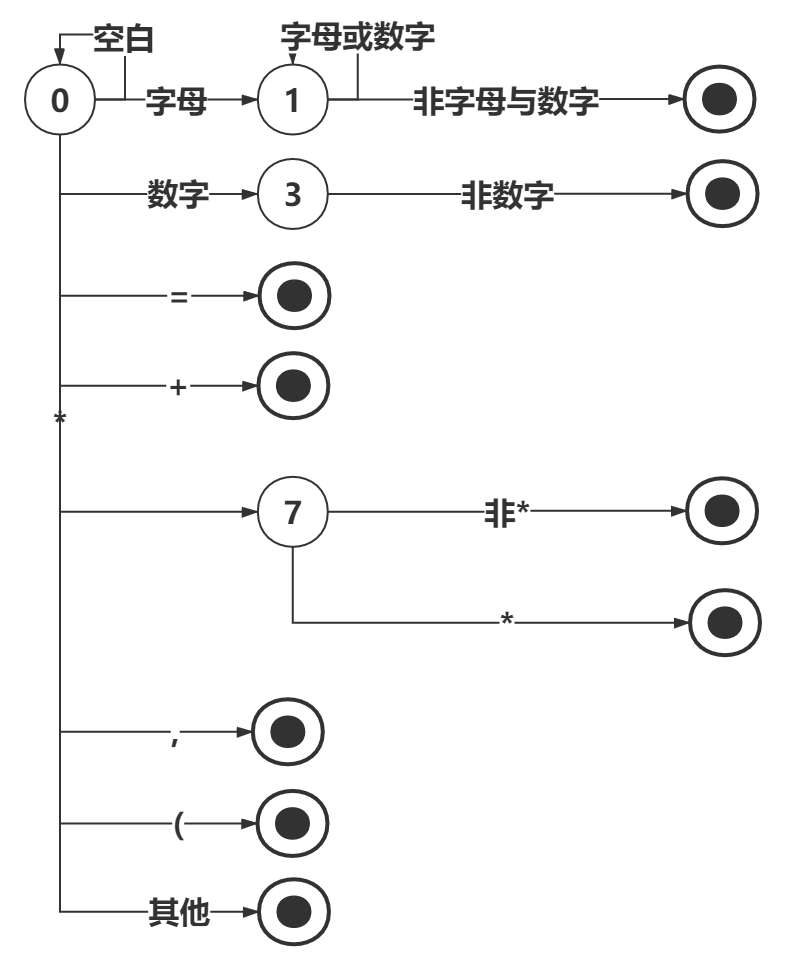


图4：词法分析词法转换图

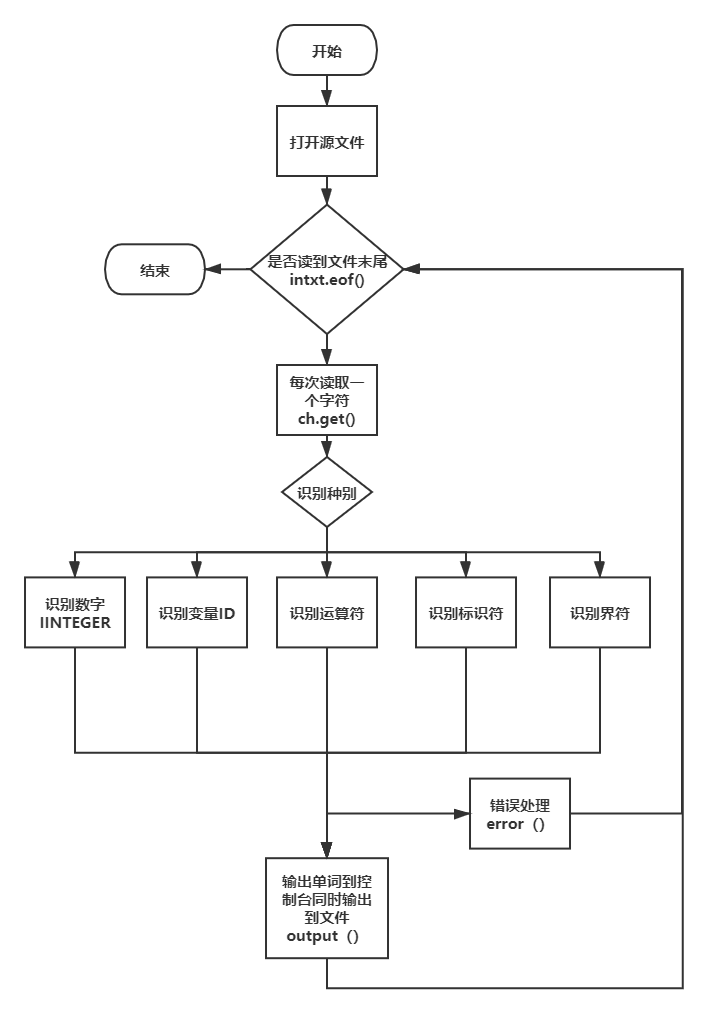


图5：词法分析程序框图

### 2.1.2符号种别码

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类型 | 种别 | 种别码 | 类型 | 种别 | 种别码 |
| KEYWORD | Program | 0 | lop | = | 16 |
| KEYWORD | const | 1 | lop | <> | 17 |
| KEYWORD | var | 2 | lop | < | 18 |
| KEYWORD | procedure | 3 | lop | <= | 19 |
| KEYWORD | begin | 4 | lop | > | 20 |
| KEYWORD | end | 5 | lop | >= | 21 |
| KEYWORD | if | 6 | aop | + | 22 |
| KEYWORD | then | 7 | aop | - | 23 |
| KEYWORD | while | 8 | mop | \* | 24 |
| KEYWORD | do | 9 | mop | / | 25 |
| KEYWORD | call | 10 | sop | , | 26 |
| KEYWORD | read | 11 | sop | ( | 27 |
| KEYWORD | write | 12 | sop | ( | 28 |
| KEYWORD | odd | 13 | eop | ; | 29 |
| ID | l{l|d} | 14 |  |  |  |
| INT | d{d} | 15 |  |  |  |

### 2.1.3词法分析相关函数

|  |  |
| --- | --- |
| **词法分析函数** | **函数解释** |
| int LA(); | 词法分析主函数 |
| bool isBC(char ch); | 识别空格，制表符，换行符 |
| bool isDigit(char ch); | 识别整数 |
| bool isLetter(char ch); | 识别字母 |
| void concat(sring& strToken, char ch); | 将字符ch连接到单词 |
| int Reserve(string strToken); | 识别标识符 |
| void Retract(); | 读入单词指针后退一个字符 |

### 2.1.4词法分析错误处理

|  |  |
| --- | --- |
| **错误提示同时输出错误所在行列** | **含义** |
| "[Lexical error]" +line + column | 词法错误 |
| strToken + " Invalid ID"+ line + column | 无效变量 |
| "Lexical error" + "：=AOP"+ line +column + "/.missing =./" | 赋值符号缺少= |
| ch +"UNKNOW" + line + column | 未知词法错误 |

## 2.2语法分析器

### 2.2.1递归下降分析程序

语法分析结合BNF产生式，利用递归下降的方法实现。具体实现方式是，为每一个非终结符编写一个子程序，以<prog> → program <id>；<block>此产生式为列，用伪码描述其子程序如下：、

|  |
| --- |
| void prog**(){**  **if(**当前指向的字符**==**program**){**  指向下一个字符；  **if(**当前指向的字符**==**id**){** //是标识符  指向下一个字符；  **if(**当前指向的字符**==**；**){**  block**();**  **}else{**  error**;**  **}**  **}**  **}else{**  error  **}**  **}** |

说明：由于之前的词法分析使用数字来描述的，故此处的program，id(即标识符)，分号等都可以用相应的数字代替。

### 2.2.2语法分析器的功能

语法分析器是编译过程的核心部分，它的任务是在词法分析识别出单词符号串的基础上，分析并判定程序的语法结构是否符合词法规则。语法分析器在编译程序中的地位如图所示。

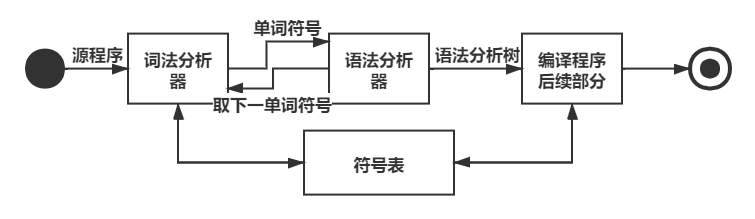


图6：语法分析器在编译过程的作用

递归下降分析器

递归下降分析器：当一个文法满足LL（1）条件时，我们就可以为它构造一个不带回溯的自上而下分析程序，这个分析程序是由一组递归过程组成的，每个过程对应文法的一个非终结符。

该PL/0编译器语法分析采用递归下降分析



图7：语法分析程序框图

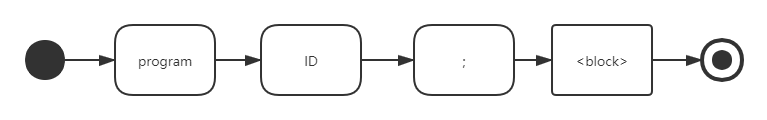


图8 <prog>程序的语法分析框图

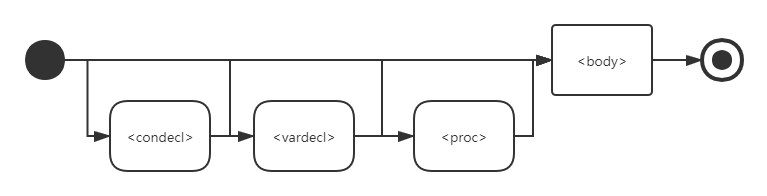


图9 <block>程序的语法分析框图

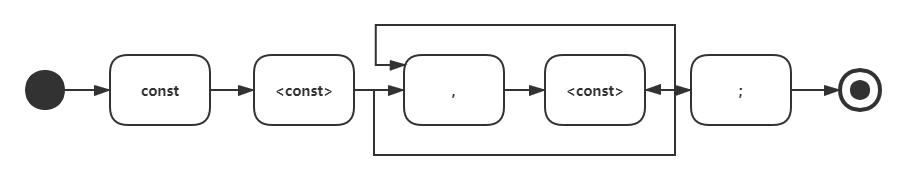


图10 <condecl>程序的语法分析框图

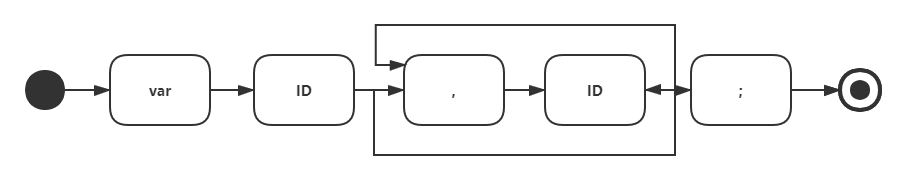


图11 <vardecl>程序的语法分析框图

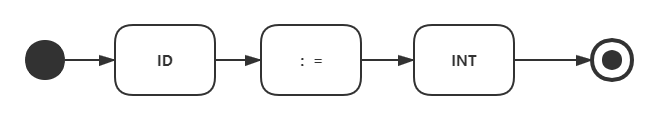


图12 <const>程序的语法分析框图

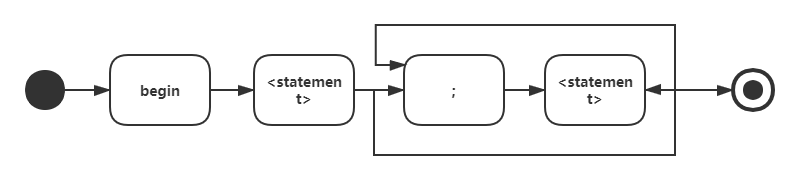


图13 <body>程序的语法分析框图

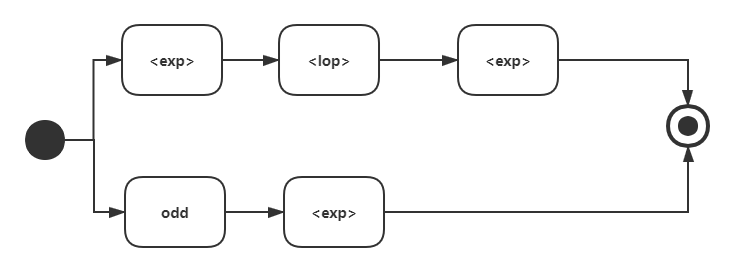


图14 <lexp>程序的语法分析框图

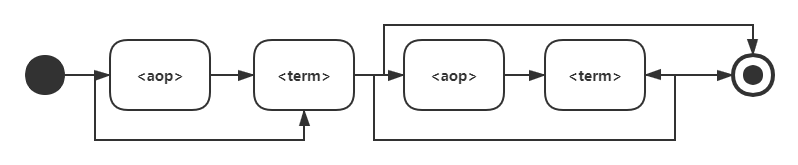


图15 <exp>程序的语法分析框图

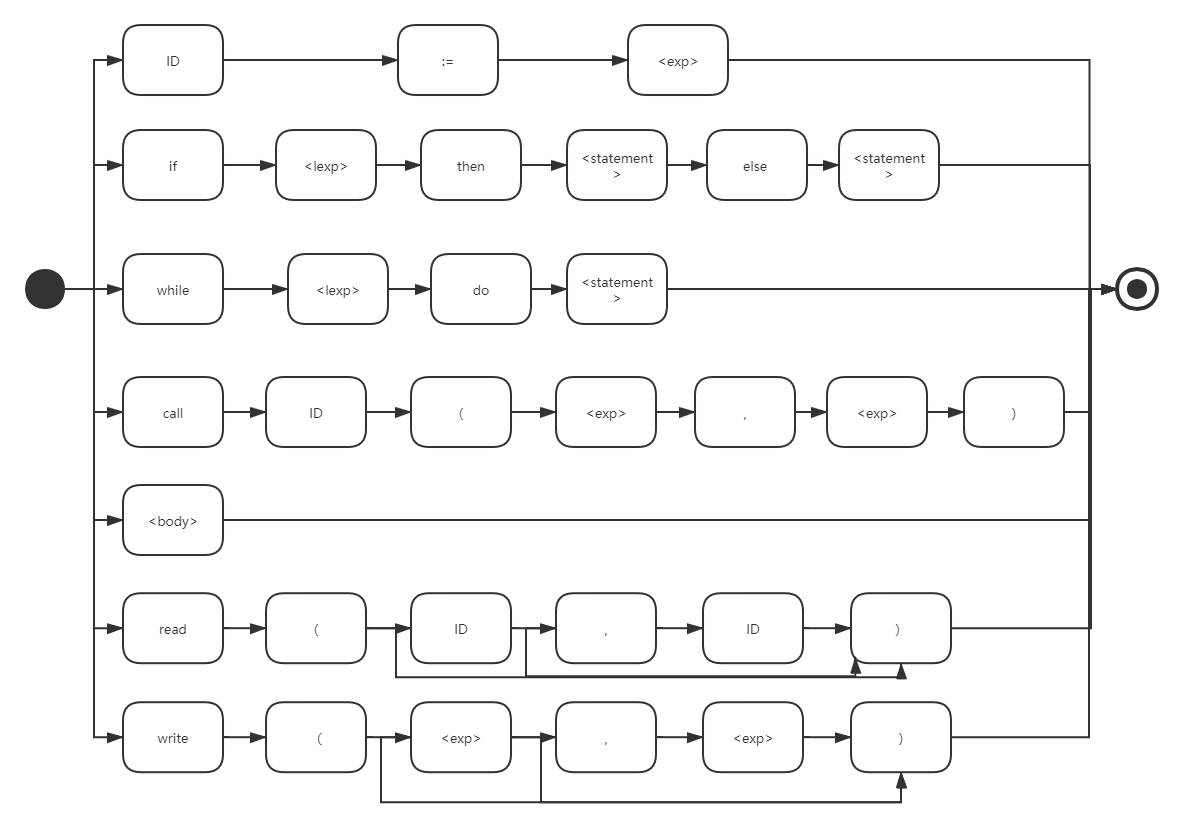


图16 <statement>程序的语法分析框图

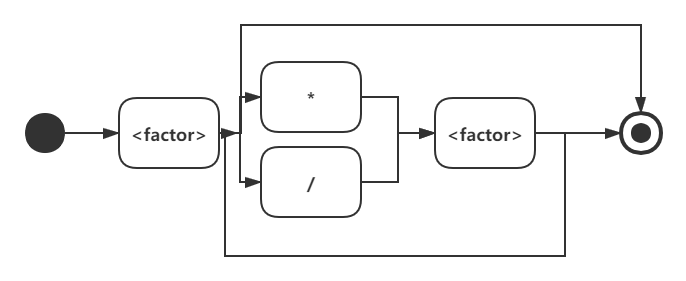


图17 <term>程序的语法分析框图

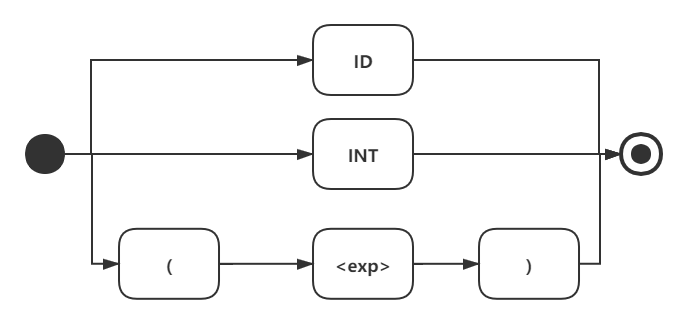


图18 <factor>程序的语法分析框图

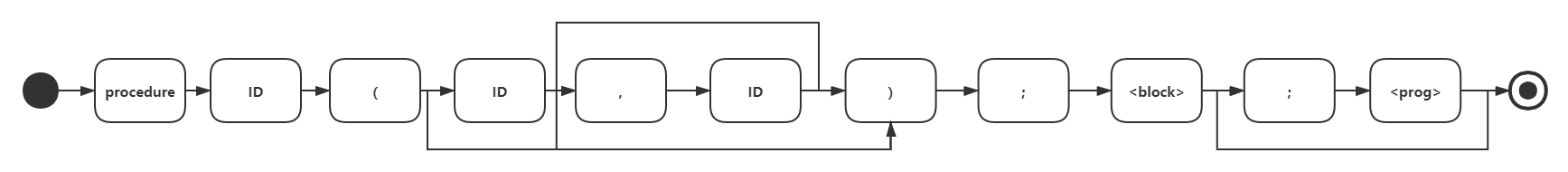


图19 <proc>程序的语法分析框图

### 2.2.3带有错误恢复的语法分析处理

同步符号集错误处理

在进入某个语法单位时，检查当前符号是否属于该语法单位的开始符号集合。若不属于，则滤去开始符号和后继符号集合外的所有符号。

在语法单位分析结束时，检查当前符号是否属于调用该语法单位时应有的后继符号集合。若不属于，则滤去后继符号和开始符号集合外的所有符号。

同步符号集的选择：

1. 把FOLLOW(A)中的所有符号放入非终结符A的同步符号集。如果我们跳读一些输入符号直到出现FOLLOW(A)中的符号，这样就可能使分析继续下去。
2. 把FIRST(A)中的符号加入非终结符A的同步符号集，当FIRST(A)中的一个符号在输入中出现时，可以进入该非终结符进行语法分析。

|  |  |  |
| --- | --- | --- |
| **非终结符** | **FIRST集** | **FOLLOW集** |
| <prog> | program |  |
| <block> | const var procedure begin | procedure begin ; |
| <condecl> | const | var procedure begin |
| <const> | <id> | ; |
| <vardecl> | Var | procedure begin |
| <proc> | Procedure | begin |
| <body> | begin | ; end else procedure begin |
| <statement> | <id> if while call begin read write | ; end else |
| <lexp> | + - <id> <integer> ( | then do |
| <exp> | + - <id> <integer> ( | ; end else then do ) = <> < <= > >= |
| <term> | <id> <integer> ( | ; end else then do ) = <> < <= > >= + - |
| <factor> | <id> <integer> ( | ; end else then do ) = <> < <= > >= + - \* / |
| <lop> | = <> < <= > >= | + - <id> <integer> ( |

## 2.3语义分析

语义分析在语法分析的基础上完成，涉及到的操作有符号表的管理和目标代码的生成，分别对应说明语句和处理语句，下面分开来说。

### 2.3.1符号表管理

(1)符号表中存储以下数据：定义的变量、定义的常量、定义的过程；

(2)定义的变量需要存储：变量的标识符，变量定义所在层次，相对于该层次基地址的偏移量（对于基地址将在后面活动记录中详细说明）

(3)定义的常量需要存储：常量的标识符，常量的值，定义所在层次

(4)定义的过程需要存储：过程名，过程处理语句的开始地址（处理语句不是说明语句，说明语句中涉及到符号表的操作，而处理语句中涉及到产生目标代码的操作），过程定义所在层次；

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名字** | **种类** | **值** | **层次** | **相对地址** | **参数个数** | **变量出现次数** |
| A | Constant | 35 | x | x | x | X |
| I | variable | x | lev | dx | x | 2 |
| Sum | procedure | x | Lev | x | 4 | X |

符号表的翻译：

|  |  |
| --- | --- |
| 登记符号表函数 | 含义 |
| addVar(string name, int level, int adr) | 将识别的常量登记到符号表 |
| addConst(string name, int level, int val) | 将识别的变量登记到符号表 |
| addProcedure(string name, int level, int adr) | 将识别的过程登记到符号表 |

1. 常量说明的翻译（伪代码表示）：

|  |
| --- |
| void const**(){**  **if(**当前指向的字符**==**id**){**  指向下一个字符；  用name记录下字符；  **if(**当前指向的字符**==** **:=** **){**  指向下一个字符；  **if(**当前指向的字符**==**数字**){**  用value记录下值；  enter**(**name**,**value**,**level**,**addr**);**//将其记录到符号表  **}**  **}**  **}**  **}** |

（2）变量说明的翻译：

将识别的常量登记到符号表 ：enter(name,level,addr);

|  |
| --- |
| void vardecl**(){**  **if(**当前指向的字符**==**id**){**  用name记录下字符；  enter**(**name**,**level**,**addr**);**  指向下一个字符；  **while(**当前指向的字符**==**，**){**  指向下一个字符；  **if(**当前指向的字符**==**id**){**  用name记录下字符；  enter**(**name**,**level**,**addr**);**  指向下一个字符；  **}**  **}**  **}**  **}** |

（3）求得层次差和偏移量：

在整个语法分析函数中，定义两个全局变量level和address，level初始化为1，其中主程序是第一层，每次遇到过程嵌套定义就将level加1，过程定义结束后再将其减1，这样下来就实现了获得说明语句所在层次的功能；address初始化为0，每次在符号表中登录变量后就将address加1，address的作用是，结合level，可以获得每一层有几个变量，这样在活动记录的分配（也叫运行时存储空间的划分，运行时只为变量事先划分存储空间），可以根据该层变量的个数划分相应的存储空间。

### 2.3.2目标代码的产生分析

这部分涉及到翻译模式相关的知识，由于说明语句是不产生目标代码的，在结合PL/0的BNF描述，会产生目标代码的只有<block>、<body>、<statement>、<lexp>、<exp>、<term>、<factor>、<lop>、<mop>,下面一次用伪码说明其翻译模式：

（1）在进入block的第一步就先产生一个无条件跳转指令，由于中间子过程嵌套还会产生目标代码 ，导致不知道跳转到哪里才是主程序的入口，这里我们先记下这个无条件跳转指令在目标代码中的位置，当程序分析完变量说明、常量说明和过程说明即将进入语句处理的分析时，这时目标代码的地址就是主程序的入口，我们根据刚才记下的无条件跳转指令的位置将这个地址回填到跳转指令的目标地址上，从而目标代码第一句就是跳转到主程序入口处的指令，对于程序中嵌套程序，用这个方法依然是可行的；

（2）除此之外，在进行其他任何分析之前，还要记录下主程序或者当前过程的数据量，即符号表中的变量数目，也即address，以便后面活动记录的划分可以在结束过程后返回到原来的数据栈的位置；

（3）另外，如果是从过程进入到block，在产生过程调用指令需要用到这个入口地址，所以这个过程入口地址也要记录下来，用来填写过程调用指令的目标位置。则在进行其他任何分析之前，此时符号表中最新的一项必定是该过程的相关信息，我们可以将本过程的value值设为其入口地址，因为value值对于符号表procedure来说是无用的，即将开始地址登入到符号表，从而在做任何分析前我们也要记录下符号表中的最新项（记录其序号），在经过变量说明分析、常量说明分析、过程分析后即将进入语句处理分析时，便得到了该过程的入口地址，将其登录到刚刚记录的那个符号表最新项的value域；

可以通过判断符号表中的最新项的序号是否为0来判断是从过程进入<block>还是从过程进入<block>

（4）进行完上述三步操作后即可进入语句的处理部分；

（5）在语句的处理部分完成后，生成目标代码的退出过程或主程序的语句，然后恢复address和符号表的序号。

（6）重点1：作用域的确定，分为两步：一步是定义变量时，我们要查询在本层有无同名的变量已经定义，若有，则报错；二步是变量使用时，首先查询本层有无定义该变量，若有，则使用本层定义的变量，若无，则寻找本过程直接外层有无定义同名变量，若仍无此变量，则继续向下一个直接外层寻找，以此类推。若有，则使用最靠近本层的那个定义，若无，则报错。由于本符号表采用的是数组管理，在实现第一步时，在变量（常数、过程）登录符号表时记录其所在的层次和所在层次的名字，若符号表中存在变量（常量、过程）的名字、所在层次和所在层次的名字均相同，则报错，若无，则将定义的变量加到符号表中。在实现第二步时，结合语法的特点分析，首先在本调用过程所在的过程进行查找有无定义该变量，若有，则就使用本过程的变量，若无，则根据前面在符号表中记录的本调用过程所在的直接外层的名字查找该直接外层有无定义此变量，以此类推，若最终有定义，则使用该变量，否则保存。

重点2：过程传递参数：我把在过程定义时的形式参数，当作在本过程定义变量进行处理，在调用过程中出传入的实在参数当作对相应的形式参数进行赋值，具体说来就是用该过程在符号表中的size域存储该过程中的形参个数，再用一个变量记录过程中定义参数的个数（具体做法是定义一个变量，每次进入block时置为3，用来存储SL、DL、RA，然后遇到变量定义就加一来记录）当遇到过程定义时，将其说明的形参按照在本过程中定义变量的相关属性登录到符号表中，并且他们在活动记录的存储位置是紧接着SL，DL，RA；在遇到调用过程时，首先根据栈顶、栈顶-1、栈顶-2、栈顶-形参个数+1根据所存储的位置进行赋值，然后再分配相应的活动存储空间（如果刚开始就分配，则会导致栈顶更新，失去了原来的栈顶元素），该活动存储空间的大小应该是参数个数+本过程定义的变量个数+3。

# 2.4语法语义分析器

### 2.4.1语法语义分析函数

|  |  |
| --- | --- |
| **语法语义分析函数** | **函数解释** |
| int GA(); | 语法分析主函数 |
| void ReadLine(); | 每次从词法分析器读入一个单词 |
| void addError(int type, string name = ""); | 错误处理函数，处理extra和missing |
| void Proc(); | 识别<prog> |
| Void Block(); | 识别<block> |
| void Condecl(); | 识别<condecl> |
| void Const (); | 识别<const> |
| void Vardecl(); | 识别<vardecl> |
| void Var(); | 识别<var> |
| void Proc(); | 识别<proc> |
| void Body(); | 识别<body> |
| void Statement(); | 识别<statement> |
| void Exp(); | 识别<exp> |
| void Term(); | 识别<term> |
| void Factor(); | 识别<factor> |
| void Lexp(); | 识别<lexp> |
| void addVar(string name, int level, int adr); | 将识别的变量登记到符号表 |
| void addConst(string name, int level, int val); | 将识别的常量登记到符号表 |
| void addProcedure(string name, int level, int adr); | 将识别的过程登记到符号表 |
| int position(string name); | 返回名称属性为name的符号在表中的位置 |
| void gen(int f, int l, int a); | 生成中间代码 |
| int findproc(); | 返回最近层次的过程在表中的位置 |
| int stringtoint(string name); | 将string转换成数值 |
| bool is\_the\_same\_level(string name, int lev); | 符号表中存在名称为name的符号且在lev层 |
| bool is\_pre\_level(string name, int lev); | 存在名称为name的符号且小于等于lev层 |
| void interpreter() | 解释器 |
| int getBase(int nowBp, int lev) | 根据层差求出符号在栈中的基地址 |
| void printPcode(); | 在控制台中输出中间代码 |

### 2.4.2错误处理

|  |  |
| --- | --- |
| **错误代码** | **错误类型** |
| -1 | [error][%d,%d] Unknown error\n |
| 0 | [Grammar error][ unit.row, unit.col] Spell error \"program\"\n |
| 1 | [Grammar error][unit.row, unit.col] Missing identifier after \"program\"\n |
| 2 | [Grammar error][ last\_unit.row, last\_unit.col] Missing \";\" \n" |
| 3 | [Grammar error][unit.row, unit.col] Missing \")\" \n |
| 4 | [Grammar error][unit.row, unit.col] Missing \"(\" \n |
| 5 | [Grammar error][unit.row, unit.col] Missing \",\" \n |
| 6 | [Grammar error][unit.row, unit.col] Missing identifier \n |
| 7 | [Grammar error][unit.row, unit.col] Missing compare operator \n |
| 8 | [Grammar error][unit.row, unit.col] Missing assignment operator \n |
| 9 | [Grammar error][unit.row, unit.col] Missing \"then\" \n |
| 10 | [Grammar error][unit.row, unit.col] Missing \"do\" \n |
| 11 | [Grammar error][unit.row, unit.col] Missing \"begin\" \n |
| 12 | [Grammar error][unit.row, unit.col] Missing \"end\" \n |
| 13 | [Grammar error][unit.row, unit.col] after\":=\" should be a number \n |
| 14 | [Grammar error][unit.row, unit.col] Missing \":\" \n |
| 15 | [Grammar error][unit.row, unit.col] Missing \":=\" \n |
| 16 | [Grammar error][unit.row, unit.col] Missing \"program\ |
| 17 | [Grammar error][unit.row, unit.col] the word is superfluous \n |
| 18 | [error][unit.row, unit.col] not exist name\n |
| 19 | [error][unit.row, unit.col] name is not a variable \n |
| 20 | [error][unit.row, unit.col] name is not a const \n |
| 21 | [error][unit.row, unit.col] name is not a procedure \n |
| 22 | [error][unit.row, unit.col] The number of parameters does not match \n |
| 23 | [error][unit.row, unit.col] Duplicate definition name \n |

## 2.5具体代码生成

（1） JMP指令的产生和回填：遇到program、procedure的定义，就产生一个jmp 0 0 的目标指令。Program和procedure都含有 block部分，这是必不可少的，所以也就都会含有body部分，这样就可以得到begin。 每当遇到begin单元的时候，就对jmp进行回填。 回填的方式是将当前的cx的值，填回 jmp指令的第二个参数值。

（2） JPC指令，跳转转移指令，判断栈顶的值是否满足。用在if语句和while语句中。If语句如果不满足条件，就会跳过then后面的执行语句，while语句类似。

（3） INT指令：遇到begin，就说明开始了 program或者 procedure的body，需要在运行栈中为它开辟空间。而dx在前面已经完成了计数，所以dx已经可以作为它的参数。

（4） LIT 0 ，a指令：将常量放到栈顶， 这是在遇到如 c:=a+10, 这是一个Exp(),10是一个因子，最终落在factor中，所以在factor中， 如果遇到一个常数，就产生LIT 0 ，a的指令。

（5） LOD L ，a指令：将变量放到栈顶，所以如上面的c:=a+10,a也是一个因子，落在factor中，判断它是一个id，然后通过 position()函数找到a在符号表中的位置，通过符号表中的信息，得到该变量的层差和偏移地址。这样就得到了 LOD指令的两个参数。

（6） STO L ，a指令：如上面的c:=a+10，再计算完a+10之后，需要将栈顶的值写入c，就是通过 position()函数找到c在符号表中的位置，通过符号表中的信息，得到该变量的层差和偏移地址。然后将栈顶的值写入。

（7） OPR指令：这是操作指令，所以像expression中，如果存在相应的操作，如+、-等，就产生OPR指令。

（8） CAL指令，遇到 call语句时，就产生CAL指令，通过 position()函数，找到想要调用的p的过程在符号表中的位置，找到p开始的地址即可。

（9） RED指令：遇到read语句时，就产生RED指令，通过然后通过 position()函数找到变量在符号表中的位置，通过符号表中的信息，得到该变量的层差和偏移地址。

（10） WRITE 指令：当遇到write（exp）指令时，当exp()函数处理完之后就可以产生WRITE指令。将栈顶的内容输出。

### 2.5.1针对CALL指令传递参数的指令设计

为实现call函数带参数的进行调用，本课程设计扩充了原有的假想机指令集，增加了一条指令：STO -1 3 ，对STO功能进行拓展，将当前栈顶的值放在栈顶加3的位置，并退栈，即S[t+3]=S[t]。语义分析时，如果调用过程包含参数，那么在Exp()中准备好和传入参数有关的目标代码，在所有形参准备好后，根据形参个数，生成相等数量的STO指令。

### 2.5.2解释执行

解释器是将目标代码解释运行的模块。语法语义分析结束后查看是否存在出错信息，如果不存在则访问生成的代码，模拟假想机解释执行代码的运行过程。

目标代码解释：

|  |  |  |  |
| --- | --- | --- | --- |
| **伪操作码** | **层差值** | **相对地址** | **解释** |
| LIT | 0 | a | 将常数值取到栈顶，a为常数值 |
| LOD | l | a | 将变量值取到栈顶，a为偏移量，l为层差 |
| STO | l | a | 将栈顶内容送入某变量单元中 |
| CAL | l | a | 调用过程，a为过程地址，l为层差 |
| INT | 0 | a | 被调用的过程开辟a个单元的数据区 |
| JMP | 0 | a | 无条件跳转至a地址 |
| JPC | 0 | a | 条件跳转，当栈顶布尔值非真则跳转至a地址，否则顺序执行 |
| OPR | 0 | 0 | 过程调用结束后,返回调用点并退栈 |
| OPR | 0 | 1 | 栈顶元素取反 |
| OPR | 0 | 2 | 次栈顶与栈顶相加，退两个栈元素，结果值进栈 |
| OPR | 0 | 3 | 次栈顶减去栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 4 | 次栈顶乘以栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 5 | 次栈顶除以栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 6 | 栈顶元素的奇偶判断，结果值在栈顶 |
| OPR | 0 | 7 |  |
| OPR | 0 | 8 | 次栈顶与栈顶是否相等，退两个栈元素，结果值进栈 |
| OPR | 0 | 9 | 次栈顶与栈顶是否不等，退两个栈元素，结果值进栈 |
| OPR | 0 | 10 | 次栈顶是否小于栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 11 | 次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 12 | 次栈顶是否大于栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 13 | 次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈 |
| OPR | 0 | 14 | 栈顶值输出至屏幕 |
| OPR | 0 | 15 | 屏幕输出换行 |
| OPR | 0 | 16 | 从命令行读入一个输入置于栈顶 |
| OPR | 0 | 3 | 将栈顶内容转移到栈顶加3的位置 |

# 3. 功能完善

## 3.1内存优化

（1）形参传参内存优化：在过程P传参给过程Q时，首先会在P活动记录栈顶上开辟形参个数的空间，将所需传参的值传到栈顶，当建立被调用过程Q的活动记录的时候，同时传入形式参数的个数，栈同时退形参个数个空间，这样可以做到形参传参内存优化。

（2）主函数变量内存优化：未使用过的变量会被优化。主过程刚开始会开三个空间加上变量个数个空间。但是如果有一些变量在主<body>中和函数调用中未被使用，此时我们就不需要为之开空间。

采取措施：在符号表多开了num属性，记录每个变量被使用的次数，如图标红一栏。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名字** | **种类** | **值** | **层次** | **相对地址** | **参数个数** | **变量出现次数** |
| A | Constant | 35 | x | x | x | X |
| I | variable | x | lev | dx | x | 2 |
| Sum | procedure | x | Lev | x | 4 | X |

当进行一遍扫描之后，此时会统计出所有符号中变量出现的次数，我们此时对Pcode进行回填，主要是INT指令开辟空间个数的回填和符号表相对地址的更新。

### 3.1.1内存优化算法分析

|  |
| --- |
| int count **=** 0**;** //记录未被使用的变量个数  int i **=** 1**;**  //在符号表中遍历第0层主函数的符号  **while** **(**SymTable**[**i**].**num **&&** **!**SymTable**[**i**].**level**)**  **{**  //如果是变量且只定义过一次，后续没有使用，进行后续优化  **if** **(**SymTable**[**i**].**num **==** 1 **&&** SymTable**[**i**].**kind**==**variable**)**  **{**  int j **=** i **+** 1**;**  //后续所有定义的变量相对地址都要减一  **while** **(**SymTable**[**j**].**num **&&** **!**SymTable**[**j**].**level**)**  **{**  SymTable**[**j**].**adr**--;**  j**++;**  **}**  //未被使用的变量个数加一  count**++;**  **}**  i**++;**  **}**  //回填INT指令开辟的空间数，一边扫描之后，解释器之前  Pcode**[**mm**].**a **-=** count**;** |

### 3.1.2算法执行数据栈和符号表变化



图20：数据栈的变化

### 3.1.3可视化举例

若主函数定义了5个变量，分别是a,b,c,d,e。但是b,d,e未使用需要优化掉。

**优化前：**

符号表(标红需要优化)：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名字** | **种类** | **值** | **层次** | **相对地址** | **参数个数** | **变量出现次数** |
| a | variable | x | 0 | 3 | x | 2>1 |
| b | variable | x | 0 | 4 | x | 1 |
| c | variable | x | 0 | 5 | x | 3>1 |
| d | variable | x | 0 | 6 | x | 4>1 |
| e | variable | x | 0 | 7 | x | 1 |

生成INT指令：INT 0 ，8

**优化后：**

符号表(标红需要优化)：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名字** | **种类** | **值** | **层次** | **相对地址** | **参数个数** | **变量出现次数** |
| a | variable | x | 0 | 3 | x | 2 |
| c | variable | x | 0 | 4 | x | 3 |

生成INT指令：INT 0 ，5

## 3.2语法分析采用同步符号集

（1）把FIRST(A)中的符号加入非终结符A的同步符号集，当FIRST(A)中的一个符号在输入中出现时，可以进入该非终结符进行语法分析。

（2）把FOLLOW(A)中的所有符号放入非终结符A的同步符号集。如果我们跳读一些输入符号同时进行出错处理直到出现FOLLOW(A)中的符号，这样就可能使分析继续下去。

（3）当已经读完主程序，即识别完主函数的end，文件仍然未读到末尾，此时应该跳过同时进行出错处理。

### 3.2.1同步符号集算法分析

|  |
| --- |
| //准备进入该非终结符进行分析  Test**:** **while** **not(**ch in FIRST**(**A**))**  //报错同时读入下一个字符  **do**  **{**  error**();**  ch**.**get**();**  **}**  **do** ch**.**get**()**  **<非终结符程序分析>**  //该非终结符分析结束  Test**:while** **not(**ch in FOLLOW**(**A**))**  //报错同时读入下一个字符  **do**  **{**  error**();**  ch**.**get**();**  **}** |

### 3.2.2语法分析错误举例

输入程序：

//<id>后面缺少;，但读入到FOLLOW集，随即进入<block>进行分析）

program **<**id**>** **<**block**>**

//程序语法分析已经结束，但尚未读入到文件末尾 报错：not is extra end is extra

**not** end

算法分析：

|  |
| --- |
| **while** **((**ch **=** intxt**.**get**())** **!=** **-**1**)**  **{**  **if** **(!**line**)**  line**++;**  //当读入注释开头  **if** **(**ch **==** '/'**)**  **{**  ch **=** intxt**.**get**();**、  //判断是block注释还是line注释  **if** **(**ch **==** '\*'**)**  **{**  **while** **(**ch **!=** '/'**)**  **{**  ch **=** intxt**.**get**();**  //如果是注释，所读人字符不存入单词  strToken **=** ""**;**  **}**  **}**  **else**  **{**  **while** **(**ch **!=** ' '**)**  **{**  ch **=** intxt**.**get**();**  //如果是注释，所读人字符不存入单词  strToken **=** ""**;**  **}**  **}**  **if** **(**isBC**(**ch**))**  **{**  strToken **=** ""**;**  **}**  **}else** **if(...)**  **{**  **...**  **}**  **}** |

## 3.3词法分析支持注释

（1）一种是以/\*开始、以\*/结束的块注释（block comment）；

（2）另一种是以//开始、以换行符结束的单行注释（line comment）

编译器在词法分析遇到注释，不会在控制台输出该单词和输出到输出文件中

### 3.3.1支持注释算法分析与程序举例

|  |
| --- |
| /\*  作者：  测试目的：  \*/  program xxx**;**  xxxxx**;**  var xxxx**;**  procedure add**(**x**,**x**);** //xxxxxxxx  begin  **...**  end**;**    begin  **...**  end |

# 4.测试分析

## 4.1测试递归

测试程序1：

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试递归  正确输入：  无输入  正确输出：  1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040  \*/  program fibonacci**;** //斐波那契数列  const index**:=**30**;**  var **return,**i**;**  procedure fib**(**x**);**  var sum**;**  begin  sum**:=**0**;**  **if** x**<**2 then  **return:=**x  **else**  begin  call fib**(**x**-**1**);**  sum**:=**sum**+return;**  call fib**(**x**-**2**);**  sum**:=**sum**+return;**  **return:=**sum  end  end  begin  i**:=**1**;**  **while** i**<=**index **do** //循环30次  begin  call fib**(**i**);**  write**(return);**  i**:=**i**+**1 //每次更新i  end  end |

正确输入：

|  |
| --- |
| 无 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| 注意数据栈要开够，否则无法正常运行 |

## 4.2测试多参数传参数

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试多参数传参数  正确输入：  101 305 42  正确输出：  549 3  \*/  program Add**;**  const index**:=**20**;**  var **return,**a**,**b**,**c**,**sum**;**  procedure add**(**a**,**b**,**c**);** //求三者之和  var sum**;**  begin  sum**:=**0**;**  sum**:=**a**+**b**;**  sum**:=**sum**+**c**;**  **return:=**sum  end**;**  procedure add1**(**a**,**b**,**c**);**  begin  sum**:=**3  end  begin  read**(**a**);** //从控制台读入数据  read**(**b**);**  read**(**c**);**  call add**(**b**+**a**,**a**,**c**);** //多参数传递  call add1**(**a**,**b**,**c**);**  write**(return);** //控制台输出  write**(**sum**)**  end |

正确输入：

|  |
| --- |
| 101 305 42 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| 注意数据栈要开够，否则无法正常运行 |

## 4.3测试循环函数

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试循环函数  正确输入：  3 (6)  正确输出：  23 (16)  \*/  program shl**;**  const a**:=**20**,**b**:=**10**;**  var c**,**d**,**e**,**flag**;**  procedure shl1**(**k**);**  var sum**;**  begin  sum**:=**20**;**  **while** k**>=**0 **do**  begin  **if** odd k then //是奇数和加加  sum**:=**sum**+**k  **else**  sum**:=**sum**-**k**;** //是偶数和减减  k**:=**k**-**1  end**;**  flag**:=**sum  end  begin  read**(**c**);**  e**:=**c**+**2**;**  call shl1**(**e**);**  write**(**flag**)**  end |

正确输入：

|  |
| --- |
| 3 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| 20+5-4+3-2+1=23 |

## 4.4测试多函数调用

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试多函数调用  正确输入：  5  14  正确输出：  7 5 3 1 52  15 13 11 9 7 5 3 1 148  \*/  program shl**;**  const a**:=**20**,**b**:=**10**;**  var c**,**d**,**e**,**flag**;**  procedure shl2**(**k**);**  var m**;**  begin  write**(**k**);**  flag**:=**flag**+**k  end**;**  procedure shl3**(**s**);**  begin  flag**:=**flag**+**s  end**;**  procedure shl1**(**k**);**  var sum**;**  begin  flag**:=**0**;**  sum**:=**20**;**  **while** k**>=**0 **do**  begin  **if** odd k then //如果k是奇数则调用shl2，否则调用shl3  begin  call shl2**(**k**);**  sum**:=**sum**+**k  end  **else**  begin  call shl3**(**k**);**  sum**:=**sum**-**k  end**;**  k**:=**k**-**1  end**;**  flag**:=**sum**+**flag  end  begin  read**(**c**);**  e**:=**c**+**2**;**  call shl1**(**e**);**  write**(**flag**)**  end |

正确输入：

|  |
| --- |
| 5 |

结果输出：

|  |
| --- |
|  |

## 4.5测试符号类型的错误使用

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试符号类型的错误使用  错误1: 子函数中使用a，但a已经被定义为常量  错误2: sum2尚未定义  错误3: 主函数中使用a，但a已经被定义为常量  错误4: 主函数中使用sum1，但sum1是过程名  \*/  program xi**;**  const a**:=**5**;**  var j**,**sum**,**x**;**  procedure sum1**(**x**);**  var j**;**  begin  j**:=**1**;**  a**:=**1**;**  sum2**:=**1**;**  **while** j**<=**x **do**  begin  sum**:=**sum**+**j**;**  j**:=**j**+**1  end**;**  write**(**sum**)**  end  begin  read**(**a**);**  read**(**sum1**);**  write**(**a**);**  read**(**j**);**  write**(**j**);**  call sum1**(**j**+**5**)**  end |

正确输入：

|  |
| --- |
| 无 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| 错误1：子函数中使用a，但a已经被定义为常量  错误2: sum2尚未定义  错误3: 主函数中使用a，但a已经被定义为常量  错误4: 主函数中使用sum1，但sum1是过程名 |

## 4.6测试重复定义，调用函数参数不匹配，使用未定义符号

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：测试重复定义，调用函数参数不匹配，使用未定义符号  错误1: 重复定义变量x  错误2: 变量m尚未定义  错误3: 变量i尚未定义  错误4: 变量i尚未定义  错误5: 主函数调用j，但变量j只在子程序定义  错误6: 主函数调用j，但变量j只在子程序定义  错误7: sum1只要传一个参数，但是调用时传了两个参数，传参个数不匹配  错误8: 主函数调用j，但变量j只在子程序定义  \*/  program xi**;**  const a**:=**5**;**  var sum**,**x**,**x**;**  procedure sum1**(**x**);**  var j**;**  begin  m**:=**1**;**  j**:=**1**;**  i**:=**2**;**  sum**:=**0**;**  **while** j**<=**x **do**  begin  sum**:=**sum**+**j**;**  j**:=**j**+**1  end**;**  write**(**sum**)**  end**;**  procedure sum2**(**x**);**  var j**,**m**;**  begin  j**:=**1**;**  i**:=**2**;**  sum**:=**0**;**  **while** j**<=**x **do**  begin  sum**:=**sum**+**j**;**  j**:=**j**+**1  end**;**  write**(**sum**)**  end  begin  read**(**x**);**  write**(**x**);**  read**(**j**);**  call sum1**(**j**+**1**,**2**);**  write**(**j**)**  end |

正确输入：

|  |
| --- |
| 无 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| 错误1：重复定义变量x  错误2: 变量m尚未定义  错误3: 变量i尚未定义  错误4: 变量i尚未定义  错误5: 主函数调用j，但变量j只在子程序定义  错误6: 主函数调用j，但变量j只在子程序定义  错误7: sum1只要传一个参数，但是调用时传了两个参数，传参个数不匹配  错误8：主函数调用j，但变量j只在子程序定义 |

## 4.7测试中间代码优化

|  |
| --- |
| /\*  作者：181910720孙海龙  测试目的：中间代码优化  正确输入：  无输入  正确输出：  1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040  \*/  program Add**;**  const m **:=**20**;**  var a**,**b**,**c**,**d**,**e**,**f**;**  procedure add**(**h**,**q**);** //求两者之和  var sum**,return;**  begin  sum**:=**0**;**  sum**:=**q**+**h**;**  sum**:=**sum**+**1**;**  **return:=**sum**;**  write**(**h**+**q**)**  end**;**  procedure sum1**(**x**);** //求自然数前x相和  var j**,**sum**;**  begin  j**:=**1**;**  a**:=**1**;**  sum**:=**0**;**  **while** j**<=**x **do**  begin  sum**:=**sum**+**j**;**  j**:=**j**+**1  end**;**  write**(**sum**)**  end  begin  read**(**a**);**  write**(**a**);**  read**(**e**);**  write**(**e**);**  read**(**f**);**  write**(**f**);**  call sum1**(**a**);**  call add**(**e**,**f**)**  end |

正确输入：

|  |
| --- |
| 无 |

结果输出：

|  |
| --- |
|  |

输出解释：

|  |
| --- |
| （1）定义了6个变量，主函数本来应该开9个空间，但优化后只需要开6个空间  （2）同时更新a,e,f的相对地址分别为3，4，5 优化前是3，7，8 |

# 5.课设总结

通过此次编译原理课程设计任务，我觉得我完成的速度以及质量应该还是不错的，不仅对理论知识进行了完善，更是锻炼培养了自己的动手实践能力，在实践的过程中，加深了对知识的理解，将编译底层过程及实现牢牢印在了心中。同时完成了课程设计，也是让自己感受到了满满的成就感。

在教务处第一次公布考试时间为12月27号时，课设一丁点没开始的我，突然开始紧张了起来，也是从那一天的第二天开始，我就泡在了图书馆，希望能早日啃下这个硬骨头。当时虽然很累，但是这种累是完全值得的！那种忘了全世界，全天泡在图书馆，沉浸在代码世界里的感觉真的很幸福、很陶醉，一个小小的变量能让你抓破头脑，一份满意的解题成果能让你开心到起飞！在专业知识上，不仅加深了对理论知识的理解，也真正实现了对课本知识的实践。真正彻底地明白了一个基础编译器的底层实现，从客户的角度转变成了一个开发人员的角度去使用开发工具，这对我来说是一种质的飞跃。让我对编译器的理解从一个黑盒子逐渐分解成一个个模块，词法分析，语法分析，语义分析，目标代码生成，解释器，一个环节一个环节的去理解，专业知识更上了一个台阶。在课设实现的过程中，也更好地帮助我复习了课程考试的内容，我也是一个不喜欢拖延的人，所以进行了连续3天的高强度课设代码的编写，我终于实现了所有功能！

编译原理的内容是一环扣一环的，如果你在某一个地方没有理解，后面的学习中将会变得特别吃力，所以我在每次课后都会认真复习理论知识，希望在做课设的过程中能比较顺利。

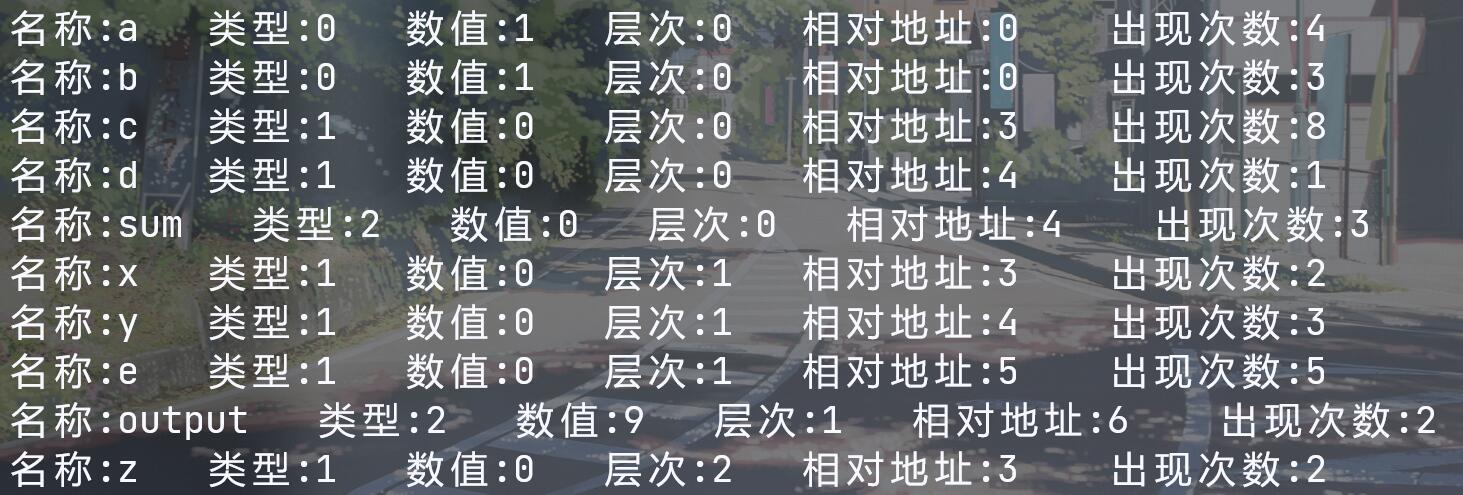
由于老师说理论知识很重要，如果理论知识没学透彻，课程设计根本做不深入，于是我每节课基本都坐在前几排，认真完成老师布置的作业，果然如老师所说。从词法分析开始，要利用识别字符的自动机转换图，这部分内容完成得相对来说比较顺利；在第二阶段语法分析时，刚开始我不太理解要怎么入手，在思考了很久以后，也在网上找了相关教程研究了一下，但是我总觉得大家的语法分析不是很完善，很多错误并没有报出来，然后我就与室友进行了交流、找往届的学长学姐指导了一下，最终知道了有些错误是不能错的，比如说必须读到const才会进入到<condecl>，这时候我才恍然大悟。在基本框架想明白以后，没过多久就很顺利地完成了语法分析，在第三次上机之前就成功完成了这部分内容。在语义分析中，我争取把每个实现细节理解清楚，实现了两种内存优化，一个是定义了未被使用的变量的空间优化，一个是在传参过程中的空间优化。

理论是实践的基础，实践是检验整理的唯一标准。在课设任务开始前，我就默默地为自己定下了一个目标，一定要拿满分！有了目标以后才会更有动力，不像有些同学一样，在学习这门课之前就有了一定的基础，我是真的从0开始，但是我坚信，从0开始也不会比任何人差，只要我足够认真，只要我足够努力，我也一样能完成得很好！

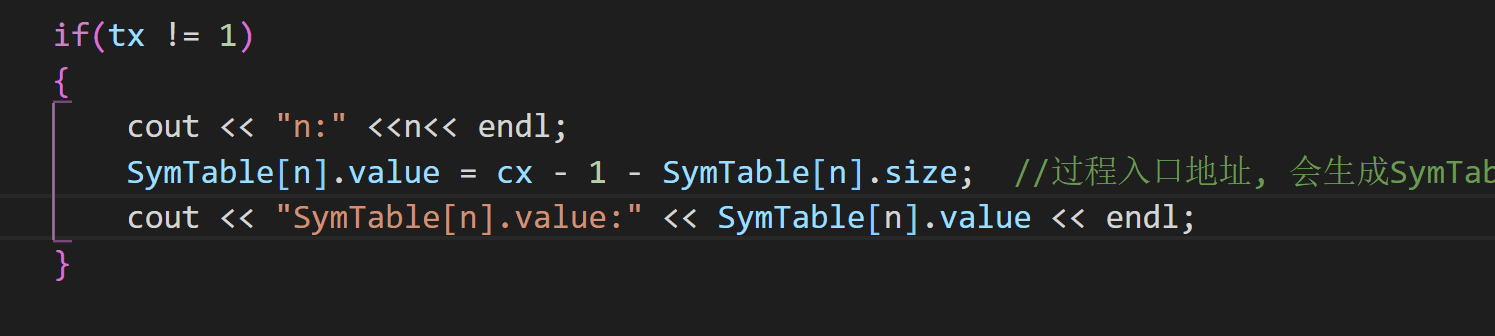
在学习方面我有一套自己的学习方法，最主要的一环就是上课一定要认真听，与其课后花双倍的时间自学还不如上课一遍听懂。在课后我也找到了一个不错的教程，MOOC里面国防科技大学的编译原理课，如果我有哪些地方不太理解我就会在网课上寻找答案，如果还是没想明白我就会主动去询问老师，直到彻底学懂这个知识点。在做课设期间，当我们遇到坎子的时候，我们应当主动和同学交流，并积极问他们有没有更好的更高效的方法，因为这样会让我们避免很多弯路。作为一名合格的程序员，一定要具有全局观，在写代码之前一定要把框架搭好，再去不断完善里面的细节，正所谓建房一定要打地基，如果地基没打好，很有可能后面的工作全是徒劳。比如说错误的统计归类，中间代码生成及解释，都应当提前做好设计，然后顺水推舟。

在课设的过程中难免会遇到让人头疼的bug，我也不例外。在第一遍的课设完成时，我编写了测试代码进行测试，并没有发现什么错误，当时if条件里面写的是-5>6，所以指令会跳转到else的部分，而我在把大于变成小于以后，就出现了奇怪的一幕，无论如何，运行的结果仍然是else部分的结果，这让我很疑惑，我就进行单步调试，一开始以为是解释OPR的时候出现了问题，但是经过一段时间的验证，这部分并没有问题，然后又经过了几个小时的检查，终于发现原来是把JPC写成了JMP，然后跳转的时候出现了问题，这种细小的bug是真的不容易查找，但是找出来的一瞬间，自己也松了一口气，毕竟作为程序员，找bug的能力也是很重要的。

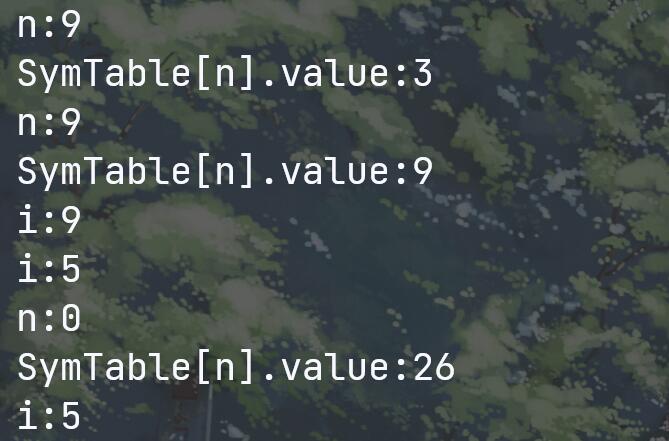
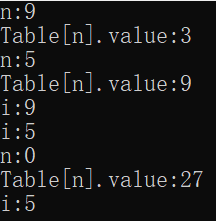
第二次遇到bug还得感谢苌艳同学，苌艳同学在21号写完了课设，这时候的我已经认为我的课设很完美了，她说要我们相互把自己写的测试代码共享测一下，然后我们交换了测试代码，在她的测试代码的函数调用中，我又发现了一个很奇怪的bug，因为我自己也写了一个斐波拉契数列的递归测试，但是在她的测试代码中，我的运行结果无论如何都是不正确，这时候我又拿出我的vscode进行了单步调试，发现竟然根本没有进入调用的函数，产生的中间代码竟然是CAL 0 0，因为我用的是符号表的value属性，跳转地址设置的是value，在对符号表进行输出的时候，我们对比了一下，如下图所示：



我的sum函数的value始终是0，而output函数的value按理来说应该是3，却变成了9。这时候我想到在写入value的地方进行输出，看看是不是写入的问题。



果然出现了问题，在findproc（）函数的返回中，我第二次调用该函数出现了不一样的地方。左图为我的输出结果，右图为苌艳同学的输出结果。

最后我发现我寻找过程的函数，出现了问题，在每一次进入procedure的时候，符号表读入了一个id，此时符号表的指针往后移了一个单位，我用tx0记录了下来(tx0=tx+1)，但是findproc()函数当中，我仍然是从tx开始寻找的，也就是我忽略了tx+1这个位置的变量，导致了错误，这时候我加上了一个传入参数，成功解决了这个bug。在这个竞争压力的社会，我们不仅要提高自己的实力，更需要学会合作，三个臭皮匠顶个诸葛亮，一个人的思维永远是不够的，要不是苌艳同学这个测试文件，我可能还以为自己的嵌套调用是没问题的。

在生活方面，写课设这几天，真的就是每天围着课设转，没有写完一个函数，没有找出一个bug，我都没有什么心情吃饭。其实作为一个将来的程序员，我们应该保持一颗积极向上的心，无论面对什么挫折，我们都要迎难而上，不应该被困难打败（但不是说要不吃饭，相反要好好吃饭保持健康），合理利用资源，实现价值最大化！

在本篇总结的最后，我想特别感谢下谢强老师，他真的真的是一门特别负责的老师，每次都很认真的批改作业，一旦有错误也会认真的检查出错误的位置。同时谢强老师也不会让每个人的分数有水分，认真做了就一定能拿到好分数，相反有些同学会去抄袭别人的代码，这样对认真完成的同学其实是很不公平的，但是谢强老师会认真的检查，提问，然后综合评价每个人，在我看来这样是很好的一个决策。同时谢强老师也十分为同学们着想，考试通知出来的第一时间，怕我们来不及复习，立马去教务处申请了修改考试时间，让我们拥有充足 的时间复习，同时以防考试时间无法调整，还做好了第二手准备，也就是调整课设时间，这么好的老师我们怎么能不好好复习，怎么能不考个高分呢！所以在后面的一段时间内，我一定会努力复习刷题，朝着100分的成绩努力，力争不留下任何遗憾。最后，再次感谢谢强老师，您辛苦了！

181910720孙海龙

2021/12/24

# 6．参考文献

[1]李文生.编译原理与技术（第二版）[M].北京:清华大学出版社,2016.

[2]陈火旺,刘春林等.程序设计语言编译原理（第3版）[M].北京:国防工业出版社,2014.

[3]PL/0语言编译器的设计与实现-博客园https://www.cnblogs.com/hf-z/p/5542070.html