|  |  |
| --- | --- |
|  |  |

**——《编译原理》实验报告——**

C0编译器的设计与实现

学    院：计算机学院（软件学院）

年    级：2019级

专    业：软件工程

班    级：软件工程二班

组    别：第三组

二〇二一年 12月 10日

目录

[一、成员 3](#_Toc91927235)

[二、题目 3](#_Toc91927236)

[2.1 C0语言的语法结构定义 3](#_Toc91927237)

[2.2 假想的栈式指令系统表 4](#_Toc91927238)

[三、分析与设计 5](#_Toc91927239)

[3.1 使用环境 5](#_Toc91927240)

[3.2 整体框架 5](#_Toc91927241)

[3.3 模块介绍 6](#_Toc91927242)

[四、测试设计 26](#_Toc91927243)

[五、问题解答 33](#_Toc91927244)

[5.1 如何解决变量的作用域问题 33](#_Toc91927245)

[5.2 如何实现函数的递归调用 33](#_Toc91927246)

[5.3 函数返回值的处理问题 34](#_Toc91927247)

[5.4 如何实现函数的先调用后定义 34](#_Toc91927248)

[六、各自的贡献 34](#_Toc91927249)

[七、心得总结 35](#_Toc91927250)

## 一、成员

|  |  |
| --- | --- |
| 姓名 | 学号 |
| 李纪元 | 0191121383 |
| 韩允杰 | 0191121355 |
| 苏日乐格 | 0181442341 |

## 二、题目

项目题目：C0编译器的设计与实现

### 2.1 C0语言的语法结构定义

<程序>->[<变量定义部分>] {<自定义函数定义部分>} <主函数>

<变量定义部分>-> int id {, id};

<自定义函数定义部分>-> ( int id | void id) '(' ')' <分程序>

<主函数>->void main'(' ')' <分程序>

<分程序>->'{' [<变量定义部分>] <语句序列> '}'

<语句序列>-><语句> {<语句>}

<语句>-> <条件语句>｜<循环语句> | '{'<语句序列>'}' | <自定义函数调用语句>

| <赋值语句> | <返回语句> | <读语句> | <写语句> | ;

<条件语句>->if '('<表达式>')' | <语句> [else <语句> ]

<循环语句>->while '(' <表达式>')' <语句>

<自定义函数调用语句>-><自定义函数调用>;

<赋值语句>->id = <表达式>;

<返回语句>->return ['(' <表达式> ')'] ;

<读语句>->scanf '(' id ')';

<写语句>->printf '(' [ <表达式>] ')';

<表达式>-> [+｜-] <项> { (+｜-) <项>}

<项> -> <因子>｛(\*｜/) <因子>｝

<因子> -> id｜'(' <表达式>')' | num | <自定义函数调用>

<自定义函数调用>->id '(' ')'

其中，id代表标识符，num代表整数，其含义及构成方式与C语言相一致；C0源程序中的变量需先定义后使用，其作用域与生存期与C语言相一致；自定义函数可超前使用（调用在前，定义在后）。

根据上面给定的C0文法及其说明和下列定义的假想栈式指令系统，按递归下降分析法设计并实现该C0语言的编译器，生成栈式目标代码；编写栈式指令系统的解释执行程序，输出目标代码的解释执行结果。

### 2.2 假想的栈式指令系统表

|  |  |
| --- | --- |
| LIT 0 a | 将常数值取到栈顶，a为常数值 |
| LOD t a | 将变量值取到栈顶，a为相对地址，t为层数 |
| STO t a | 将栈顶内容送入某变量单元中，a为相对地址，t为层数 |
| CAL 0 a | 调用函数，a为函数地址 |
| INT 0 a | 在运行栈中为被调用的过程开辟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，当栈顶值为0，则跳转至a地址，否则顺序执行 |
| ADD 0 0 | 次栈顶与栈顶相加，退两个栈元素，结果值进栈 |
| SUB 0 0 | 次栈顶减去栈顶，退两个栈元素，结果值进栈 |
| MUL 0 0 | 次栈顶乘以栈顶，退两个栈元素，结果值进栈 |
| DIV 0 0 | 次栈顶除以栈顶，退两个栈元素，结果值进栈 |
| RED 0 0 | 从命令行读入一个输入置于栈顶 |
| WRT 0 0 | 栈顶值输出至屏幕并换行 |
| RET 0 0 | 函数调用结束后,返回调用点并退栈 |

## 三、分析与设计

### 3.1 使用环境

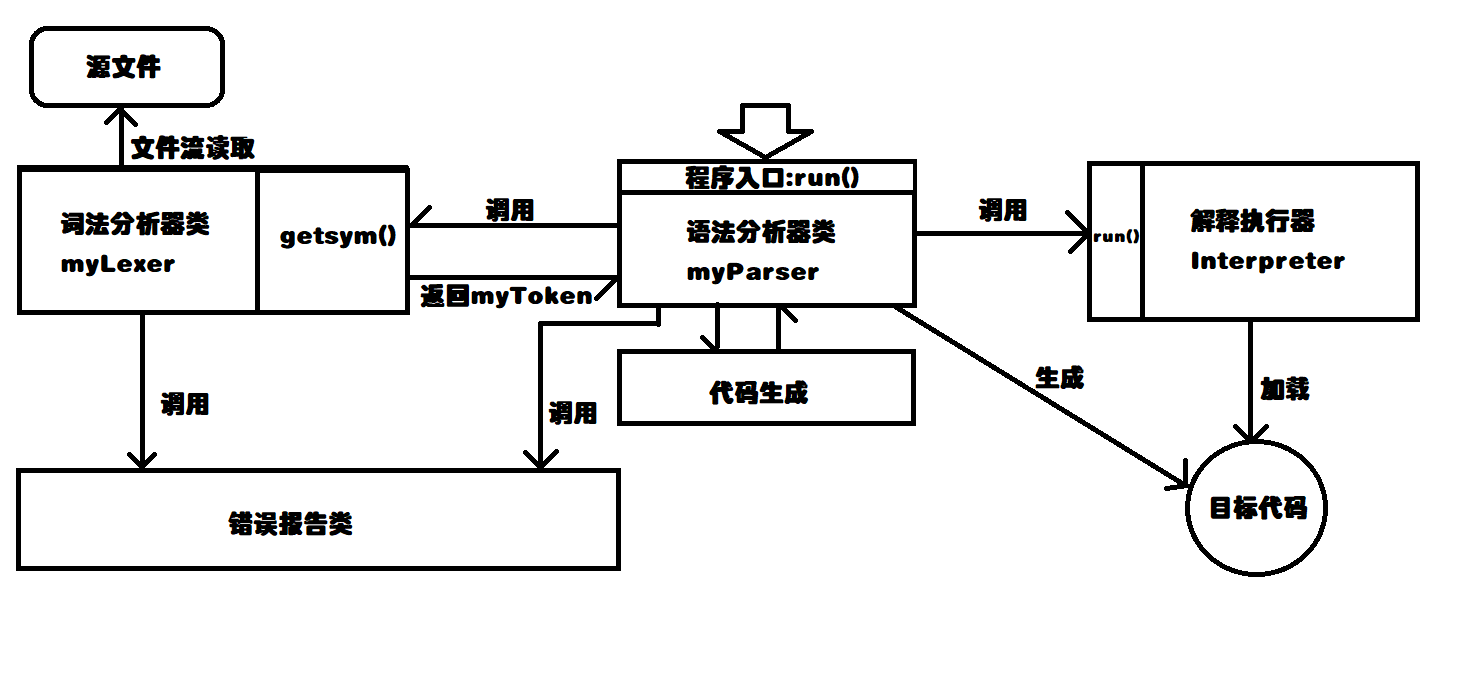
OS环境：Windows 10

编译集成环境：Microsoft Visual Studio 2019

### 3.2 整体框架

程序采用单趟遍历实现编译。

我们的整体框架可以分为三个部分：词法分析器、语法分析器、解释程序。整体框图如下：



如上图所示：词法分析器通过文件流读入别源文件，并在源文件中识别文件里的每一个词法单元。

语法分析器通过调用词法分析与语法分析的接口getsym()函数调用词法分析生成的token来进行语法分析。每次读入词法分析器返回结果，并据此识别并处理当前语句。

在语法分析每识别到一个完整的语句后，就会生成对应的目标代码。

完整的目标代码生成后，会调用run()函数将目标代码转交给解释程序进行解释与执行，得到程序的运行结果。

不可忽略的是，在上述的每一步处理中都会将其产生的错误汇报给错误报告并停止程序的运行。

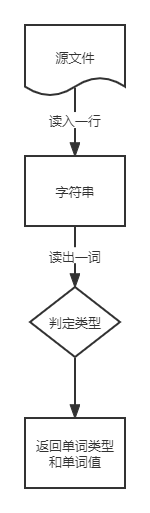
### 3.3 模块介绍

在上文中我们已经说明了C0编译器主要被分为三个部分：词法分析器、语法分析器和解释程序，根据实际情况，在不影响性能的情况下，我们将语法分析和目标代码生成合并到了一个文件myParser.cpp中，在内部进行功能划分。后又对每个功能进行划分后分别编写了若干个文件：

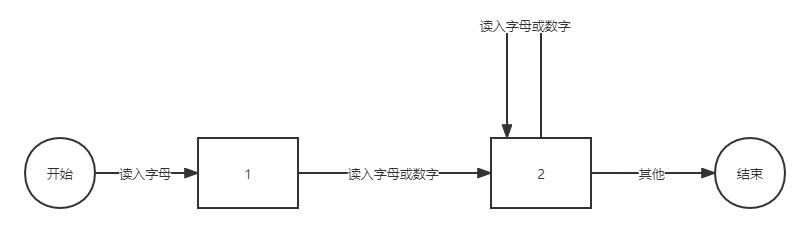
下面我们将对几部分进行介绍。

#### 3.3.1 词法分析器

在词法分析器部分，我们并没有采用lex工具自动生成词法分析器，而是自行构建了一系列的类（myLexer）和函数用于词法分析，词法分析的整体结构结构示意如下：



实现原理极为简单，就是从头开始读入单词，并识别单词及单词类型。



由上述流程图，我们给出各部分相关文件以及功能：

1）myLexer.h

定义了词法分析所用到的相关变量以及函数，如ifstream fin用于文件打开和关闭；string path用于存储路径；int curpos指向当前处理字符。

2）myLexer.cpp

里面包含了各种相关功能函数，下面对这些函数进行介绍：

|  |  |
| --- | --- |
| 函数 | 功能 |
| myLexer(string) | 词法分析类构造函数 |
| isKeyword(string) | 判断关键字 |
| isOperator(char) | 当前字符是否为运算符作出判断 |
| isDelimiter(char) | 判断界符 |
| myToken getsym() | 主体函数，判断数字、调用上述函数以及将分析结果返回 |

3）mysymTable.h

符号表类数据结构，里面存放了符号表的类型以及相应的值。如标识符ident类型的值为1。

4）mysymTable.cpp

里面包含了符号表里的相关函数：

|  |  |
| --- | --- |
| 函数 | 功能 |
| mysymTable(string) | 符号表构造函数 |
| findWord(string) | 查找符号 |
| getWord(string) | 获取符号 |
| findsymType(int) | 判断相关值所代表的单词类型 |

5）实现思想

对于程序中的每一个语法单元，我们都作如下动作：

1. 利用字符指针读入每个字符存入缓冲区，待单词读入完毕后对单词进行分类，分类成功进入该类处理程序，分类失败报错。
2. 对单词分类后，我们需要找到它在符号表中相对于的值，若在符号表中查不到该值则报错，若查到该值则以一个token的形式返回。

6）相关数据结构

词法分析所用数据结构

class myLexer {

public:

ifstream fin;//文件流用于打开关闭源文件

string path;//存储文件路径

string buffer;//存储当前分析的单词

int curpos = 0;//字符指针

int line = 0;//当前处理的行数

// 相关函数 //

myLexer(string);

myToken getsym();

int isKeyword(string);

int isOperator(char);

int isDelimiter(char);

};

符号表所用数据表结构

class mysymTable

{

public:

int nul = 0, ident = 1, number = 2, plus = 3, minus = 4, multiply = 5, divide = 6,

assign = 7, lparen = 8, rparen = 9, comma = 10, semicolon = 11, lbrace = 12,

rbrace = 13;

int elsesym = 14, ifsym = 15, intsym = 16, mainsym = 17, printfsym = 18,

returnsym = 19, scanfsym = 20,voidsym = 21, whilesym = 22;

string belong;//当前符号表属于哪个域(全局、自定义函数、main函数)

vector<myToken>words;//保存符号

mysymTable(string);

int findWord(string);//查找符号

myToken& getWord(string);//获取符号

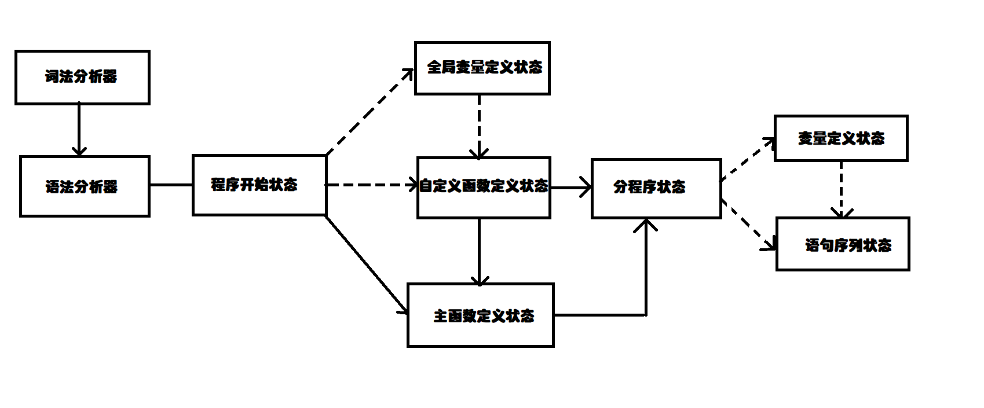
string findsymType(int);

};

#### 3.3.2 语法分析器

语法分析器的基本逻辑是：每次调用词法分析器读取一个词，然后根据读到的词与当前所处的状态转换到下一个状态，在读到一个完整的、可生生成代码的语句时，就在这时所处的状态函数中生成对应代码，并加入到对应函数的目标代码栈中。这其中还涉及到寻址、定位等问题。

我们用下面这张图来展示语法分析器的整体框架。



语法分析器的细节比较复杂，为了方便接下来的讲解，我们首先介绍一下我们的文法结构。（注：下面用e表示epsilon）

program -> int id S2 | void S3

S2 -> ( ) subprogram S4 | S5 ; S4

S5 -> , id S5 | e

S4 -> int id ( ) subprogram S4

    | void S3

S3 -> id ( ) subprogram S4 | main ( ) subprogram

vardefine -> int id S5 ;

subprogram -> { S7 sentenceseq }

S7 -> vardefine | e

sentenceseq -> sentence S8

S8 -> sentence S8 | e

sentence -> ifsentence

ifsentence -> if ( expression ) sentence S9

S9 -> else sentence | e

sentence -> whilesentence

whilesentence -> while ( expression ) sentence

sentence -> { sentenceseq }

sentence -> id S10

S10 -> ( ) ; | = expression ;

sentence -> returnsentence

sentence -> scanfsentence

sentence -> printfsentence

sentence -> ;

returnsentence -> return S11;

S11 -> ( expression ) | e

scanfsentence -> scanf ( id ) ;

printfsentence -> printf ( S12 ) ;

S12 -> expression | e

expression -> S13 item S14

S13 -> + | - | e

S14 -> + item S14|- item S14 | e

item -> factor S16

S16 -> \* factor

S16 -> / factor

S16 -> e

factor -> ( expression )

factor -> num

factor -> id S17

S17 -> ( ) | e

该文法是LL(1)文法，我们可以求出其FIRST集和FOLLOW集。

|  |  |  |
| --- | --- | --- |
| 非终结符 | FIRST集 | FOLLOW集 |
| S17 | ( e | \* / + - if while return scanf printf { id ; ) |
| factor | ( num id | \* / + - if while return scanf printf { id ; ) |
| S16 | \* / | if while return scanf printf  { id ; + - ) |
| item | ( num id | if while return scanf printf  { id ; + - ) |
| S14 | + - e | if while return scanf printf  { id ; ) ; |
| S13 | + - e | ( num id |
| expression | + - ( num id | ; ) |
| S12 | + - ( num id e | ) |
| printfsentence | printf | if while return scanf printf  { id ; else |
| scanfsentence | scanf | if while return scanf printf  { id ; else |
| S11 | ( e | ; |
| returnsentence | return | if while return scanf printf  { id ; else |
| whilesentence | while | if while return scanf printf  { id ; else |
| ifsentence | if | if while return scanf printf  { id ; else |
| S10 | ( = | if while return scanf printf  { id ; else |
| S9 | else e | if while return scanf printf  { id ; else |
| sentence | if while return scanf printf  { id ; | if while return scanf printf  { id ; else |
| S8 | if while return scanf printf  { id ; e | } |
| sentenceseq | if while return scanf printf  { id ; | } |
| vardefine | int | if while return scanf printf  ; id { |
| S7 | int e | if while return scanf printf  ; id { |
| subprogram | { | int void $ |
| S3 | id main | $ |
| S4 | int void e | $ |
| S5 | , e | ; |
| S2 | , e ( ; | $ |
| program | int void | $ |

我们将文法中的每一个状态作为语法分析器的一个函数，状态间的相互转换就是函数间的相互调用过程。下面，我们介绍一下语法分析器各个类的接口。

1）func类

该类用于保存关于函数的各种信息。

成员变量：

|  |  |
| --- | --- |
| string name | 函数名 |
| int willreturn | 声明中是否有返回值 |
| int hasreturn | 代码中是否实际有返回值，语法分析结束后才能确定 |
| int varcnt | 变量个数 |
| mysymTable funcTable | 当前函数变量表，使用定义好的mysymTable类型 |
| commandSet commandset | 当前函数指令集合，使用定义好的commandSet类型，每条指令的格式都是“操作，一地址，二地址” |
| loc | 函数在指令栈中的地址 |

2）myParser类

成员变量：

|  |  |
| --- | --- |
| myLexer lexer | 词法分析器 |
| commandSet mainSet | main函数指令集合，同时负责为全局变量开辟空间 |
| mysymTable globalTable | 保存全局变量 |
| mysymTable mainTable | 保存main函数中的变量 |
| myToken cur | 当前的输入符号 |
| vector<func>funcs | 函数集合 |
| int p | 是否打印调试信息 |
| int mainreturn | main函数中是否读到return |
| vector<command>commandRes | 最终的指令栈 |

成员函数：

|  |  |
| --- | --- |
| myParser(string s) | 构造函数，使用文件路径进行初始化 |
| void run(); | 程序接口，从这个函数开始执行 |
| int findFunc(string); | 根据函数名查找函数是否出现过 |
| func& getFunc(string); | 根据函数名得到函数的引用 |
| int program(); | program状态，语法分析的开始状态 |
| int S2(myToken); | 中间状态S2 |
| int S3(); | 中间状态S3 |
| int S5(string); | 中间状态S5 |
| int S4(); | 中间状态S4 |
| int subprogram(string); | subprogram状态，负责处理分程序 |
| int vardefine(string); | vardefine状态，负责处理变量定义 |
| int sentenceseq(string); | sentenceseq状态，负责处理语句序列 |
| void printfsinglesymTable(string,mysymTable&); | 打印指定函数的变量表 |
| void printsymTable(); | 打印所有的变量表 |
| int sentence(string); | sentence状态，处理语句 |
| void printsinglecommandstk(string, commandSet&); | 打印指定的指令栈 |
| void printfcommandstk(); | 打印所有指令栈 |
| int ifsentence(string); | ifsentence状态，处理if语句 |
| int expression(string); | expression状态，处理表达式 |
| int lodIdent(commandSet&, mysymTable&, string,string); | 为指定的指令栈生成将标识符取到栈顶的命令 |
| int whilesentence(string); | whilesentence状态，处理while语句 |
| int returnsentence(string); | returnsentence状态，处理return语句 |
| int returnRight(func&, commandSet&); | 判断函数的返回值是否和声明一致 |
| int scanfsentence(string); | scanfsentence状态，处理scanf语句 |
| int printfsentence(string); | printfsentence状态，处理printf语句 |
| int stoIdent(commandSet&, mysymTable&, string, string); | 为指定的指令栈生成将栈顶值存到标识符的命令 |

语法分析器的每个状态函数都有类似的结构，它们是这样的：

读取下一个词法单元cur;

（有的不用读，因为在上一个状态可能读过了）

if(cur属于SELECT(下一个状态))

{

    执行一些动作;

    if(下一个状态()==1)//返回值为1表示状态分析无误

    {

        执行一些动作;

    }

    else

    {

        报错;

    }

}

else if(...)//其他状态

{

    ...

}

else{

    报错;

}

依据具体状态的不同，会有许多处理上的细节问题，我们这里仅就其中几个关键、复杂的状态展开解释，一些比较直观简单的状态比如输入输出语句处理状态等我们就不赘述了。

1. 变量定义

if(读到标识符)

{

    if(标识符不在当前域变量表中){

        加入当前域的变量表;

    }

    else

    {

        报错;

    }

}

else if(读到逗号)

{

    递归处理下一个标识符;

}

else if(读到分号)

{

    生成INT指令并插入当前域指令栈;

}

else{

    报错;

}

注意在实际的代码中，全局变量和函数变量的处理不在同一个状态中，它们各自也没有只是用一个状态，而是共用了一个中间状态S5，但是它们的基本逻辑都是和伪代码一致的。

1. 表达式处理

逐个读入表达式的词法单元，同时进行括号匹配：

    if(读到分号)表达式结束;

    else if(读到无法匹配的右括号)表达式结束;

    else if(读到-)

    {

        if(前面没有符号或者是左括号或者是运算符)

        {

            当前的-是负号，转化为#;

        }

        else{

            当前的-是减号，直接读入;

        }

    }

    else if(读到数字)

    {

        if(前一个词是负号)

        {

            前一个词弹出，数字值取负读入;

        }

        else{

            直接读入;

        }

    }

    else if(读到标识符)

    {

        if(前一个词是负号)

        {

            转化为 0-标识符 然后读入;

        }

        else{

            直接读入;

        }

    }

    else{

        报错;

    }

此时表达式完成了读入，重新扫描一遍表达式，对所有 标识符() 特殊标记为函数调用;

如果表达式中只有一个量，直接生成对应目标代码然后return即可;

接下来我们将中缀表达式转为后缀表达式:

定义操作符栈，变量栈，结果表达式;

从头到尾扫描原表达式:

    if(遇到变量、函数、数字)直接入变量栈;

    else if(遇到左括号)直接入操作符栈;

    else if(遇到右括号)

    {

        不断弹出操作符栈符号并加入结果表达式，直到遇到左括号;

    }

    else if(遇到运算符)

    {

        若当前运算符优先级小于等于操作符栈顶运算符优先级，不断将栈顶运算符弹出并加入结果表达式;

        将当前运算符压入操作符栈;

    }

接下来只需要对后缀表达式顺序生成目标代码即可，比如遇到变量就生成LOD语句，遇到函数就生成CAL语句，遇到运算符就生成对应运算符语句等等;

1. if语句处理

读到if;

读到左括号;

处理表达式;

读到右括号;

生成JPC指令，暂时不设置目的地址;

处理语句;

if(读到else)

{

    生成JMP指令，这个指令是用于if分支执行完后不要错误地顺序执行else分支;

    处理语句;

    为JPC指令和JMP指令设置地址;

}

else

{

    直接为JPC指令设置地址;

}

4）while语句处理

读到while;

读到左括号;

处理表达式;

读到右括号;

生成JPC指令，暂时不设置目的地址;

处理语句;

生成JMP语句，用于跳回while的开头;

为JPC语句和JMP语句设置地址;

5）函数返回值判断

if (函数的hasreturn等于函数的willreturn)

{

    return 1;

}

else

{

    if (函数的hasreturn == -1)//没有return语句

    {

        if (函数的willreturn == 0)

        {

            生成一条RET指令;

            return 1;

        }

        else

        {

            return 0;

        }

    }

    else

    {

        return 0;

    }

}

这部分用于判断函数的返回值是否和声明的一致。

6）LOD语句生成

if(在当前函数的变量表中找到变量)

{

    取得变量引用;

    利用变量偏移量生成LOD语句;

}

else if(在全局函数变量表中找到变量)

{

    取得变量引用;

    利用变量偏移量生成LOD语句;

}

else{

    变量未声明，报错;

}

STO语句和LOD语句的逻辑几乎是一样的。

7）寻址问题

在语法分析、代码生成结束后，我们得到了每个函数的指令栈

但此时，我们仍有两项工作需要做

1.将所有函数的指令栈连起来，同时指定每个函数的地址，并据此转换其指令栈内所有的相对地址

2.根据上一步确定的函数地址，为所有函数调用语句确定地址。

生成JMP语句，暂不指定地址，并加入结果指令栈;//最开始跳转到main的语句，之后确定main的地址后会修改其地址

cnt指针指向1;

for(函数集合中所有函数){

    函数地址等于cnt;

    for(当前函数指令栈中所有指令)

    {

            command& c = funcs[i].commandset.stk[j];

        if (当前指令是JMP或JPC)

        {

            指令二地址加上cnt;//从相对地址变为绝对地址

        }

        将指令压入结果指令栈;

    }

    cnt加上当前函数指令栈指令数;

}

结果指令栈的第一条指令二地址指向现在的cnt;

对main函数重复一遍上面的函数处理过程;

for (结果指令栈的每条指令)

{

    if (当前指令是CAL指令)

    {

        扫描函数集合并找到CAL指令记录的函数，将其二地址设置为函数的地址;

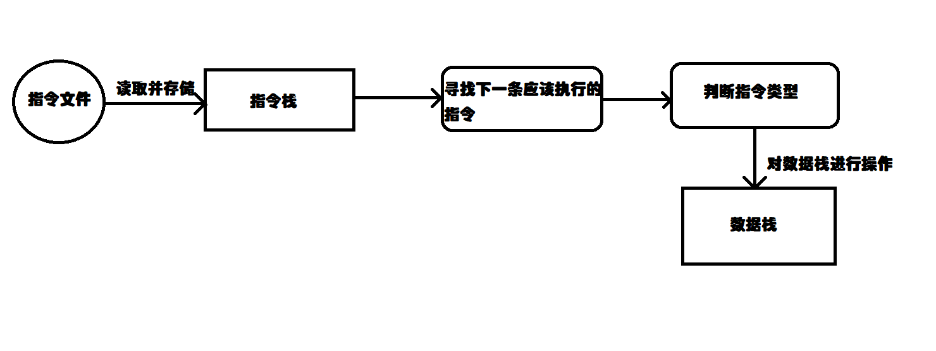
        如果没找到说明函数被调用了却没有定义，报错;

    }

}

#### 3.3.3 解释执行程序

解释执行程序的整体逻辑如下图所示。



解释执行各个类的接口定义如下：

1）commandStack类

成员变量：

|  |  |
| --- | --- |
| int ne | 指向下一条要执行的指令 |
| int outputmessage | 是否打印调试信息 |
| vector<tuple<int, string, int, int> >c\_stk | 保存指令，格式为“标号，操作符，一地址，二地址” |
| DataStack d\_stk | 数据栈 |

成员函数：

|  |  |
| --- | --- |
| void run(); | 程序入口，从这里开始执行 |
| void readCommand(); | 读取指令文件 |
| void findNextCommand(int&); | 寻找下一条要执行的指令 |

2）DataStack类

成员变量：

|  |  |
| --- | --- |
| vector <tuple<int, int, int> >stk | 数据栈主体，每个单元格式为“所在层，偏移量，值” |
| vector<int>base | base[i]表示第i层的基地址 |
| int cur\_layer | 当前所在层 |
| int backpoint | 当前函数返回点 |
| int top | 指向栈顶的指针 |

成员函数：

|  |  |
| --- | --- |
| void show() | 打印信息 |
| tuple<int, int, int>getTop(); | 取得栈顶单元 |
| int getCuurentLayer(); | 取得当前层 |
| int getCurrentBase(); | 取得当前层的基地址 |
| int getCurrentBaseValue(); | 取得当前层基地址中的值 |
| void pushTop(int); | 将值压入栈顶 |
| void pushTop(int, int); | 将带偏移量的值压入栈顶 |
| void popTop(); | 弹出栈顶 |
| tuple<int, int, int> getValue(int, int); | 如果层数为0，取得第0层指定偏移量单元；如果层数为1，取得当前函数所在层指定偏移量单元格 |
| void setValue(int, int, int); | 与上一个函数类似，不过该函数用于设置指定单元格的值 |
| void CallFunc(int); | 执行CAL语句时，需要开辟空间以及保存返回点 |
| void LeaveFunc(); | 离开函数时的善后处理 |

以下是解释执行器主要过程的伪代码描述：

读入指令文件;

while(可以找到下一条指令)

{

    找到下一条指令;

    执行当前指令;

}

其中，大多数步骤只是单纯的按序模拟，不需要复杂的逻辑；只有一部分涉及到地址跳转的逻辑需要进一步解释。

寻找下一条需要执行的指令，在不涉及跳转时，我们只需要单纯地递增指令指针即可；如果涉及到跳转时，根据指令操作不同，我们有不同的寻址方式，其伪代码如下：

if(当前指令为CAL或JMP)

{

    下一条指令地址是当前指令语句的二地址;

}

else if(当前指令为JPC)

{

    if(栈顶值为0)

    {

        下一条指令是当前指令语句的二地址;

    }

    else

    {

        递增指令指针;

    }

}

else if(当前指令为RET)

{

    if(当前在第0层)

    {

        指令指针置-1;

        return;

    }

    else

    {

        指令指针置当前层基址单元值（即返回地址）;

    }

}

else{

    递增指令指针;

}

其他指令就不再赘述了，都比较简单，唯一需要补充的有三点：

1.DIV指令需要判断除数是否为0；

2.调用CAL语句需要将返回地址保存在函数所在层基址以便调用结束后返回；

3.我们将第0层第3个单元格（偏移量为2）作为函数的返回值单元，所有函数在返回时如果有返回值都会存到这里再退栈，等我们回到调用方就可以从这里取出返回值使用。

更多的细节请查看源码。

# 四、测试设计

**测试文件1：**

int base,power,ans;

int quick\_pow()

{

    int res;

    res=1;

    while(power)

    {

        if(power-power/2\*2)

        {

                res=res\*base;

        }

        power=power/2;

        base=base\*base;

    }

    return res;

}

void main()

{

    scanf(base);

    scanf(power);

    ans=quick\_pow();

    printf(ans);

}

这是一个快速幂的实现程序，由于完整的调试信息过长，我们仅展示生成的指令栈：

JMP 0 30

INT 0 3

LIT 0 1

STO 1 2

LOD 0 4

JPC 0 27

LOD 0 4

LOD 0 4

LIT 0 2

DIV 0 0

LIT 0 2

MUL 0 0

SUB 0 0

JPC 0 18

LOD 1 2

LOD 0 3

MUL 0 0

STO 1 2

LOD 0 4

LIT 0 2

DIV 0 0

STO 0 4

LOD 0 3

LOD 0 3

MUL 0 0

STO 0 3

JMP 0 4

LOD 1 2

STO 0 2

RET 0 0

INT 0 6

RED 0 0

STO 0 3

RED 0 0

STO 0 4

CAL 0 1

LOD 0 2

STO 0 5

LOD 0 5

WRT 0 0

RET 0 0

我们输入了2 3（计算2的3次方）以及3 3（计算3的3次方），结果分别是8和27。

**测试文件2：**

int n,i,s1;

int sum()

{

    int s;

    s=0;

    while(n)

    {

        s=s+n;

        n=n-1;

    }

    return s;

}

void main()

{

    s1=0;

    scanf(i);

    if(i)

    {

        scanf(n);

        s1=sum();

    }

    else

    {

        s1=1;

        s1=s1\*6;

        s1=s1/3;

    }

    printf(s1);

}

生成的栈式指令：

JMP 0 18

INT 0 3

LIT 0 0

STO 1 2

LOD 0 3

JPC 0 15

LOD 1 2

LOD 0 3

ADD 0 0

STO 1 2

LOD 0 3

LIT 0 1

SUB 0 0

STO 0 3

JMP 0 4

LOD 1 2

STO 0 2

RET 0 0

INT 0 6

LIT 0 0

STO 0 5

RED 0 0

STO 0 4

LOD 0 4

JPC 0 31

RED 0 0

STO 0 3

CAL 0 1

LOD 0 2

STO 0 5

JMP 0 41

LIT 0 1

STO 0 5

LOD 0 5

LIT 0 6

MUL 0 0

STO 0 5

LOD 0 5

LIT 0 3

DIV 0 0

STO 0 5

LOD 0 5

WRT 0 0

RET 0 0

我们输入了1 3（计算3+2+1），结果为6；输入了0（直接输出2），结果为2。

**测试程序3：**

int n,a,b,c,d;

int f()

{

    c=a+b;

    return c;

}

void main()

{

    a=1;

    b=1;

    scanf(n);

    printf(a);

    printf(b);

    while(n)

    {

        d=f();

        a=b;

        b=d;

        printf(d);

        n=n-1;

    }

}

生成的栈式指令：

JMP 0 9

INT 0 2

LOD 0 4

LOD 0 5

ADD 0 0

STO 0 6

LOD 0 6

STO 0 2

RET 0 0

INT 0 8

LIT 0 1

STO 0 4

LIT 0 1

STO 0 5

RED 0 0

STO 0 3

LOD 0 4

WRT 0 0

LOD 0 5

WRT 0 0

LOD 0 3

JPC 0 36

CAL 0 1

LOD 0 2

STO 0 7

LOD 0 5

STO 0 4

LOD 0 7

STO 0 5

LOD 0 7

WRT 0 0

LOD 0 3

LIT 0 1

SUB 0 0

STO 0 3

JMP 0 20

RET 0 0

我们输入了5（输出前2+5项斐波那契数列），结果为1 1 2 3 5 8 13。

另外，我们还设计了许多其他的关于错误处理的程序，比如返回值不一致、程序结构错误、各种语法错误、函数重定义、函数未定义、标识符未定义等，这里限于篇幅，不再一一给出它们的测试结果。

## 五、问题解答

### 5.1 如何解决变量的作用域问题

在使用标识符时，如果存在多个同名的标识符，那么我们应该优先使用当前定义域中的标识符，如果不存在，再去外层标识符寻找，这也就是作用域的嵌套原则。

那么如何实现这一原则呢？我们在这里为每个函数和全局变量各设置了一个符号表，在遇到一个变量时，首先会去当前函数的符号表寻找变量，如果找不到，再去全局变量的符号表寻找变量。

### 5.2 如何实现函数的递归调用

在之前书写栈式指令时，我们曾使用调用层数来标识变量的位置，但实际上这是一个错误的考虑，因为一个函数的调用层数只有在运行时才能得到，是无法事先确定的。如果试图用事先确定的方式规定层数，那么将无法实现函数的递归调用。

不难注意到，我们的程序中每个函数只会使用到两层的变量：全局变量和函数内部的变量，所以不使用层数，即不令LOD、STO等栈式指令可以随意使用任何层的变量，而是只能使用0层和当前层的变量，便可以规避我们的问题。

### 5.3 函数返回值的处理问题

函数的返回语句有三种形式：写了return，但是没有返回值；写了return，且有返回值；没有写return。

如何确定函数声明的返回值类型和实际的返回值类型是统一的呢？这里有两种解决方式：一种是从文法上，为返回值为void和int的函数分别规定文法；另一种是在程序里判断函数的返回值类型是否一致。

由于这个问题是比较后期时注意到的，修改文法的代价太大，所以我们使用了第二种方式，为每个函数设置了两个标志位，分别标识其声明的返回类型和实际的返回类型，通过两个标志位来判断返回值是否一致。

### 5.4 如何实现函数的先调用后定义

在生成所有CAL语句时，我们不立刻生成函数的地址，而是先把所调用的函数的名字保存在当前的命令对象中，等到所有目标代码生成结束，所有函数的地址得到确定后，我们再根据CAL指令保存的函数名一一去寻找地址。

## 六、各自的贡献

苏日乐格完成了词法分析器、词法单元、符号表等的设计、实现；解释程序的接口设计与定义；

李纪元完成了文法的设计、转化，语法分析器的设计与实现，解释程序的指令栈、主函数实现；

韩允杰完成了目标代码生成功能、错误报告模块的设计与实现，解释程序的数据栈实现，并承担了绝大部分的差错与测试工作。

文档由三人共同编写，贡献均等。

## 七、心得总结

1.写代码之前一定要把设计想好，这次开发过程中有许多次是由于之前设计得不构完善或兼容导致回头修改从前的代码，十分痛苦。

2.一定要注意分类的清晰，这次的程序中有许多分类处理的例子，如果想得不清楚，就可能漏掉某些情况。

3.测试工作要逐步进行，每开发一部分就测试一部分。

4.勤写注释，尤其是那些设计时没考量充分的部分，一些冗余、缺失、或者与其他架构不同的实现的细节都需要记录下来，这样回头再看的时候才不会迷惑。