

基于递归下降的 C0 文法编译器

姓名：陈豪博

学号：2016302862

班号：10011604

时间：2019-07-01

计算机学院

目录

- 前言3
- 文法定义.....4
- 文法说明.....6
- 系统总体设计.....14
 - 系统结构.....14
 - 模块说明.....15
- 详细设计.....16
 - 简单文法的递归下降实现.....16
 - 表达式文法的实现.....18
 - 常量和变量的引入.....20
 - 语句的使用与简化版程序的诞生.....23
 - 函数的出现与符号表的一次改革.....25
 - 前端工作的初步完成.....26
 - 堆栈体系的理解与设计.....27
 - 寄存器分配算法的历史性翻车.....29
 - 真香定律之简单汇编代码生成.....31
 - 中间代码优化与符号表的二次改革.....32
 - 拒绝真香定律之寄存器分配算法.....37
- 测试40
 - 系统测试之优化前的测试.....40
 - 系统测试之优化后的测试.....41
- 问题总结.....45
 - 难点一：函数的实现.....45
 - 难点二：变量的位置.....46
 - 难点三：参数的位置和数组的计算.....46
- 附录47
 - 符号表的第一次改革.....47
 - 前端测试结果演示.....48

前言

本项目采用增量模型实现了基于递归下降的 C0 文法编译器开发，包括前端和后端两个部分，由文法的递归下降解析、词法分析器、语法分析器、错误处理、中间代码生成（四元式）、中间代码优化、MIPS 汇编代码生成、寄存器分配等过程组成。项目的文法与原始 C0 文法有所出入，是 C0 文法的子集，但足以支持面向过程程序的编译——顺序、选择、循环的编译均可实现，因此后面提到的 C0 文法均为本项目的文法，而非原始 C0 文法。编译器可以完成 C0 文法的完整编译并最终生成被优化过的 MIPS 汇编代码，并且具有错误处理功能，使用恐慌模式的处理方法，遗憾的是，由于时间关系，错误提示信息仅能报告错误内容，对错误位置的位置信息不能反映。开发过程中测试与开发并行，完成一个子功能进行一次测试，渐增式的测试保证了各个功能模块之间的正确调用，黑盒测试的方法搭配 Mars 模拟器对汇编代码的校验确保了编译器的正确运行。全项目代码量总计 4457 行，包括空行和注释，开发环境为 windows10/Dev C++。

文法定义

<加法运算符> ::= + | -

<乘法运算符> ::= * | /

<关系运算符> ::= < | <= | > | >= | != | ==

<字母> ::= _ | a | ... | z | A | ... | Z

<数字> ::= 0 | <非零数字>

<非零数字> ::= 1 | ... | 9

<字符> ::= '<加法运算符>' | '<乘法运算符>' | '<字母>' | '<数字>'

<字符串> ::= " {十进制编码为 32,33,35-126 的 ASCII 字符} "

<程序> ::= [<常量说明>] [<变量说明>] { <有返回值函数定义> | <无返回值函数定义> } <主函数>

<常量说明> ::= const <常量定义>; { const <常量定义>; }

<常量定义> ::= int <标识符> = <整数> { , <标识符> = <整数> }
| char <标识符> = <字符> { , <标识符> = <字符> }

<无符号整数> ::= <非零数字> { <数字> }

<整数> ::= [+ | -] <无符号整数> | 0

<标识符> ::= <字母> { <字母> | <数字> }

<声明头部> ::= int <标识符> | char <标识符>

<变量说明> ::= <变量定义>; { <变量定义>; }

<变量定义> ::= <类型标识符> (<标识符> | <标识符> '[' <无符号整数> ']') { , (<标识符> | <标识符> '[' <无符号整数> ']') } // 不许在定义变量时赋初值.

<常量> ::= <整数> | <字符>

<类型标识符> ::= int | char

<有返回值函数定义> ::= <声明头部> '(' <参数> ')' '{' <复合语句> '}'

<无返回值函数定义> ::= void <标识符> '(' <参数> ')' '{' <复合语句> '}'

<复合语句> ::= [<常量说明>] [<变量说明>] <语句列>
 <参数> ::= <参数表>
 <参数表> ::= <类型标识符> <标识符> { , <类型标识符> <标识符> } |
 <空>
 <主函数> ::= void main('(',')' '{' <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> | <标识符> '[' <表达式> ']' | <整数> | <字符>
 > | <有返回值函数调用语句> | '(' <表达式> ')'
 <语句> ::= <条件语句> | <循环语句> | '{' <语句列> '}' | <有返回值
 函数调用语句>; | <无返回值函数调用语句>; | <赋值语句>; | <空>; | <返
 回语句>;
 <赋值语句> ::= <标识符> = <表达式> | <标识符> '[' <表达式> ']' = <
 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> //表达
 式为0 条件为假，否则为真
 <循环语句> ::= while '(' <条件> ')' <语句>
 <有返回值函数调用语句> ::= <标识符> '(' <值参数表> ')'
 <无返回值函数调用语句> ::= <标识符> '(' <值参数表> ')'
 <值参数表> ::= <表达式> { , <表达式> } | <空>
 <语句列> ::= { <语句> }
 <返回语句> ::= return '(' <表达式> ')']

文法说明

<加法运算符> ::= + | -

/*加法运算符由+号、-号组成*/

<乘法运算符> ::= * | /

/*乘法运算符由*号、/号组成*/

<关系运算符> ::= < | <= | > | >= | != | ==

/*关系运算符由<、<=、>、>=、!=、==组成*/

<字母> ::= _ | a | . . . | z | A | . . . | Z

/*字母由下划线与大小写字母组成*/

<数字> ::= 0 | <非零数字>

/*数字由 0、<非零数字组成>*/ 举例：0|1...|9，记为 **number**

范例：0

分析：数字包含了所有的单个的数字，不同于整数

<非零数字> ::= 1 | . . . | 9

/*非零数字由 1~9 组成*/

分析：非零数字也是单个的数字，但是不含 0。

<字符> ::= '<加法运算符>' | '<乘法运算符>' | '<字母>' | '<数字>'

/*字符包含加法运算符、乘法运算符、字母、数字？记为 **character***/

范例：'+','-','*','/'等单字符

分析：必须添加单引号，必须是单个的！注意和字符串的区分

<字符串> ::= " {十进制编码为 32,33,35-126 的 ASCII 字符} "

/*规定合法的字符串，必须要用双引号括起来，记为 **chstring***/

范例：“abc”

<程序> ::= [<常量说明>] [<变量说明>] { <有返回值函数定义> | <无返回值函数定义> } <主函数>

/*程序组成部分包含常量说明、变量说明、有返回值函数定义、无返回值函数定义、主函数，主函数过后整个程序结束*/

范例以及说明： **const int a=1,b=1;** /*常量说明部分

```
const char c='a';  /*
```

```
int a_1;    /*变量说明部分
```

```
char c_1;  /* /*在这里定义和说明的常量变量都是全局的*/
```

```
int fun1(int a, int b){
```

```
    XXXX
```

```
    return 1;/*有返回值的函数定义必须要有返回值*/
```

```
}
```

```
void fun2() { /*无返回值的函数定义*/
```

```
    return;
```

```
}
```

```
Void main () { /**/
```

```
XXXXXX
```

Return ;//虽然从文法上可以推导出 **void** 可以返回 **return 0**，不过语法上应该检查对不对

}

分析：不能随便改变顺序，例如把函数的定义放在全局变量或者全局常量的前面的做法是不对的。常量说明，变量说明以及函数定义都可有可无，但是主函数是必要的。

<常量说明> ::= **const**<常量定义>;{ **const**<常量定义>;}

/*常量说明的标志是 **const** *， **const int a1=1***/

分析：可以一个可以多个，不同类型的 **const** 中间用分号隔开。

<常量定义> ::= **int**<标识符>=<整数>{,<标识符>=<整数>}
| **char**<标识符>=<字符>{,<标识符>=<字符>}

/*常量定义为 **int a1=12** 或者 **char c='a'**，允许连续定义，如 **int a=1, b=2***/

<无符号整数> ::= <非零数字> {<数字>}

/*无符号整数举例：10,11,12,13,14，不能以 0 开头，前面不能有正负号，不能为 0， **unsigned integer***/

<整数> ::= [**+** | **-**] <无符号整数> | 0

/*整数在无符号整数前添加正负号，也可以不带+/-，也包含了 0，记为 **integer** /

<标识符> ::= <字母> {<字母> | <数字>}

/*标示符由字母开头，可以纯字母，可以加上数字编号，记为 **ident***/

<声明头部> ::= int<标识符> | char<标识符>

/*函数定义的声明头部为 **int a** 开头，但是因为是函数定义所以紧跟在后的是括号（和变量说明部分开始一样，但是要注意区分！）*/

<变量说明> ::= <变量定义>;{<变量定义>;}

/*变量说明由一个或多个变量定义组成，多个变量定义之间由分号隔开*/

<变量定义> ::= <类型标识符>(<标识符>|<标识符>['<无符号整数>']){(<标识符>|<标识符>['<无符号整数>'])}

举例： **int a1,ax[2],ax3,ax[4]**

分析：允许多个，必须要有标识符。

<常量> ::= <整数>| <字符>

/*常量为整数或者字符*/

举例： **112, 'a'**。

<类型标识符> ::= int | char

/*类型标识符为 **int** 或者 **char**，用在变量定义的开头*/

<有返回值函数定义> ::= <声明头部>('(<参数>'){'<复合语句>'}

/*有返回值的函数定义格式为声明头部(参数){复合语句}，举例 **int function(int a,int b) {xxxxx}***/

分析：有返回值的函数定义，**return** 的时候必须要有值，因为有值，所以有返回值的函数在被调用的时候也可以作为因子。

<无返回值函数定义> ::= void<标识符>‘(’<参数>‘)’‘{’<复合语句>‘}’

/*无返回值的函数定义格式为声明头部(参数){复合语句}, 举例 **void function(int a,int b) {xxxxx}*/**

分析：**void** 函数无返回值，但是从语法上来讲不会报错，应该从语义分析上进行报错。

<复合语句> ::= [<常量说明>] [<变量说明>] <语句列>

/*复合语句包含 0~多个常量说明, 0~多个变量说明, 语句列, 举例: **const int a=1, int b, XXXXX*/**

分析：其实每一个符合语句就是平时写的程序里面的函数过后的{}内部的内容，这样一说是不是就好理解得多了呢？

<参数> ::= <参数表>

/*参数即为参数表*/

<参数表> ::= <类型标识符><标识符>{,<类型标识符><标识符>}|
<空>

/*参数表举例: **int a,int b*/**

或者就是空，什么都没有

分析：主要用在涉及到函数的声明定义以及调用传参里面(所以可以为空)

<主函数> ::= void main‘(’‘)’ ‘{’<复合语句>‘}’

/***void main() {XXXX}*/**

XXXX 即为复合语句

分析：主函数是程序必须的元素，整个程序只允许出现一个主函数，并且，如果主函数分析完毕，则整个编译程序分析完毕

<表达式> ::= [+ | -] <项> {<加法运算符> <项> }

/*表达式由项（一个项或者多个项）加上加法运算符组成，前面可以添加正负号*/

范例： **a+b, a*b+c*d, a+ (b+c)**

分析：每一个表达式其实是有值的，可以用来进行赋值或者进行条件判断

<项> ::= <因子> {<乘法运算符> <因子> }

/*项由因子（一个或者多个）乘除运算符组成*/

范例： **a*b, a, (a+b)*c ,a/c**

分析：因子与因子之间的链接符号只能为乘法运算符*或者/

<因子> ::= <标识符> | <标识符> '[' <表达式> ']' | <整数> | <字符> > | <有返回值函数调用语句> | '(' <表达式> ')'

/*因子包含的内容有：indent, indent[表达式], 123 等整数, 'c'等字符, call(a), (a+b)

*/

分析：以上的距离分别对应标识符、<标识符> '[' <表达式> ']'、<整数>、<字符>、call(a)、(' <表达式> ')

<语句> ::= <条件语句> | <循环语句> | '{' <语句列> '}' | <有返回值函数调用语句>; | <无返回值函数调用语句>; | <赋值语句>; | <读语句>; | <写语句>; | <空>; | <情况语句> | <返回语句>;

/*语句分为条件语句、循环语句、{语句列}、call(int a);、call(int a);无返回值函数调用语句;有返回值函数调用语句;、赋值语句;、读语句;、写语句;;、情况语句、返回语句;*/

分析：注意这里是有些语句后面是有分号的，不过条件语句，循环语句循环语句最后是没有分号，不过不保证其内部不出现分号。

<赋值语句> ::= <标识符>=<表达式>|<标识符>['<表达式>']=<表达式>

/*赋值语句举例：a1= 1+2、a1[2]=1*2 */

分析：应该先分析后面的部分，这里应该考虑下计算的先后顺序。

<条件语句> ::= if '('<条件>')'<语句> [else<语句>]

/*条件语句举例：if(a>0) XXX else XXXX*/

分析：(a>0)即为条件，XXX 为语句部分，XXXX 为 else 后的语句部分，else 及其之后部分为可有可无部分，但是 if 以及(条件)是必须的

<条件> ::= <表达式><关系运算符><表达式> | <表达式> //表达式为 0 条件为假，否则为真

/*a>0、a<0、a*/

分析：主要看的是表达式的值，如果表达式为变量，则是变量的值。

<循环语句> ::= while '('<条件>')'<语句>

/*循环语句举例:while(条件) XXXX*/

<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'

/*func (a, b)*/

<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'

/*func (a,b)*/

<值参数表> ::= <表达式>{,<表达式>} | <空>

/*传递给调用函数的参数，允许一个或者多个，允许无参调用函数*/

范例： **int a=func(b,c);** /*有返回值函数调用，参数有多个，这里的 b，c 就是参数*/

function(a); /*无返回值函数调用*/

分析：调用之前函数必须被声明。

<语句列> ::= {<语句>}

/*语句列就是{xxxxxx}，把语句加上大括号*/

范例：{ **int a=1;**

If (a>0)

Printf(a);

}

<返回语句> ::= **return**['(<表达式>')]

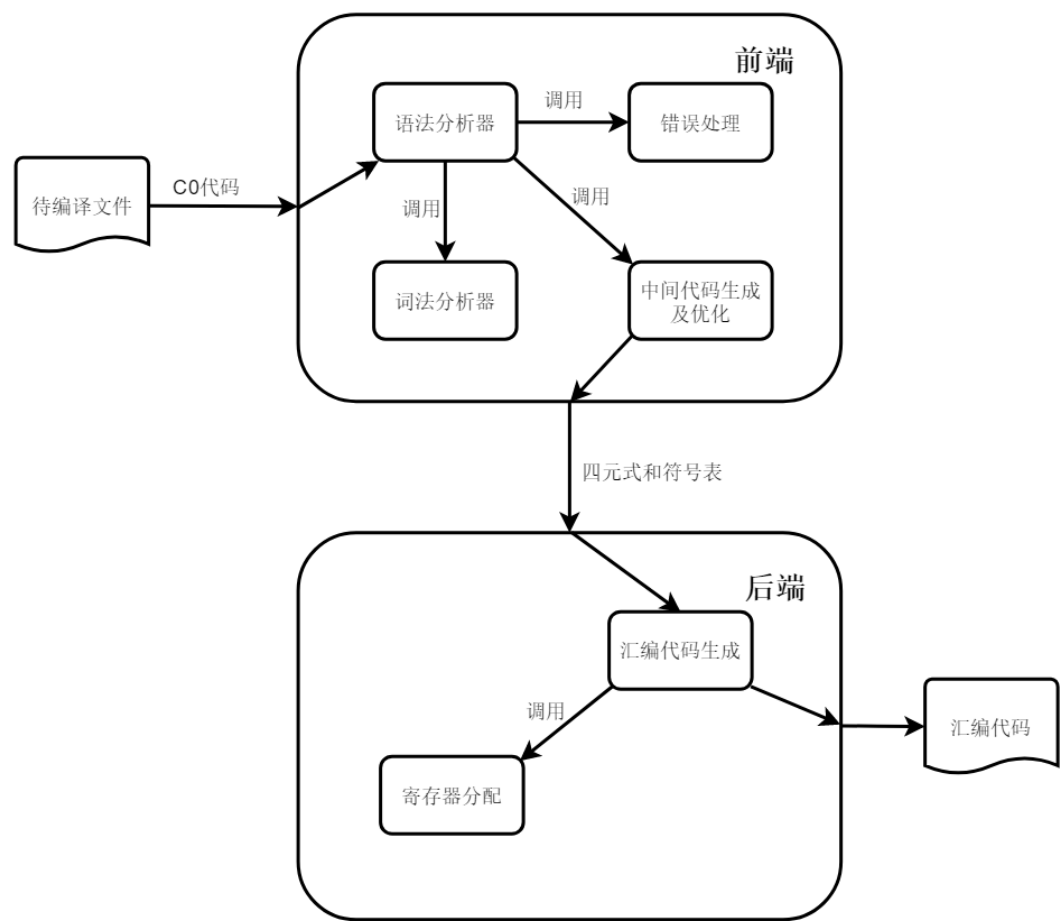
/*返回语句举例：**return** 、**return a+b**、**return 0***/

分析：返回语句主要用于函数的定义当中

系统总体设计

系统结构

本编译器的系统框架如下：



编译器读取待编译代码文件 `C_test.c`，词法分析器调用语法分析器和错误处理模块以递归下降的方式对代码进行分析，同时调用代码生成模块生成中间文件——四元式和符号表，利用中间代码优化程序对中间代码进行表达式优化，得到优化后中间代码和符号表，前端任务完成。汇编代码生成模块通过对中间文件进行处理，在生成汇编代码的时候调用寄存器分配模块进行寄存器分配，完成汇编代码的生成，输出汇编代码文件 `MipsOutput.asm`，后端任务完成。

模块说明

项目共包含 1 个.h 头文件和 8 个.cpp 文件，下面分别介绍各个文件的功能和作用：

文件名	功能
myMain.h	项目的统一头文件，包括所有函数和共享变量的定义
main.cpp	完成编译器的初始化和过程中主要入口函数的调用工作
lexical.cpp	完成词法分析器的工作
parser.cpp	完成语法分析的工作，包括中间代码生成等函数的调用
idTable.cpp	完成与符号表有关的数据结构和函数的定义及使用
infixNotation.cpp	完成与中间代码有关的数据结构和函数的定义及使用
error.cpp	完成与错误处理有关数据结构和函数的定义及使用
infixOptim.cpp	完成与中间代码优化有关的数据结构和函数的定义及使用
target.cpp	完成与汇编代码生成有关的数据结构和函数的定义及使用

详细设计

详细设计包含上述 9 个文件内容的具体实现和接口使用，由于项目开发阶段采用增量开发的方式，完成一些功能后对已有程序同时进行测试，各个模块也是一点点加进去的，因此我将按照时间（开发）顺序对各个模块的详细设计进行解释，并伴随测试的解释和结果的呈现。

简单文法的递归下降实现

时间：2019/04/13~2019/04/14

此次完成文法如下：

$E \rightarrow TE'$	$E' \rightarrow +TE' $	$T \rightarrow FT'$
$T' \rightarrow *FT' $	$F \rightarrow (E) i$	$i \rightarrow 0 \dots 9$

在初期，需完成老师布置的上述文法，它是一个简单的表达式文法，因此只需建立以下 6 个文件：

1. **lexical.cpp** 由于此文法很简单，其不会涉及到词的识别，一个词就是一个字符，因此只需负责取词（字符）并放入全局变量 **ch** 中，供给语法分析器使用，另外还存在全局变量 ***pc**，保存 **ch** 的位置信息；

- a) 主要数据结构

`char ch;` //全局变量，存储当前字符

- b) 主要函数接口

`void nextchar();` //获得下一个字（符）

2. **parser.cpp** 语法分析器，负责所有文法（5 条）的函数实现，当需要读取下一个词时，便调用词法分析器中的函数，使用 **ch** 中的值即可，在需要生成四元式的时候，调用四元式的插入函数，插入四元式；注意在递归下降的涉及到参数的传递，因此函数的参数使用引用类型，方便返回使用；主要函数接口如下：

- a) `void E(string &infixString);` //文法 E 函数，参数为 E 返回表达式

- b) `void E_(string &infixString);` //文法 E' 函数

- c) `void T(string &infixString);` //文法 T 函数

- d) `void T_(string &infixString);` //文法 T'函数
- e) `void F(string &infixString);` //文法 F 函数
- 3. `error.cpp` 不具备错误处理功能, 仅仅输出错误信息, 方便调试;
`void error(string s);` //输出错误信息 s
- 4. `infixNotation.cpp` 此时的四元式仅仅包含+和*两种语句, 因此只需实现这两种的插入即可;
 - a) 主要数据结构
 - i. 四元式表项


```
typedef struct{
                string _operator; //操作符
                string operand1; //源操作数 1
                string operand2; //源操作数 2
                string operand3; //目的操作数
            }infixNotation;
```
 - ii. 四元式表


```
vector<infixNotation> infixTable;
```
 - b) 主要函数接口
 - i. `string createTempVar();` //产生表达式计算的中间变量的唯一表示
 - ii. `void insertInfix(string _operator, string operand1, string operand2, string operand3);` //插入四元式到四元式表中
 - iii. `void outputInfix();` //打印四元式的可读形式
 - iv. `void printInfixTable();` //打印四元式表
- 5. `main.cpp` 初始化环境, 调用递归下降入口函数 `E()`开始并打印结果信息;
- 6. `myMain.h` 包含头文件的引入, 1, 2, 3, 4 中所有函数定义、四元式数据结构的定义和其他必须结构的定义;

测试结果如下, 细节问题请查看代码, 见文件夹: 1.简单文法的递归下降实

现

```

(9+9*9)*9
(9+9*9)*9
Succeed!
#t0 = 9 * 9
#t1 = 9 + #t0
#t2 = #t1 * 9

MUL      9      9      #t0
ADD      9      #t0     #t1
MUL      #t1     9      #t2
请按任意键继续

```

表达式文法的实现

时间：2019/04/15~2019/04/18

此次完成的文法如下（不包括红色部分）：

```

<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> | <标识符> '[' <表达式> ']' | <整数> | <字符>
           > | <有返回值函数调用语句> '(' <表达式> ')'
<标识符>  ::= <字母> {<字母> | <数字>}
<整数>    ::= [+|-] <无符号整数> | 0
<无符号整数> ::= <非零数字> {<数字>}
<数字>    ::= 0 | <非零数字>
<字符>    ::= '<加法运算符>' | '<乘法运算符>' | '<字母>' | '<数字>'
<字母>    ::= _ | a | ... | z | A | ... | Z
<非零数字> ::= 1 | ... | 9
<加法运算符> ::= + | -
<乘法运算符> ::= * | /

```

由于简单文法中不存在词法分析过程（每个单词只有一个字符，且是特殊字符 E, F 等），而真正的表达式文法中含有标识符，数字等，需要进行词法分析，因此在原先的基础上须改进词法分析器 `lexical.cpp` 功能和接口；文法也发生了改变，需写语法分析器 `parser.cpp`；加入错误处理，修改 `error.cpp`；具体如下：

1. `lexical.cpp` 添加了真正的词法分析部分，实现了对当前所需词的识别，包括标识符，+，-，*，/，常量(int, char)，括号((,),[,])，识别的词信息放在全局变量 `symbol sy` 中，并提供接口函数 `insymbol()` 读取下一个词；

- a) `symbol` 的数据结构如下：

```

typedef enum{
    identi, // 标识符

```

```
plus, minus, times, idiv, // + - * /(int)
intcon, charcon, stringcon, // int char string
voidsy, // void
lparent, rparent, lbrack, rbrack, // ( ) [ ]
}symbol;
```

b) 主要函数如下:

- i. void insymbol(); //获得下一个词, 词信息放入 sy 中;
- ii. void nextchar(); //获取下一个字符, 字符信息放入 ch 中;

2. parser.cpp 重新完成了上述新文法对应的函数, 实现方法大致相同, 以表达式项为例说明一些细节, 以 $a+b*c$ 为例:

- a) 传参问题: 当表达式函数调用项函数对 $b*c$ 进行解析时, 需要利用引用传参的方法, 获取 $b*c$ 的返回结果 temp, 然后用于生成 $a+temp$, 其中 temp 是新的临时变量, 代表 $b*c$ 的结果, 之所以不用项函数的返回值返回 temp 是因为返回值需要返回 $b*c$ 的类型信息, 用于判断当前运算数据的统一性。目前还没有使用到返回信息, 因此此时项函数的返回值还都是空。其他函数也一样。
- b) 错误处理: 在语法解析的同时, 需要完成错误处理, 此时的错误处理还没有完全使用“恐慌模式”, 而是采用打印错误信息的方式帮助调试。
- c) 主要函数接口

- i. void expression(string &infixString); //表达式文法函数, 参数为其表达式结果
- ii. void term(string &infixString); //项文法函数, 参数为项的结果
- iii. void factor(string &infixString); //因子文法函数, 参数为因子的结果
- iv. void integer(string &infixString); //整数文法函数, 参数为整数的结果

3. error.cpp 无须改动

4. infixNotation.cpp 添加减法和除法(整数除)的四元式生成, 与前面相似;

5. main.cpp 主入口函数变为表达式函数 expression(), 初始化程序改为新的初始化;
6. myMain.h 将新的数据结构加入头文件, 包括 symbol 类型 (枚举类型, 包括标识符 identi, 加号 plus 等), 以及其余 lexical.cpp 中的数据结果, 所有文法函数的定义也发生了改变;
7. 还有部分表达式文法的功能没有实现, 函数、数组、数据类型、标识符的取值寻址等, 但都留有相应的接口, 避免后面添加时代码修改量较大;

测试结果如下, 由于实现时细节问题较多, 详情请查看代码, 见文件夹: 2.

表达式文法的实现

```
( 'A' + 1 ) * ( 20 + b[0] ) + f(c) * a / ( d - 44 )
Compiler end!
expression: #t8
#t0 = 65 + 1
#t2 = 20 + b[0]
#t3 = #t0 * #t2
#t5 = f(c) * a
#t6 = d - 44
#t7 = #t5 / #t6
#t8 = #t3 + #t7

ADD      65      1      #t0
ADD      20      b[0]   #t2
MUL      #t0     #t2     #t3
MUL      f(c)    a       #t5
SUB      d       44     #t6
DIV      #t5     #t6     #t7
ADD      #t3     #t7     #t8
请按任意键继续
```

常量和变量的引入

时间: 2019/04/19~2019/04/30

此次在上次文法的基础上, 添加了如下文法:

```
<常量说明> ::= const<常量定义>; { const<常量定义>; }
<常量定义> ::= int<标识符>=<整数> {,<标识符>=<整数>}
               | char<标识符>=<字符> {,<标识符>=<字符>}
<整数>      ::= [ + | - ] <无符号整数> | 0
<标识符>    ::= <字母> {<字母> | <数字>}
<变量说明>  ::= <变量定义>; {<变量定义>;}
<变量定义>  ::= <类型标识符>(<标识符>|<标识符> '[' <无符号整数> ']')
               {(<标识符>|<标识符> '[' <无符号整数> ']') }
```

上次完成表达式功能时, 由于除了数据类型和函数主要问题以外, 还有一个

重要的问题是标识符，因为没有进行变量的声明和定义，因此当时的标识符解析仅仅是将它的字符串表示出来，并没有进行变量/常量是否存在的校验、取值以及类型的判断等等，因此此次加入变量和常量的部分，使表达式的功能得以完成，除了函数以外的功能这次都得以实现。

1. lexical.cpp 在原先词法分析的基础上增加了对变量、常量等 symbol 的识别，如 int, char, const 等，对 insymbol()函数进行修改即可；
2. parser.cpp 增加了常量说明，定义等部分的实现，并对原先的函数做以修改，加入数据类型部分、标识符的校验、查询以获取其类型、值信息的部分。部分函数接口：
 - a) type expression(string &infixString); //返回值为表达式结果类型
 - b) type term(string &infixString); //返回值为项的结果类型
 - c) type factor(string &infixString); //返回值为因子的结果类型
 - d) type integer(string &infixString); //返回值为整数的结果类型
 - e) void constState(); //常量声明文法函数
 - f) void constDef(); //常量定义文法函数
 - g) void varState(); //变量声明文法函数
 - h) void varDef(); //变量定义文法函数
3. idTable.cpp 引入了符号表以进行标识符等变量的存储，主要结构体如下：
 - a) 符号表数据结构
 - i. typedef enum{
 voids, ints, chars,
 }type; //all types in C0
 - ii. typedef enum{
 consts, vars, params, funcs,
 }kind; // all kinds in C0
 - iii. typedef struct{
 alphabet name; //标识符名， alphabet 是 char 数组
 kind kd; //标识符类别
 type typ; //变量类型： void, int, char

```

        int addr;          //变量地址
        int length;        //变量长度
        int level;         //变量的等级，0 为全局，1 为局部
    }tableElement; //表项

```

iv. `vector<tableElement> idTable;` //符号表

b) 符号表函数接口

- i. `int lookup(const char name[]);`//查找标识符，返回其在表中位置
- ii. `void insertTable(const char name[], kind kd, type tpy,int addr, int length, int level);`//插入标识符
- iii. `bool isDefinable(const char name[]);`//判断标识符是否可定义
- iv. `void printIdTable();`//打印符号表

4. `error.cpp` 加入恐慌模式的错误处理方式，例如常量说明时出错为分号，则一直读取下一个词直到其为分号，则说明当前的常量说明部分结束，进行下一条语句的处理。从出错处到分号处中间的词都被忽略，即恐慌模式。主要函数接口如下：

- a) `void error(string s);` //打印错误信息
- b) `void warn(string s);` //打印警告信息
- c) `void skipUntil(symbol nexts[]);` //跳过符号直到出现 `nexts` 中的符号

5. `infixNotation.cpp` 添加了对常量、变量、数组获取的中间代码表示；

6. `main.cpp` 基本功能不变，细节有变动来适应改动后的程序；

7. `myMain.h` 加入了上述及其他的数据结构和函数接口；

测试结果如下，详情请查看代码，见文件夹：3.常量和变量的引入

```

const int a=11,b=2;const char C='C',D='D';('A'+1)*(20+b[0])+f(c)*a/(d-44)
const int a=11,b=2;const char C='C',D='D';('A'+1)*(20+b[0])+f(c)*a/(d-44)Compiler end!
a 11 int
b 2 int
C 67 char
D 68 char
#t0 = 65 + 1
#t2 = 20 + b[0]
#t3 = #t0 * #t2
#t5 = f(c) * a
#t6 = d - 44
#t7 = #t5 / #t6
#t8 = #t3 + #t7

CONST 11 int a
CONST 2 int b
CONST 67 char C
CONST 68 char D
ADD 65 1 #t0
ADD 20 b[0] #t2
MUL #t0 #t2 #t3
MUL f(c) a #t5
SUB d 44 #t6
DIV #t5 #t6 #t7
ADD #t3 #t7 #t8

idTable
name kind type addr length level
a 0 1 11 0 0
b 0 1 2 0 0
C 0 2 44 0 0
D 0 2 59 0 0
staticTable
a 0 1 11 0 0
b 0 1 2 0 0
C 0 2 44 0 0
D 0 2 59 0 0
请按任意键继续. . .

```

语句的使用与简化版程序的诞生

时间：2019/05/01~2019/05/05

此次添加的文法如下：

```

<程序> ::= [<常量说明>][<变量说明>]{<有返回值函数定义>|<无返回值函数定义>}<主函数>
<主函数> ::= void main('(')'{'<复合语句>'}'
<复合语句> ::= [<常量说明>][<变量说明>]<语句列>
<语句列> ::= {<语句>}
<语句> ::= <条件语句>|<while 循环语句>|<for 循环语句>|{'<语句列>'}|<有返回值函数调用语句>;|<无返回值函数调用语句>;|<赋值语句>;|<读语句>;|<写语句>;|<空>;|<情况语句>|<返回语句>;
<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> //表达式为 0 条件为假，否则为真
<while 循环语句> ::= while('<条件>')<语句>
<赋值语句> ::= <标识符>=<表达式>|<标识符>['<表达式>']='<表达式>

```

前面完成了变量的声明和定义，表达式的解析后，但是变量依然不具有值，计算仍不完整，因此应该进一步为标识符赋值，这样表达式计算才完整。因此加入语句，使计算得以完整，计算以便成了简化版程序。没有函数功能（包括函数调用和返回），没有 for 循环和 switch 多分支语句。本次程序中出现了第一个函

数，也是唯一一个函数——main 函数，因此变量/常量有了局部和全局之分，前面的表达式中的变量因为没有域的概念，全都以全局变量看待，而简化版程序将变量的作用域引入了进来。

1. lexical.cpp 加入了对 if, while, else 等新加入的 symbol 的识别和定义；
2. parser.cpp 增加了一些新的文法对应的函数：
 - a) void program(); //程序文法的函数
 - b) void mainDef(); //主函数文法的函数
 - c) void complexState(); //复合语句的函数
 - d) void stateList(); //语句列的函数
 - e) void statement(); //语句的函数
 - f) void ifState(); //if 语句的函数，注意分支选择利用 label 来实现，将条件的值与 1 比较，根据比较结果跳到对应的 label 完成分支选择
 - g) void whileState(); //while 循环语句的函数，在循环开始和结束都生成 label，分别在开始和结束同样根据条件决定是否跳到对应的 label
 - h) void judgement(string &infixString); //条件的函数，参数返回值是条件的结果
 - i) void assignState(); //赋值语句的函数
3. error.cpp 在 parser 中继续添加错误处理的方法，不同的文法对应不同的忽略值 symbol nexts[]，当发生错误时，继续调用 skipUntil()函数采取恐慌模式；
4. infixNotation.cpp 添加对 EQL, GTR 等比较操作，JMP 跳转操作，label 等用于 if, while 语句的四元式生成
5. idTable.cpp 全局变量和局部变量都存在，lookup 函数在查找时需按照先局部，后全局的顺序进行，才有自底向上遍历方式；
6. main.cpp 基本功能不变，主入口函数和初始化发生了改变
7. myMain.cpp 对上述所有的函数和数据结果改动添加进去

测试结果如下，详情请查看代码，见文件夹：4.语句的使用和简化版程序的诞生


```

#t0 = 65 + 1
#t1 = 20 + b
#t2 = #t0 * #t1
#t4 = f(c) * a
#t5 = 20 + 2
#t6 = cc[#t5]
#t7 = #t6 - 45
#t8 = #t4 / #t7
#t9 = #t2 + #t8
#t10 = 1 + 2
#t11 = #t9 == #t10
goto LABEL0 if #t11 == 0
LABEL1
goto LABEL2 if bb == 0
#t12 = bb + 1
bb = #t12
jmp LABEL1
LABEL2
jmp LABEL3
LABEL0
#t13 = 33 + 33
aa = #t13
LABEL3
CONST    int    11    a
CONST    int     2    b
CONST    char   67    c
CONST    char   68    d

```

函数的出现与符号表的一次改革

时间：2019/05/06~2019/05/12

本次添加文法如下：

<有返回值函数定义> ::= <声明头部> '(' <参数> ')' '{' <复合语句> '}'
 <声明头部> ::= int <标识符> | char <标识符>
 <无返回值函数定义> ::= void <标识符> '(' <参数> ')' '{' <复合语句> '}'
 <参数> ::= <参数表>
 <参数表> ::= <类型标识符> <标识符> { <类型标识符> <标识符> } | <空>
 <返回语句> ::= return ['(' <表达式> ')']
 <有返回值函数调用语句> ::= <标识符> '(' <值参数表> ')'
 <无返回值函数调用语句> ::= <标识符> '(' <值参数表> ')'
 <值参数表> ::= <表达式> { <表达式> } | <空>
 <语句> ::= <条件语句> | <while 循环语句> | <for 循环语句> | '{' <语句列>
 '}' | <有返回值函数调用语句>; <无返回值函数调用语句>; | <赋值语句>; | <读语句>;
 | <写语句>; | <空>; | <情况语句> | <返回语句>
 <因子> ::= <标识符> | <标识符> '[' <表达式> ']' | <整数> | <字符> | <有返回值函数调用语句> | '(' <表达式> ')'

前面已经完成了一个简化版程序，除了没有子函数以外，它具备面向过程语句的所有要求——顺序选择分支都有，因此此次将函数的文法也加入进去，C0 文法将完全被实现。

1. lexical.cpp 词法分析器继续加入 return 等和函数有关 symbol 的解析；
2. parser.cpp 完成函数有关的函数，函数实现时注意参数顺序的排序，由于 C 语言的压参方式是自右向左，具体见问题总结：[难点一：函数的实现](#)（此处存在超链接，后面所有相似的地方也是超链接）；

- a) `void globalVarState();` //全局变量声明 虽然前面已经定义了变量声明 `varState()`函数,但由于程序文法中,全局变量声明后是有返回值的函数定义,这两者的第一个词都有可能是 `int/char`,而普通的变量声明函数中没有考虑后面出现有返回值函数的情况,因此有了全局变量说明。
 - b) `void funcWithReturnDef();` //有返回值函数定义文法的函数
 - c) `void funcWithoutReturnDef();` //无返回值函数定义文法的函数
 - d) `void parameter(int &count);` //参数文法的函数,参数是参数个数
 - e) `void paramList(int &count);` //参数列文法的函数
 - f) `void funcWithReturn(string &infixString);` //有返回值函数调用文法的函数,参数是函数的结果
 - g) `void funcWithoutReturn();` //无返回值函数调用文法的函数
 - h) `void paramValueList(int funcIndex);` //值参数表文法的函数,参数是函数在符号表中的 `index`
 - i) `void returnState();` //返回语句文法的函数
3. `idTable.cpp` 由于函数的引入,全局变量和局部变量交错出现在,原先符号表 `lookup` 的自底向上方式不再适用,所以需对符号表进行改进,使用一个子文法表对符号表中变量的域进行管理,并使用基于子文法表的方式改进 `lookup` 函数的实现,从而对不同域中的变量进行合理查找。由于篇幅原因,不在这里说明,具体内容可见附录——[符号表的第一次改革](#)。
4. `infixNotation.cpp` 加入对参数、返回语句、调用语句等的四元式生成;
5. `main.cpp/myMain.h` 与前面一样,进行扩增即可。

测试结果见[前端测试结果演示](#),详情请查看代码,见文件夹: 5.函数的出现和符号表的一次改革

前端工作的初步完成

自上节实现完函数以后, [所有的文法](#)已经被实现,而前端工作到现在也已经基本实现,麻雀虽小五脏俱全,该有的东西都有了,虽然没有做优化,没有 `for` 循环和 `switch` 多分支,但 `while` 和 `if` 语句足以代替其实现面向过程的程序设计。

前端了产生了后端所需的中间文件——符号表和四元式。前端的测试也随着功能的增加而进行，目前为止前端的代码在对应的测试用例下正确运行，没有出现错误。下一步将开始后端的实现。

堆栈体系的理解与设计

时间：2019/05/13~2019/05/19

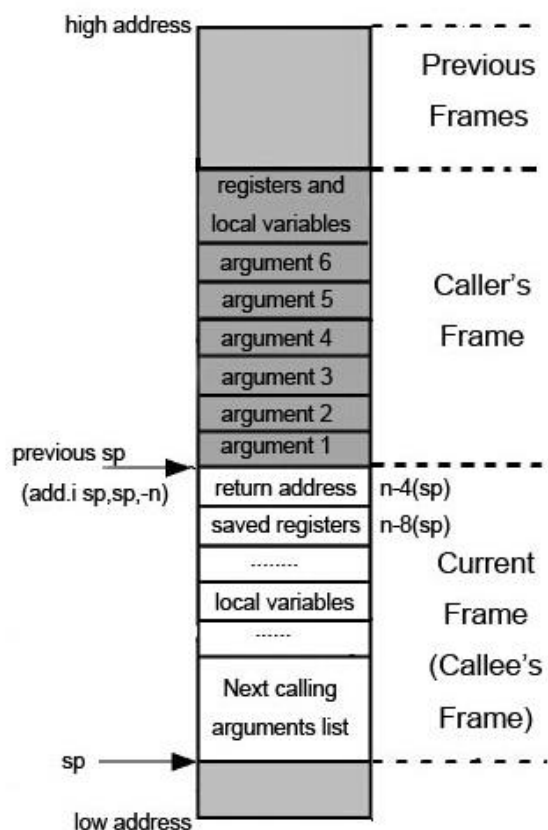
完成了前端工作以后，[变量的位置问题](#)一直困扰着我，因为计算和代码生成时都涉及到变量的取值，而这就涉及到变量的存储位置。后面在一步步继续了解后，难题逐渐解决。后端开始之前，先得理解 MIPS 汇编代码的惯例和 ARM 堆栈体系，函数调用过程，这样才能生成合理而正确的目标码。

首先了解 x86 和 ARM 的结构，了解函数执行的过程和编码方法，再使用 G++ 编译简单代码生成其 MIPS 代码进一步理解其过程。同时，我也对我的参考代码进行了阅读，这里与他产生了分歧，而我也最终使用了我自己的方法完成代码生成。

起初我打算使用这样的结构，这是参考 x86 的体系：

调用者完成的部分	调用者 CALL 之前的栈顶	
	参数 N	
	...	
	参数 2	
	参数 1	
	返回地址（代码区 CALL 之后的指令地址）	
被调用者完成的部分（函数体）	旧%ebp/%fp	← %ebp/%fp
	局部变量 1	
	局部变量 2	
	...	
	局部变量 N	← %esp/%sp

但是后面再阅读 ARM 体系结构时发现他是这样的：



显然这两者是不同的，传统的 ARM 体系中不适用栈帧\$fp，而用\$sp 完成栈中元素的定位，x86 则使用%ebp 和%esp 交互的方式，这种方法对栈中元素的定位更加方便。因此我采用栈帧，栈顶交互的方法，使用了如下的体系：

调用者完成的部分	参数 N	
	...	
	参数 2	
	参数 1	
	旧\$fp	\$fp
被调用者完成的部分（函数体）	return address	-4(\$fp)
	saved registers	-8(\$fp)
	局部变量 1	
	局部变量 2	← \$sp
	...	
	局部变量 N	

前面我也提到说“参考的代码中与我有了分歧”，指的就是这个函数的栈变化情况，我个人觉得他的方式不太对，他将参数放在了返回地址的下面，而 x86 和 ARM 都是在上面（高地址），因此我没有使用它中的方法，这也算是“取其精华去其糟粕”吧。

基于上面的架构，在函数生成 MIPS 时只需要进行下面的步骤即可：

1. 保存旧的堆栈

```
addi $sp, $sp, -4
sw $fp, ($sp)
move $fp, $sp    sp->fp
addi $sp, $sp, -8
sw $ra, -4(fp)
```

-----即如下-----

```
addi $sp, $sp, -12
sw $fp, 8($sp)
sw $ra, 4($sp)
addi $fp, $sp, 8
```

2. 局部变量入栈

```
addi $sp, $sp, 4
```

3. 函数语句

...

4. 函数返回

```
lw $ra, -4($fp)
move $sp, $fp
lw $fp, ($sp)
addi $sp, $sp, 4
jr $ra
```

函数生成框架已经有了，下一步是进行寄存器分配。

寄存器分配算法的历史性翻车

时间：2019/05/20~2019/05/21

完成了框架的设计之后，就需要对具体的四元式进行逐条翻译，生成对应汇编代码。寄存器分配是其中的主要问题，在了解了寄存器分配的相关算法之后，有两种分配方案可以使用。

其一，指定寄存器法。每次都使用固定的寄存器，在使用前必须从内存中取值，操作完成后必须将目的寄存器值重新存储到内存。这样做生成的代码效率比较低，有较多的冗余操作，但简单。

其二，动态分配法。采用龙书上介绍的方法，对寄存器和变量进行跟踪，根据其状态动态的分配寄存器，效率高，没有冗余操作，但较复杂。

不想当将军的士兵不是好士兵，所以我选择方法二，其中包含两个数据结构：寄存器描述符和地址描述符。前者指出寄存器中存着哪些变量的值，是一对多的；后者指出每个变量存在 哪些地址中，也是一对多的。一对多就意味着复杂。

对这些数据结构的管理依照下面操作：

1. LD R, x

- a) 修改 R 寄存器的描述符，使之只含有 x
- b) 修改 x 地址描述符，使之含有 R
- c) 从其他含有 R 的变量的地址描述符中删除 R

2. ST x, R

修改 x 的地址描述符，使之包含自己的内存地址（R 已经在其中）

3. op Rx, Ry, Rz

- a) 改变 Rx 的寄存器描述符，使之只含 x
- b) 改变 x 地址使它只包含 Rx（不再含有其内存地址）
- c) 从其余变量的地址描述符中删除 Rx

4. 复制语句 x=y

- a) LD Ry, y（有必要的话）
- b) x 的寄存器描述符中加入 Ry
- c) 修改 x 的地址描述符，只含 Ry

鉴于两种描述符需要不断进行上述的遍历、匹配工作，由于本编译器的处理量较小，因此使用 vector 实现即可，每个寄存器描述符使用 `vector<string>`，地址描述符使用 `vector<int>` 其中内存为正，寄存器为负（包括-1 ~ -32）。

MIPS 中可用于分配的寄存器有 8-15(t0~t7)，16-23 (s0~s7)，24-25 (t8~t9)，其余的寄存器没有使用到，其中使用 s0-s7 时需要预先存储，简单期间本算法暂时不使用，因此实际需要 t0~t9 寄存器。

在对上述的方法进行编程实现时，遇到了巨大的阻碍，一晚上寸步难行。逻辑复杂程度高，且编程实现复杂，由于时间原因，我没有继续方法二，而是改用简单的方法一。这是开发过程中的一次历史性翻车。

真香定律之简单汇编代码生成

时间：2019/05/21~2019/05/23

方法一真好用！不当将军了，还是先从士兵做起吧，真香！上次实现方法二失败后，选择使用指定寄存器的方法进行寄存器分配，每一条指令都使用\$t0, \$t1, \$t2, \$t3 中的一个。增加文件 target.cpp，实现目标代码生成。实现时关于[参数的位置和数组的计算](#)又出现了一些问题，见问题总结处。整个过程主要函数和功能如下：

1. void mipsProgram(); //目标代码生成主函数入口
2. void mipsMainDef(); //Main 函数目标代码生成主函数
3. void mipsFuncDef(); //其他函数的目标代码生成主函数
4. void funcContent(); //函数主体的目标代码生成函数
5. void operandToRegister(string operand, string reg); //将四元式中的操作数取出来放进寄存器 reg，其中需根据操作数的类别对其采取不同的操作：
 - a) 非标识符（数字）：直接放入寄存器；
 - b) 常量：直接取符号表中 addr 部分（真值）放入寄存器；
 - c) 局部变量/全局变量：计算其在栈中位置，并取出来放入寄存器；
6. void storeThirdOperand(string operand, string reg); //将寄存器中的值存回存储器，其中需判断其为全局变量还是局部变量，采取不同的位置；
7. int getGlobalOffset(string name); //获得全局变量的偏移；
8. int getLocalOffset(string name); //获取局部变量的偏移；
9. void mipsVarDef(bool Global); //变量定义的目标代码生成；
10. void funcTail(); //函数尾目标代码生成
11. void funcHead(); //函数头目标代码生成
12. void mainHead(); //main 函数头目标代码生成

测试结果见[系统测试之优化前的测试](#)，详情请查看代码，见文件夹：6.真香定律之简单汇编代码生成

就此，已经完成了从语法分析到汇编代码生成的整个过程，编译器已经算完整的被实现了，经过系统测试，出现的错误都已经被修改，没有新的错误发生。

中间代码优化与符号表的二次改革

时间：2019/05/28~2019/06/02

虽然编译器已经被完整实现，但现在的它还是非常的粗糙的，没有任何的优化。因此，将对中间代码进行优化，使用基本块的划分来对表达式进行优化，使用 DAG 来减少不必要的计算和中间变量，优化后产生新的四元式序列和新的符号表。主要经过三个步骤：

1. 划分基本块得到跨块变量 crossing var
2. 利用四元式对每个基本块建立 DAG
3. 对每个基本块解析 DAG 得到新四元式
4. 重新计算符号表

主要数据结构：

1. DAG 结构体

```
struct DAGNode{  
    int number;  
    string content;  
    set<DAGNode*> parents;  
    DAGNode* lchild;  
    DAGNode* rchild;  
};
```

2. vector<DAGNode*> allNodes; //所有 DAG 节点
3. map<string, int> varNodeTable; //变量与节点的映射
4. map<string, int> varWithInitial; //变量的首次出现位置
5. set<string> crossingVars; //跨基本块变量的集合
6. vector<infixNotation> newInfixTable; //新的四元式表

主要函数和接口：

1. inline bool isManageable(string operator_); // 四元式可优化
2. inline bool isVarName(string operand); //操作数是变量
3. void insertNewInfix(string _operator, string operand1, string operand2, string operand3); //插入新的四元式
4. void insertOperands(set<string>& varInBlock, infixNotation infix); //将四元式插入当前基本块集合
5. void checkVarExistenceCount(map<string, int>& varExistenceCount, set<string>& varInBlock); //统计基本块中元素个数
6. void splitBlocks(); //划分基本块
7. int setupOperand(string operand); //为变量建立其对应的 DAG
8. inline bool isMidNode(DAGNode *node); //判断是否为中间节点
9. void infixToDAGNode(infixNotation infix); //为四元式建立 DAG
10. void setMidNode(DAGNode* node, vector<DAGNode*>& queue, set<int>& inQueue); //将所有中间节点插入计算队列
11. DAGNode* findNodeWithNumber(int number); //返回节点在所有 DAG 节点中的位置信息
12. void exportCodesFromDAG(); //从 DAG 解析出四元式
13. void resettlAddress(); //重新计算符号表

具体实现算法：

1. 各个数据结构之间的联系与使用
 - a) DAGNode 的数据结构包括一个 number，这是用来定位所有 Node 的唯一性的，content 存放的是自己的操作符，parents 是一个 set，存放自己的父 node，左右儿子分别是两个 DAGNode 的指针；
 - b) allNodes 的 vector 存放所有 DAGNode；
 - c) 使用 map 来 varNodeTable 进行每个变量和其在 allNodes 中存储位置的映射，是最后一次出现的位置，其值是 number，也是自己的 index；利用一个额外的 varWithInitilial 的 Map 来存放每个标识符变量的第一次出现的初始 Index；

- d) 使用 `newInfixTable` 存放优化后的代码
- 2. 基本块划分——当一个变量在多个块中存在时，不能在优化时将其丢弃，通过划分时将其记录在 `crossing var` 的集合中，从而避免在处理时丢弃。
 - a) 统计当前基本块中出现的变量，将其出现次数加 1；
 - b) 统计下一基本块的出现变量， 将其出现次数加 1；
 - c) 重复 1,2 直到所有块被统计，则一旦某一变量的次数大于 1，则其出现在两个基本块中，则为 `crossing var`；
- 3. DAG 的构建——对于每一条可优化的中间代码（`infix`），都会构建 `DAGNode` 并与其父子节点联系，构建节点时进行如下操作（算法）：
 - a) 不是赋值语句的中间代码：
 - i. 左、右操作数建立 `DAGNode` 并返回它在 `allNodes` 中的位置 `index`（建立方法如下）：
 - 1. 利用 `varNodeTable` 根据操作数的名字寻找是否已经被建立
 - a) 是，返回其在 `allNodes` 中的 `index`；
 - b) 否，建立一个 `DAGNode` 节点，填充节点信息，插入 `allNodes` 并建立 `varNodeTable` 的联系，返回其在 `allNodes` 中的 `index`（即为其 `number`，因此每次建立 `DAGNode` 时会同时加入 `allNodes` 并给 `number` 赋值）
 - 2. 按上述方法返回操作数对应的 `DAGNode` 信息；
 - ii. 为目的操作数建立 `DAGNode`
 - 1. 根据操作符和左右操作数的信息遍历 `allNodes` 寻找是否存在相同操作的 `DAGNode`
 - a) 存在，返回其 `number` 值（即 `allNodes` 中的位置，后面不再赘述）
 - b) 不存在，建立 `DAGNode`，填充节点信息，包括子节点、父节点、`number` 和内容 `content`（后面不再赘述，操作相同），设置子节点的父信息，插入 `allNodes` 中，返回其 `number` 值；
 - iii. 在 `varNodeTable` 中根据目的操作数名字寻找是否已存在对应

DAGNode,若不存在,则将得到的 number 赋值给他;做变量对应的 DAGNode 已存在,则将其对应信息改为新的 number (只有在赋值语句时才会出现已存在,因为会赋值,而非赋值语句的变量均为中间变量#t,是不会重复的,改变变量对应的 DAGNode 相当于完成变量的赋值)

b) 是赋值语句的中间代码

- i. 对操作数 1 建立 DAGNode(因为赋值语句的源操作数只有一个,在操作数 1 中存放),和上面方法一致,返回其 index;
- ii. 为目的操作数建立 DAGNode
 1. 若操作数 1 是中间节点(左右儿子均存在),则操作数 1 的节点 number 直接返回,作为目的操作数的 Node,因为该节点的中间代码产生汇编代码后,无论是 add 还是 sub 都是顺便更改目的操作数的值,达到赋值的效果,不需要再赋值;
 2. 若操作数 1 不是中间节点,则其是变量节点或赋值节点,则为目的节点建立 DAGNode,填充节点信息,加入 allNodes 并返回 number 值;
- iii. 在 varNodeTable 中根据目的操作数名字寻找是否存在,不存在则插入 number 的映射信息,若存在则说明该变量前面已经被赋值或作为操作数使用一次了,应将映射关系改为最新的。

c) 综上则完成了 DAG 的构建

- i. 将 $a = 1 * 1 + 1 * 1; b = a + 1; c = a + 1 + 1;$ 进行建图输出,发现与预估吻合,说明建图正确;测试完成。

----- UarNodeTable -----			
Name	Num		
#t0	1		
#t1	1		
#t2	2		
#t3	3		
#t4	3		
#t5	4		
1	0		
a	2		
b	4		
c	3		
----- DAG -----			
num	cont	lchild	rchild
0	1		
1	MUL	0	0
2	ADD	1	1
3	ADD	2	0
4	ADD	3	0

4. DAG 的解析

- a) 根据 DAG 根节点，得到 DAG 的节点计算顺序。注意：
 - i. 叶子节点（即变量初始节点）不计算；
 - ii. 寻找计算顺序时，递归的对子节点进行判断，若为中间节点（即不是叶子节点，赋值语句也是中间节点），则加入计算队列并继续递归；
 - iii. 同一节点只进入一次计算队列；
- b) 对所有变量中初始节点 `varWithInitial` 与最终节点 `varNodeTable` 不同的节点进行一下操作：
 - i. 首先，解释此步的必要性。以 `a=a+1;b=a` 为例。只有叶子节点可能存在于 `varWithInitial`，即变量的初始值的映射，因为叶子节点的 `content` 存放的是变量，此例子中 `a` 就是，因为赋值语句会对变量值进行更改，即优化后会有 `a` 和 `b` 指向节点 `a+1`，此时前后的 `a` 已经不是一个 `a`，在符号表中的须加以区别，因此将所有的叶子节点的 `a` 改为 `a0`，并且将 `a` 值给 `a0`，这样就不担心 `a` 的值改变影响 `a0` 的值；
 - ii. 对所有上述中涉及的点，插入其增广形式到符号表其当前函数的末端，插入 `infix`，更改其叶子节点的 `content` 为增广形式；
- c) 根据 DAG 计算顺序生成对应 `infix`，对每个 `DAGNode` 进行如下操作：
 - i. 找出所有与该节点对应的变量（即该节点的计算结果就是这些变量的值），放在 `varNodes` 中；
 - ii. 对这些变量的生成计算其值对应的 `infix`（如果必要的话），由于有些变量的值在后面块中也会使用，有些变量不会，因此对于同一节点对应的变量，优先选择后面会使用的或许是定义好的变量，中间产生的 `#t` 类型变量尽可能删除：
 1. 将后面不会使用的放在 `Leave` 中，而会使用（`crossing var`）或者是定义的变量放在 `stay` 中保存下来，防止过度优化；
 - iii. 生成对应的 `Infix`

1. 若 stay 为空，则从 leave 中或产生一个 #t 来存放当前节点的值；
 2. 若非空，则取第一个生成 infix，对 stay 中其余的变量都生成赋值语句；
- iv. 将节点的内容改为 stay 中第一个变量，则为该节点对应的变量；
 - v. 删除 leave 在符号表中的值；
- d) 经过上述步骤，已经完成了解析。如图是对 $a = 1 * 1 + 1 * 1; c = a + 1; c = c + 1; b = a + 1 + 1;$ 的重构。经过测试，可以看出中间代码已经得到了优化，符号表也正确，现在需要做的是对地址进行计算，因为地址重计算很简单，无须赘述。

```

#t0 = 1 * 1
a = #t0 + #t0
#t3 = a + 1
b = #t3 + 1
c = b

FUNC      Void      main
VAR      int        a
VAR      int        b
VAR      int        c
MUL      1          1   #t0
ADD      #t0        #t0  a
ADD      a          1   #t3
ADD      #t3        1   b
ASSIGN   b          c

-----Static Id Table-----
name  kind  type  addr  length  level
main  func  voids  0     0      global
a     vars   ints   0     0      local
b     vars   ints   1     0      local
c     vars   ints   2     0      local
#t0   vars   ints   3     0      local
#t3   vars   ints   6     0      local

```

到此，优化部分已经结束，因为优化过程的算法比较复杂，所以引发了很多的错误，但都被一一解决了，因为是编程性错误，就不在问题总结中叙述。上次系统测试的代码优化前中间代码为 55 行，优化后为 44 行，效果显著！

测试过程在算法叙述时已经叙述，细节详情请查看代码，见文件夹：7.中间代码优化与符号表的二次改革

拒绝真香定律之寄存器分配算法

时间：2019/06/03~2019/06/11

先前在自己设计寄存器分配受阻后，毅然决然的选择了简单的方法一。在完成中间代码优化后，士兵已经羽翼丰满，可以尝试再做一次将军。拒绝真香！

在使用指定寄存器的函数时，我留了接口为后面的修改做准备，修改策略如下：

1. 增加寄存器数组<int, string> reg2var 存放寄存器当前存放的变量；
2. 增加变量 map<string, int> var2reg 记录当前变量最后存放的寄存器位置；
3. 增加寄存器数组<int, int>regTime 存放寄存器最近被使用的时间；
4. 当源操作数需要进行取变量时，进行如下操作获取当前变量的寄存器：
 - a) 查找 var2reg
 - i. 若操作数不是变量，则进行 ii，否则进行查找，若存在，则返回该寄存器，结束 a 步骤；
 - ii. 不存在，查找 reg2var 有没有空的
 1. 如果有，则使用该寄存器进行 iii 操作
 2. 如果没有，查找 regTime 寻找最近没有被用过的寄存器，进行 iii 操作，并更改其 regTime 内容；
 - iii. 找到的寄存器，调用原先的 operand2Reg 函数，为寄存器取变量，先用 reg2var 将原先变量和 reg 的关系（var2reg 中）清除，如果操作数是变量，再将 reg2var 中的此寄存器对应项更改为当前变量，否则改为空；
5. 利用取到的寄存器进行 add 等操作；
6. 为目的操作数同样适用上述操作获得寄存器，注意不需要进行 iii 中的取变量操作；
7. 存目的操作数；
8. 对于任何给寄存器里赋值的操作，都需要更改寄存器对应关系；

根据上述算法对原方法进行修改，主要变化函数与接口如下：

1. int getRegister(string operand, bool isSrc) //新的寄存器分配算法，为操作数分配寄存器；
2. int getRegister2(string operand, bool isSrc, bool isCondition); //为有比较操作的四元式分配寄存器；
3. int getNearest(); //根据 LRU 方法获取寄存器

4. `void funcContent();` //使用新的分配方法重写了函数体的目标代码生成

根据上述方法修改代码后，对其进行测试，见[系统测试之优化后的测试](#)，其中出现了一些问题，在测试中将其改出，已经正确。细节请查看代码，见文件 8. 拒绝真香定律之寄存器分配算法-最终版代码。

至此，所有的工作已经做完，编译器已经被完整实现，此次的代码也是最终的编译器版本代码。

测试

各个单元的功能测试和模块之间的集成测试已经在详细设计阶段进行，因此本测试是从整体的角度对整个编译器进行系统测试。

系统测试之优化前的测试

测试代码：

```
int aaa[2], bbb;

int f(int x, int y){
    y = x + 1;
    return (y);
}

void main(){
    int aa
    aa = 1+2*1/2; //加、乘等表达式运算/赋值
    aaa[1] = aa + 1; //局部/全局/数组变量寻址测试
    bbb = f(aa, aaa[1]); //函数测试
    ///条件语句测试
    if(aa + 1 > 0){
        aaa[0] = aa + 1;
    }
    else{
        aaa[0] = aa + 2;
    }
    //循环测试
    while(aa < 5){
        aa = aa + 1;
    }
    bbb = aa;
}
```

测试方法：

测试主要采用对全局变量 `aaa[2]` 和 `bbb` 进行赋值，通过 Mars 查看全局区 `aaa` 和 `bbb` 对应的值的变化情况来判断代码是否正确运行，其中涉及条件、循环、赋值等语句、加/乘等表达式基本运算、数组/局部/全局变量/函数的测试，所有的功能都被测试到了。

测试过程：

1. 局部变量（基于\$fp）测试通过，aa 结果显示为 1
2. 全局变量（基于\$gp）测试未通过，经修改后正确，aaa[1]显示为 3

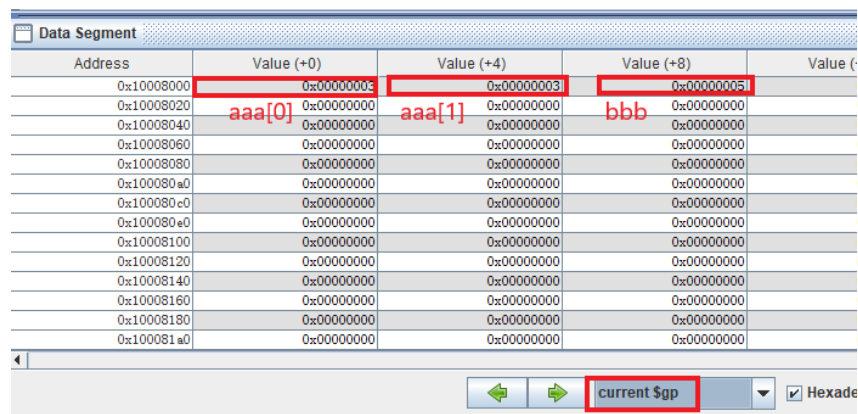
全局区\$gp 的在低地址，而\$fp 是在高地址，因此这两者的 offset 计算不同，一个是正，一个是负，我之前都按\$fp 的情况处理为全负，导致全局变量寻址错误，因此须更改。更改之后经验证全局变量/数组的访问成功。

3. 赋值语句（包含表达式和函数）测试通过；
4. if 语句（if else 分支）测试通过，aaa[0]显示为 3；
5. while 循环语句测试通过，bbb 显示为 5

之前的 while 循环的四元式生成有点问题，因为我使用的方法是先得到条件结果，再将其与 0 进行 BEQ，决定是否继续循环，循环开始的 Label 应该放在条件判断之前，而不是 BEQ 之前，错误的代码导致没有在后继循环中进行条件判断。经修改后循环语句通过。

测试结果：

最终全局区的数据依次为 aaa[0]=3, aaa[1]=3, bbb=5，Mars 演示如图。



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)
0x10008000	0x00000003	0x00000003	0x00000005	
0x10008020	aaa[0]	aaa[1]	bbb	
0x10008040	0x00000000	0x00000000	0x00000000	
0x10008060	0x00000000	0x00000000	0x00000000	
0x10008080	0x00000000	0x00000000	0x00000000	
0x100080a0	0x00000000	0x00000000	0x00000000	
0x100080c0	0x00000000	0x00000000	0x00000000	
0x100080e0	0x00000000	0x00000000	0x00000000	
0x10008100	0x00000000	0x00000000	0x00000000	
0x10008120	0x00000000	0x00000000	0x00000000	
0x10008140	0x00000000	0x00000000	0x00000000	
0x10008160	0x00000000	0x00000000	0x00000000	
0x10008180	0x00000000	0x00000000	0x00000000	
0x100081a0	0x00000000	0x00000000	0x00000000	

current \$gp

系统测试之优化后的测试

依然使用优化前系统测试的代码，结果错误，bbb 的预期结果是 5，实验结果为 6，说明中间产生了错误，现对所有优化后的功能重新测试。

1. ADD

a) 测试用例:

```
int aaa[2], bbb;

void main(){
    int aa;
    int a, b, c;
    aa = 1+1;
    aaa[0] = aa + 1;
    a = (1+1)+(1+1);
    c = a+1;
    c = c+1;
    b = a+1+1;
    aaa[3] = b;
    bbb = c;
}
```

b) 测试结果: 通过观察全局变量的值来判断计算是否正确, 结果显示 $aaa[0] = 3$, $aaa[1] = 0$, $bbb = 6$, $aaa[3] = 6$, 按理说 $aaa[3]$ 已经越界, 但模拟器无法检测, 另外 $aaa[3]$ 的位置就是 bbb 之后, 测试通过!

2. SUB

a) SUB 与 ADD 代码相同, 因此无需测试;

3. MUL/DIV

a) 测试用例:

```
int aaa[2], bbb;

void main(){
    int aa;
    int a, b, c;
    aa = 1+1;
    a = (1*1)+(1*1)+(2/2);
    b = (1*1)+(1*1)+(2/2)-aa;
    c = a / b;
    a = a - b;
    c = c*c - a*a + b*b;
    aaa[0] = a;
    aaa[1] = b;
    bbb = c;
}
```

b) 测试结果: $aaa[0]$, $aaa[1]$, bbb 分别是 2 1 6, 理论值是 2 1 1, 删除 $c = c*c - a*a + b*b$, 结果正确, 则说明此语句出错。

- c) 更改 $c = c * c - a * a$, 结果错误, 更改 $c = c * c$, 正确, 更改为 $c = a * a$, emmmm, 理论值计算错误了, 理论值就是 2 1 6, woc.. 目前正确;

4. if 语句测试

- a) 测试用例

```
int aaa[2], bbb;

void main(){
    int aa;
    int a, b, c;
    aa = 1+1;
    if(aa > 1 + 1*1 - 2*2 + 2*2){
        aaa[0] = 1;
        aaa[1] = 2;
    }
    else{
        aaa[0] = 2;
        aaa[1] = 1;
    }
}
```

- b) 测试结果: 2 1, 正确

- c) 测试用例

```
int aaa[2], bbb;

void main(){
    int aa;
    int a, b, c;
    aa = 1+1;
    if(aa > 1 + 1*1 - 2*2 + 2*2 - (1+1*1)){
        aaa[0] = 1;
        aaa[1] = 2;
    }
    else{
        aaa[0] = 2;
        aaa[1] = 1;
    }
}
```

- d) 测试结果: 1 2 正确

5. 测试 while 循环

- a) 测试用例:

```
int aaa[2], bbb;
```

```

void main(){
    int aa;
    int a, b, c;
    aa = 1+1;
    a = 1;
    while(a < 5){
        a = a + 1;
    }
    aaa[0] = a;
}

```

b) 测试结果：理论是 5，结果是 6，若将条件改为 $a \leq 5$ ，则为 7，因此是哪里出现了错误，分析汇编代码。

- i. 通过读汇编代码，因为动态分配的存在，导致在进行 $a < 5$ 判断时，这里的 a 没有重新取值，而是默认上面 $a=1$ 操作分配的寄存器 $t0$ 为此处的 a 的寄存器 $t0$ 进行比较，若在循环中 $t0$ 被占用，则此处的 a 就不在 $t0$ 中了，因此须对比较语句的寄存器分配进行更改，不再使用先前的值，而是每次都进行取值，这样就不会产生使用固定的寄存器。
- ii. 建立 `getRegister2` 进行比较类的寄存器分配，避免直接寻找寄存器，每次都进行取的寻找，即可完成。
- iii. 经过测试，正确。

6. 使用先前的代码重新测试，结果与优化前一致，测试通过。

lta Segment				
Address	Value (+0)	Value (+4)	Value (+8)	
0x10008000	0x00000000	0x00000003	0x00000005	
0x10008020	aaa[0]	aaa[1]	bbb	
0x10008040	0x00000000	0x00000000	0x00000000	
0x10008060	0x00000000	0x00000000	0x00000000	
0x10008080	0x00000000	0x00000000	0x00000000	
0x100080a0	0x00000000	0x00000000	0x00000000	
0x100080c0	0x00000000	0x00000000	0x00000000	
0x100080e0	0x00000000	0x00000000	0x00000000	
0x10008100	0x00000000	0x00000000	0x00000000	
0x10008120	0x00000000	0x00000000	0x00000000	
0x10008140	0x00000000	0x00000000	0x00000000	
0x10008160	0x00000000	0x00000000	0x00000000	
0x10008180	0x00000000	0x00000000	0x00000000	
0x100081a0	0x00000000	0x00000000	0x00000000	

current \$gp

问题总结

在实验中遇到过很多问题，有编程 BUG，有算法实现困难，也有逻辑错误，这里的问题总结只要针对逻辑错误，即在编译器开发过程中设计的合理性与正确性，而编程的 BUG 和算法的具体实现与设计无关，是个人编程能力的不足。

难点一：函数的实现

语句中的有返回值和无返回值函数调用时，注意区别其与赋值语句的冲突，因为两种文法都是以标识符 `identi` 开始，处理方法依然是保存当前状态的处理点，然后继续读入下一符号，判断是哪种情况，确定后再返回先前的保存点，调用相关函数；此外有无返回值的判断需要访问符号表。

函数调用的难点集中在参数类型的匹配，我们必须在程序中检测返回值和传入参数的类型是否对应一一匹配，由于符号表中函数名紧接着就是其参数，因此需以函数表的 `index` 为基准，便可获取定义时的参数信息，从而进行类型匹配。

另外需要注意的是传参方式的一致性，C 语言函数的传参方式是**从右向左**，即右边的参数在高地址，左边的参数在低地址。从右向左的传参方式相比于从左向右可以实现**可变长参数的函数调用**，因为从右向左的情况下无须知道参数总数即可算出第 k 个参数的地址，而从左向右的传参情况则需要知道总数。我们这里使用**从右向左**的传参方式。

在进行返回值的调用语句编写时发现了新的问题：`lookup` 出错问题，某些存在的函数在查找时显示未定义，经过长时间的调试分析，发现由于局部变量与全局变量交替在符号表中出现的缘故，导致子程序表的功能出现缺陷，即全局变量不全出现在符号表最顶端，因此部分中间的符号表会不能被检索到，因此须改进。现有两种方法，第一种是将全局搜索的范围方法，使用 `level` 进行检测，这样比较耗时，因为需要对所有符号表中元素进行搜索；第二种是建立一个单独存放全局变量的全局表进行搜索，以空间换时间；考虑到本次的实验变量较少，耗时不大，因此选择第一种简单的方法。

难点二：变量的位置

在对符号表中变量的 `addr` 进行处理时，参考了其它代码中的设计方法，在参照的时候其实有很多的疑惑和不解，但随着后端的一步步进行，对编译原理的进一步理解，这些疑惑也烟消云散了，而正因为这些参考，使我少走了很多弯路。

在变量的属性（尤其是地址 `addr`）在插入符号表时，依照了如下的规律，并产生了相应的疑惑：

1. 为什么常量的 `addr` 项是其真值？

在常量的编译时，对应的标识符的运算会直接以其真值代替，因此不需要记录其地址，编译时也用不到。

2. 为什么变量的 `addr` 项是一个随着数量变多递增的值？其代表着什么含义？

局部变量在栈中保存，其地址是相对于当前栈帧的偏移地址，而在函数生成汇编代码时会首先为所有变量空出位置，因此其 `addr` 对应的偏移地址为其在栈中的实际位置。全局变量在静态区保存，其 `addr` 对应其相对于静态区基址的偏移量。

3. 为什么函数的 `addr` 项为 0？

函数的 `addr` 是没有用的，编译时也不会用到。

4. 为什么参数的 `addr` 项也递增？

参数的 `addr` 是在当前函数栈帧的高地址方向，而变量是低地址方向，也同样是依次排开的，所以就是线性的（递增的）。

难点三：参数的位置和数组的计算

由于参数是在栈帧的上面（高地址），而变量在下面（低地址），因此在计算偏移量时变量的负的，而参数是正的。参数的压栈方式是从右向左，越左边的参数偏移量越小，地址越低。

数组的地址计算时需要先取出数组的基址，放入寄存器中。再取出偏移量，乘以 4 计算相对于基址的偏移，然后加上基址，得到数组中元素的实际栈中位置。乘法、加法、存储或取值四种操作都用到了。

附录

符号表的第一次改革

因为原来的遍历法比较费时，而且可能会导致错误，因此改进符号表的实现。

C0 文法比较简单，它仅仅在程序的开始（全局定义）和函数的开始（局部变量定义）进行变量声明定义，不允许在函数过程中或某一个循环语句中进行定义，因此此次符号表的设计不需要像课本上的“分程序结构语言符号表的建立”那么复杂，只需要实现 C0 文法所需即可，不必引入过多的指针、全局变量等。

对课本实现方法进行简化及优化，大体如下：

1.符号表

名字	标识符信息				
标识符名称	种类	类型	地址	长度	域 全局/局部
identi name	const, vars, params, func	void int char	addr	length	level

2.分程序表

上级域索引	符号个数	符号表指针
parent	idNum	idPtr
-1/0 全局的上级为-1，局部为 0	数字	符号表索引

符号表符号类型符合如下规律，都是根据 C0 文法而定，不具备泛化功能：

1. 常量：常量是全局或局部，无数组类型，长度均为 0，域为 0，地址为其实际值
2. 变量：变量是全局或局部，即域为 0 或 1，包含数组类型，长度不定，地址为其在当前域（栈帧）中的相对地址（偏移量）
3. 参数
4. 函数：类型为 void/int/char

关于标识符的定义时的查重或使用时的搜索：

存在一个全局指针：当前的分程序序号 subGrammar = 0,每当进入一个函数定义时 subGrammar++;

1. 定义：
 - a) 全局定义：从全局标识符索引（0）开始遍历所有全局标识符，若重

复，则重复；

- b) 局部定义：由于此时在某个函数中定义，先利用 `subGrammar` 取当前函数的符号表索引，从当前索引开始遍历当前函数的符号，若重复，则重复定义，若不重复，则利用上级域索引取上级函数的符号表索引（即全局搜索，因为局部变量仅含有一次定义机会，不存在多层局部定义，若想实现多层局部定义只需稍作一点修改，即可实现，基本原理相同）

2. 使用

利用当前 `subGrammar` 取当前符号表索引，在当前域中搜索当前符号，若存在则返回符号表索引，否则在全局中搜索，存在则返回其索引，不存在则出错。

前端测试结果演示

```
const int a=11,b=2;
const char c='C',d='D';
int aa,bb,cc[100];
char dd,ee;
int f(char x, int y){x='d'; return (x);}
void ff(){};
int fff(char ch){return (c);}
void main(){
    const int a=11,b=2;
    const char c='C',d='D';
    int aa,bb,cc[100];
    char dd,ee;
    if(('A'+1)*(20+b)+fff(c)*a/(cc[20+2]-45)==1+2){
        while(bb){
            bb = bb+1;f('a'+c, a+1);
        }
    }
    else
        aa = 33+33;
}
Compiler end!
int      a      11
int      b       2
char     c      67
char     d      68
int      aa
int      bb
```



```

int      cc[100]
char     dd
char     ee
param char x
param int y
int f()
x = 100
return x
void ff()
param char ch
int fff()
return c
void main()
int      a    11
int      b    2
char     c    67
char     d    68
int      aa
int      bb
int      cc[100]
char     dd
char     ee
#t0 = 65 + 1
#t1 = 20 + b
#t2 = #t0 * #t1
push c
call fff
#t3 = #RET
#t4 = #t3 * a
#t5 = 20 + 2
#t6 = cc[#t5]
#t7 = #t6 - 45
#t8 = #t4 / #t7
#t9 = #t2 + #t8
#t10 = 1 + 2
#t11 = #t9 == #t10
goto LABEL0 if #t11 == 0
LABEL1
goto LABEL2 if bb == 0
#t12 = bb + 1
bb = #t12
#t13 = 97 + c
#t14 = a + 1
push #t13

```

```

push #t14
call f
#t15 = #RET
jmp LABEL1
LABEL2
jmp LABEL3
LABEL0
#t16 = 33 + 33
aa = #t16
LABEL3

```

CONST	int	11	a
CONST	int	2	b
CONST	char	67	c
CONST	char	68	d
VAR	int		aa
VAR	int		bb
VAR	int	100	cc
VAR	char		dd
VAR	char		ee
PARAM	char		x
PARAM	int		y
FUNC		int	f
ASSIGN	100		x
RET			x
FUNC		void	ff
PARAM	char		ch
FUNC		int	fff
RET			c
FUNC		void	main
CONST	int	11	a
CONST	int	2	b
CONST	char	67	c
CONST	char	68	d
VAR	int		aa
VAR	int		bb
VAR	int	100	cc
VAR	char		dd
VAR	char		ee
ADD	65	1	#t0
ADD	20	b	#t1
MUL	#t0	#t1	#t2
PUSH			c
CALL			fff

ASSIGN	#RET		#t3
MUL	#t3	a	#t4
ADD	20	2	#t5
GETARR	cc	#t5	#t6
SUB	#t6	45	#t7
DIV	#t4	#t7	#t8
ADD	#t2	#t8	#t9
ADD	1	2	#t10
EQL	#t9	#t10	#t11
BEQ	#t11	0	LABEL0
LABEL			LABEL1
BEQ	bb	0	LABEL2
ADD	bb	1	#t12
ASSIGN	#t12		bb
ADD	97	c	#t13
ADD	a	1	#t14
PUSH			#t13
PUSH			#t14
CALL			f
ASSIGN	#RET		#t15
JMP			LABEL1
LABEL			LABEL2
JMP			LABEL3
LABEL			LABEL0
ADD	33	33	#t16
ASSIGN	#t16		aa
LABEL			LABEL3

subGrammarTable

parent	IdNum	IdPtr
-1	10	0
0	3	10
0	1	13
0	2	14
0	26	16

idTable

name	kind	type	addr	length	level
a	consts	ints	11	0	global
b	consts	ints	2	0	global
c	consts	chars	67	0	global
d	consts	chars	68	0	global
aa	vars	ints	0	0	global
bb	vars	ints	1	0	global

cc	vars	ints	2	100	global
dd	vars	chars	102	0	global
ee	vars	chars	103	0	global
f	funcs	ints	0	2	global
x	params	chars	0	0	local
y	params	ints	1	0	local
ff	funcs	voids	0	0	global
fff	funcs	ints	0	1	global
ch	params	chars	0	0	local
main	funcs	voids	0	0	global
a	consts	ints	11	0	local
b	consts	ints	2	0	local
c	consts	chars	67	0	local
d	consts	chars	68	0	local
aa	vars	ints	0	0	local
bb	vars	ints	1	0	local
cc	vars	ints	2	100	local
dd	vars	chars	102	0	local
ee	vars	chars	103	0	local
#t0	vars	chars	104	0	local
#t1	vars	ints	105	0	local
#t2	vars	chars	106	0	local
#t3	vars	ints	107	0	local
#t4	vars	ints	108	0	local
#t5	vars	ints	109	0	local
#t6	vars	ints	110	0	local
#t7	vars	ints	111	0	local
#t8	vars	ints	112	0	local
#t9	vars	chars	113	0	local
#t10	vars	ints	114	0	local
#t11	vars	ints	115	0	local
#t12	vars	ints	116	0	local
#t13	vars	chars	117	0	local
#t14	vars	ints	118	0	local
#t15	vars	ints	119	0	local
#t16	vars	ints	120	0	local