《编译技术》课程
设计文档

学号：_____14061001_____

姓名：_____田争曦_____

2016 年 12 月 31 日

目 录

一．需求说明

1．文法说明

1)

> <加法运算符> ::= + | -
>
> <乘法运算符> ::= * | /
>
> <关系运算符> ::= < | <= | > | >= | != | ==

【分析】

这三条文法对运算符进行了说明。加法运算符可以是+或-，乘法运算符可以是*或/。关系运算符有小于等于≤，小于<，大于>，大于等于≥，不等于!=，赋值==。

【范例】

```
int a=6, b=135, c=0, d;                    //变量说明
char ch1='w', ch2="www+aaa";               //变量说明

d = a + b;
```

2)

> <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} "

【分析】

这五条文法是对字母、数组、非零数字、字符和字符串的组成说明。

字母可以是下划线_，小写字母a-z，或大写字母A-Z。数字可以是0或非零数字，其中，非零数字由1-9组成，即最终数字是0-9。

字符可以是加法运算符，乘法运算符，字母或数字，字符都需要用单引号' '圈起来表示。字符串由十进制编码为32，33，35-126的ASCII字符表示，用双引号" "圈起来，包括空格、叹号、井号等符号，数字和字母。需要注意的是，字符串中不允许再出现双引号" "了。

【范例】

```
const int a=6, b=135, c=0;                  //常量说明
const char ch1='w', ch2="www+aaa";          //常量说明
```

3)

<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>']' }

<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>'}'

【分析】

根据程序的文法可知，程序各个组成成分的声明顺序已经被限定好，不能随意更改声明顺序，如"var a,b; const x=10;"此种顺序是不允许的。

程序的主体是主函数。每一个程序按顺序由常量说明，变量说明，有返回值函数定义或无返回值函数定义，主函数组成。其中，常量说明和变量说明用[ ]括出，表示可有可无，有返回值函数定义或无返回值函数定义由{ }括出，表示该定义0-n次重复。

常量说明由const和常量定义组成，以分号;结束符。const<常量定义>可以累积，但是至少要有一个。常量定义可以是int<标识符>=<整数>，或char<标识符>=<字符>，二者均可以累积定义，若要定义多个Int类型常量或char类型常量，需要在之间加上逗号。常量由整数或字符组成。

无符号整数由非零数字组成和数字组成，数字可以0-n次重复，也就是多位数。整数由带符号的整数或0组成，带符号的整数是+或-加上无符号整数，其中，+或-可有可无。

标识符由字母组成，后面跟着字母或数字0-n次重复。

声明头部由int<标识符>或char<标识符>组成。

变量说明由变量定义组成，以分号;结束。一个变量定义后面可以继续变量定义，0-n次重复。但是至少要有一个变量定义，若有多个，中间用逗号隔开。变量定义是由类型标识符，标识符或标识符和[无符号整数]组成，类型标识

符是int或char，标识符或标识符和[无符号整数]代表单个变量或数组。其中，标识符或标识符和[无符号整数]可以0-n次重复。

有返回值函数定义由声明头部，(参数)和{复合语句}组成，()和{}均是符号不是BNF表示。无返回值函数定义由void，标识符，(参数)和{复合语句}组成，()和{}均是符号不是BNF表示。

【范例】

```
const int a=6,b=5;                  //常量说明
const char ch1='w',ch2="www";       //常量说明
int arrary[5],food[10];             //变量说明
char word[3],wods[9];               //变量说明

int count(int x,char y){            //有返回值函数定义
  ...
}

void edit(char s){                  //无返回值函数定义
  ...
}
void main(){                        //主函数
  ...
}
```

4)

> <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>

【分析】

复合语句由常量说明，变量说明和语句列组成，主体是语句列。常量说

明和变量说明可有可无，但是顺序一定不得更改。

参数的主体是参数表。参数表由类型标识符和标识符，空组成。其中，类型标识符和标识符后面可以0-n次重复，但是至少有一个，若有多个中间用逗号隔开。空表示 ε 。

【范例】

```
int count(int x,char y){          //()内为参数表
  ...
}

void edit(char s){                //()内为参数表
  ...
}
```

5)

> <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>')'

【分析】

主函数由void main(){复合语句}组成，其中()和{}均是字符不是BNF扩充。

表达式由+或-，项，加法运算符和项组成。其中，+或-可有可无，加法运算符和项是0-n次重复。

项由因子，乘法运算符和因子组成，乘法运算符和因子是0-n次重复。因

子由标识符，或标识符[表达式]，或整数，或字符，或有返回值函数调用语句，或(表达式)组成。[]和()都是字符不是BNF扩充。

【范例】

```
void main(){                              //主函数
  int a,b,c,d;
  int array[3];
  c=-a*b;
  d=--a*c;
  array[0]=c+d;
  array[1]=array[0]/a;
}
```

6)

> <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>;
>
> <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>
>
> <情况语句> ::= switch '('<表达式>')' '{'<情况表> [<缺省>] '}'
>
> <情况表> ::= <情况子语句>{<情况子语句>}
>
> <情况子语句> ::= case<常量>: <语句>
>
> <缺省> ::= default : <语句>

【分析】

语句的组成部分分别是，条件语句，或循环语句，或{语句列}，或有返回值函数调用语句，以分号;结束；或无返回值函数调用语句，以分号;结束；或赋值语句，以分号;结束；或读语句，以分号;结束；或写语句，以分号;结束；

或空，以分号;结束；或情况语句；或返回语句，以分号;结束。

赋值语句包括，标识符=表达式，标识符[表达式]=表达式。首先要判断标识符，因为只有变量才能被赋值。

条件语句包括，if(条件)语句，else语句，其中else语句可有可无。条件由两部分组成，二者为或的关系。第一种情况是表达式和表达式之间有关系运算符比较，第二种情况是只有表达式，这种情况下，表达式为0则条件为假，否则为真。

循环语句由while(条件)语句组成，()是字符不是BNF扩充。

情况语句由switch(表达式){情况表和缺省}组成，()和[]是字符不是BNF扩充，缺省可有可无。情况表由情况子语句组成，后面可以跟情况子语句的0-n次重复，但至少要有一个情况子语句。情况子语句由case常量: 语句组成。缺省由default: 语句组成。

【范例】

```
int x;
int array[10];
x=5;                          //赋值语句
array[0]=0;                   //赋值语句

if(x>3){                      //条件语句,语句列
  array[0]=array[0]+1;
  x=x-1;
}
else
  array[1]=array[0]-1;

while(x>3)                    //循环语句
```

```
  array[2]=array[0]*x;

switch(array[2]){          //情况语句
  case 5: array[3]=x;     //情况表
  case 8: array[4]=x;
  default: array[5]=x;    //缺省
}
```

7)

> <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>}
>
> <读语句> ::= scanf '('<标识符>{,<标识符>}')'
>
> <写语句> ::= printf '(' <字符串>,<表达式> ')'| printf '('<字符串> ')'| printf '('<表达式>')'
>
> <返回语句> ::= return['('<表达式>')']

【分析】

有返回值函数调用语句由标识符和(值参数表)组成，其中()是字符不是BNF扩充。无返回值函数调用语句由标识符和(值参数表)组成，其中()是字符不是BNF扩充。

值参数表由表达式或空组成，表达式后面可以跟表达式的0-n次重复，但是至少要有一个表达式，若有多个表达式，中间用逗号隔开。

语句列由语句的0-n次重复组成。

读语句由scanf(标识符)组成，标识符后面可以跟标识符的0-n次重复，但至少要有一个标识符，若有多个标识符，中间用逗号隔开。()是字符不是BNF扩

充。

写语句的组成有三部分，都以printf开头，后面可以跟(字符串,表达式)，或(字符串)，或(表达式)，()是字符不是BNF扩充。

返回语句由return和(表达式)组成，表达式可有可无。

【范例】

```
const int a=6,b=5;                  //常量说明
const char ch1='w',ch2="www";       //常量说明
int arrary[5],food[10];             //变量说明
char word[3],wods[9];               //变量说明

int count(int x,char y){            //有返回值函数定义
  ...
}

void edit(char s){                  //无返回值函数定义
  ...
}

void sum(){                          //无返回值函数定义
  ...
}
void main(){                         //主函数
  scanf(array[0]);                  //读语句
  printf(ch1,a+b);                  //写语句

  count(array[0],a);                //有返回值函数调用语句
  edit(ch2);                        //有返回值函数调用语句
  sum();                            //无返回值函数调用语句
  printf(array[0]);                 //写语句
}
```

2. 目标代码说明

在生成目标代码时，需要先组建一个临时符号表，用来存储临时变量和它们在内存中的位置。其次需要设计一个运行栈，每个函数有一个运行栈，栈顶所处位置是$sp 寄存器的所处位置，地址是 0x7fffeffc。Data 段的起始在0x00400000。为简化操作，所有的参数在内存中传递。

生成目标代码的操作大致如下文所述，更详细部分请见程序 main.cpp 第2722 行至 3971 行。

2.1 函数调用生成方案

在中间代码中，若读到第一个操作符是 push，则把参数加入参数表。

若读到的是 call，表明函数调用开始。将栈顶指针下移符号表中 offset 偏移量的大小，将需要保存的寄存器保存到内存中。保存结束后，偏移量下移到下一函数定义开始的地方。将参数按照存储顺序 sw 到对应区域，保存对应的常量，计算当前函数的偏移量大小。该偏移量大小的计算方式是，参数个数乘以4加上函数内部常量变量所需的空间大小。

下一步，跳转到函数名，生成 jal 语句。函数执行结束之后，跳转到函数调用的位置，将$sp 加上函数的偏移量，popstack。

若该函数有返回值，则把$v0 寄存器中的值赋值给该变量。至此，函数调用结束。

2.2 运算类指令生成方案

对于运算类指令来讲，首先要判断操作数是否为临时变量。如果是，则

查询参数符号表，查看表中是否有该临时变量。若有，则 lw $s1 寄存器。若无，则将 offset 加 4，分配空间，然后 lw $s1 寄存器。

若不是临时变量，先查找该函数对应的符号表，然后查找全局符号表，找到对应的地址，lw 到$s1 寄存器。

2.3 比较类指令生成方案

对于比较类指令，同样的方法把两个需要比较的操作数加载到寄存器当中，然后根据比较符号生成 OP A B label，比如 beq A B label1。

2.4 赋值语句生成方案

对于单纯赋值语句 B=1，找到 A，加载到寄存器中，找到 B 的地址，sw。

对于数组赋值语句 A[B]=C，找到 C 的地址，加载到$s1 寄存器中，找到 A 的地址，加载到$s2 寄存器中，然后

li $t1 B

mul $t2 $t2 $t1

此时$s2 中就是 A[B]的地址，sw 即可。

3. 优化方案*

void iteraeQuater()，针对连续 assign 的优化，遍历中间 diamante，对于形如下面的四元式进行合并。

```
assign var_name .factor
assign .factor .term
assign .term .expr
```

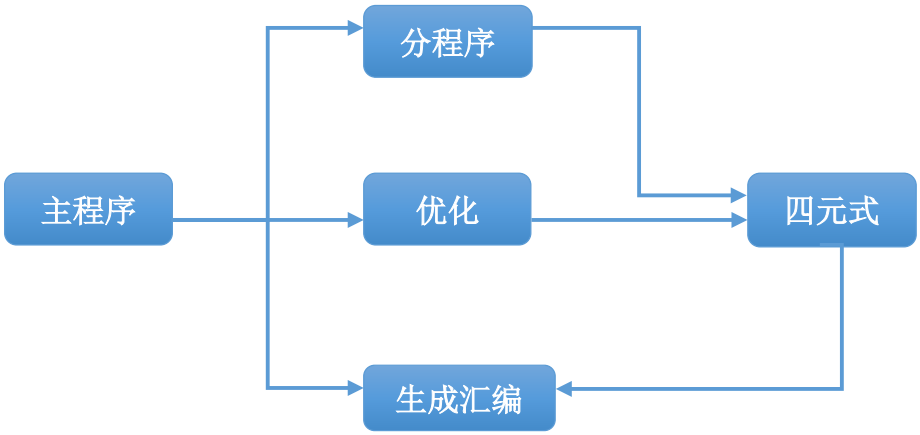
二．详细设计

1．程序结构

1.1 结构内容

程序结构由 main.cpp 和 tool.h 组成。其中，main.cpp 涵盖了词法分析，语法分析，符号表，中间代码生成，目标代码生成和对中间代码进行优化的函数。tool.h 中是 main.cpp 需要用到的工具函数，包括数字和字符串的相互转换。

1.2 结构框图



2．类/方法/函数功能

2.1 main.cpp

主函数。

2.1.1 string factor_auto( )

中间代码中因子的自动生成函数，名称为.factor+递增数字。

2.1.2 string term_auto( )

中间代码中项的自动生成函数，名称为.term+递增数字。

2.1.3 string expr_auto( )

中间代码中表达式的自动生成函数，名称为.expr+递增数字。

2.1.4 string label_auto( )

中间代码中标签的自动生成函数，名称为.label+递增数字。

2.1.5 string str_auto( )

中间代码中字符串的自动生成函数，名称为.str+递增数字。

2.1.6 string get_factor( )

得到四元式中的因子。

2.1.7 string get_term( )

得到四元式中的项。

2.1.8 string get_expression( )

得到四元式中的表达式。

2.1.9 string get_label( )

得到四元式中的标签。

2.1.10 string get_str( )

得到四元式中的字符串。

2.1.11 item_table

符号表项的结构定义。

2.1.12 void insertTable( )

插入符号表。

2.1.13 item_table* searchFunct_var(string functName, string varName)

在指定函数名的函数中查找变量，用于生成目标代码中需要查找

局部变量的地方。

2.1.14 item_table* searchGlo_var(string name)

查找全局变量，用于生成目标代码中需要查找全局变量的 地方。

2.1.15 int findMaxoff(string Functname)

在函数中找到最大的偏移量。

2.1.16 void clearTmp()

清空临时的符号表记录项。

2.1.17 void insertQuater(string op, string src1, string src2, string obj)

插入四元式。

2.1.18 void error(int i)

错误处理函数。

2.1.19 void clearQueue( )

清空读入的代码队列。

2.1.20 void clearToken( )

清空存放单词的字符串。

2.1.21 Lexical_Analysis 词法分析部分

1) **相关函数**

bool isSpace()　　//判断是否是空格

bool isNewline()　　//判断是否是回车换行

bool isTab()　　//判断是否是 tab

bool isLetter()　　//判断是否是字母

bool isDigit()　　//判断是否是数字

bool isColon()　　//判断是否是冒号

bool isComma()　　//判断是否是逗号

```
bool isSemi()          //判断是否是分号
bool isEqu()           //判断是否是等号
bool isPlus()          //判断是否是加号
bool isMinus()         //判断是否是减号
bool isDivi()          //判断是否是除号
bool isStar()          //判断是否是乘号
bool isLess()          //判断是否是小于号
bool isMore()          //判断是否是大于号
bool isSigh()          //判断是否是叹号
bool isLpar()          //判断是否是左括号
bool isRpar()          //判断是否是右括号
bool isLbrack()        //判断是否是[
bool isRbrack()        //判断是否是]
bool isLbrace()        //判断是否是{
bool isSingle()        //判断是否是单引号
bool isDouble()        //判断是否是双引号
bool catToken()        //字符串拼接
int retract()          //读字符指针后退一个
string reserver()      //检查是否为保留字
int tranNum()          //将字符串转换成整数
int print()            //打印类别编码和单词值
int getsym()           //词法分析
```

2) 关键算法

对读入的单词进行分析，读单词指针先从首字符开始，判断进入分支，通过判别比较的方法分析出单词的类型。

2.1.22 Syntax_Analysis & Mid_Code 语法分析和中间代码部分

由于中间代码（四元式）生成需要在语法分析时作出正确操作，所以这两部分结合在一起进行函数说明。

1) 相关函数

① int getsym_q()

针对于语法分析特殊情况需要多预读的 getsym 函数

② void integer()

语法分析对整数的分析

③ void constDef()

常量定义分析，＜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＞}

④ void varDef()

变量定义分析，＜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＞‘]’}

⑤ void parameter()

参数分析，＜参数＞::=＜参数表＞

⑥ void paralist()

值参数表分析，＜值参数表＞::=＜表达式＞{,＜表达式＞}｜＜

空＞

⑦ void functCall()

有返回值函数和无返回值函数调用语句分析，二者在语法分析中暂不做区分，在生成目标代码时区分。

⑧ void factor()

因子分析

⑨ void term()

项分析

⑩ void expr()

表达式分析

⑪ void condition

条件分析

⑫ void state_condition()

条件语句分析，＜条件语句＞::=if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

⑬ void loop()

循环语句分析，＜循环语句＞::=while ‘(’＜条件＞‘)’＜语句＞

⑭ void assign()

赋值语句分析

⑮ void read()

读语句分析

⑯ void write()

写语句分析

⑰ void childcase()

情况子语句分析

⑱ void caselist()

情况表分析，＜情况表＞::=＜情况子语句＞{＜情况子语句＞}

⑲ void defaultstate()

缺省分析，＜缺省＞::= default : ＜语句＞

⑳ void casestate()

情况语句分析

21 void returnstate()

返回语句分析

22 void statement()

语句分析

23 void statelist()

语句列分析

24 void compond()

复合语句分析

25 void functDef_y() & void functDef_n()

有返回值函数定义和无返回值函数定义分析

26 void funct_main()

主函数分析

27 void program()

程序分析

2) 关键算法

对输入的源代码通过递归下降进行语法分析，然后进行语义分析得到中间代码（四元式）。

2.1.22 Aimcode Analysis 生成目标代码部分

① void scanGlobal()

搜索全局的变量和常量定义，在目标代码中对应.data 后的全局常量变量定义。

② void scan_str()

搜索写语句后面的字符串，将字符串保存下来，在目标代码中对应打印字符串语句，详细说明见目标代码部分。

③ void aimcode()

生成目标代码函数。对中间代码逐一检测。检测内容有 text，begin，end，assign，const，b 类指令，算数类指令，跳转指令，set label 指令，数组赋值类指令，push，函数调用指令，返回指令，读指令，写指令。详细说明见目标代码部分。

2.1.23 优化部分

void iteraeQuater()， 针对连续 assign 的优化，遍历中间 diamante，对于形如下面的四元式进行合并。

```
assign var_name .factor
assign .factor .term
assign .term .expr
```

2.2 tool.h

此文件用于存放数字和字符串之间相互转换的函数，便于在主函数中进行调用。

3．调用依赖关系

2.1 main.cpp

main.cpp 的调用依赖关系是自顶向下的。main()函数调用了 syntax()和 aimcode()，即语法分析函数（包含中间代码生成功能）和目标代码生成函数。

syntax()函数调用了 program()函数，program()函数调用了 clearQueue()，getsym_q()， constDef()， insertQuater()， functDef_y()， functDef_n()，funct_main()，error()函数。这些函数的具体功能在函数功能部分已详细给出，这里暂不赘述。

funct_main()函数调用了 clearQueue()， getsym_q()， insertQuater()，compound()，insertTable()，clearTmp()，error()函数。

functDef_n()函数调用了 clearQueue()， getsym_q()， insertQuater()，insertTable()，clearTmp()，parameter()，compound()，error()函数。

functDef_y()函数调用了 clearQueue()， getsym_q()， insertQuater()，insertTable()，clearTmp()，parameter()，cormpound()，error()函数。

compound()函数调用了 clearQueue()，getsym_q()，varDef()，statelist()，error()函数。

statelist()函数调用了 statement()，error()函数。

statement() 函数调用了 state_condition()，loop()，clearQueue()，getsym_q()，paralist()，insertQuater()，read()，write()，casestate()，returnstate()，error()函数。

returnstate()函数调用了 clearQueue()，getsym_q()，error()，expr()，insertQuater()函数。

casestate()函数调用了 clearQueue()，getsym_q()，error()，expr()，label_auto()，expr_auto()，defaultstate()，insertQuater()函数。

defaultstate()函数调用了 clearQueue()，getsym_q()，error()，statement()函数。

caselist()函数调用了 insertQuater()，label_auto()，childcase()函数。

childcase()函数调用了 clearQueue()，getsym_q()，error()，insertQuater()，statement()函数。

write()函数调用了 clearQueue()，getsym_q()，insertQuater()，expr()，expr_auto()，searchFunct_var()，searchGlo_var，insertTable()，clearTmp()，error()函数。

expr()函数分析表达式的情况，调用了 term()项分析函数，而 term()项分析函数则调用了 factor()因子分析函数。三个函数关系依次嵌套。

2.2 tool.h

无调用依赖关系。

4. 词法分析器

4.1 扩充 C0 文法保留字表

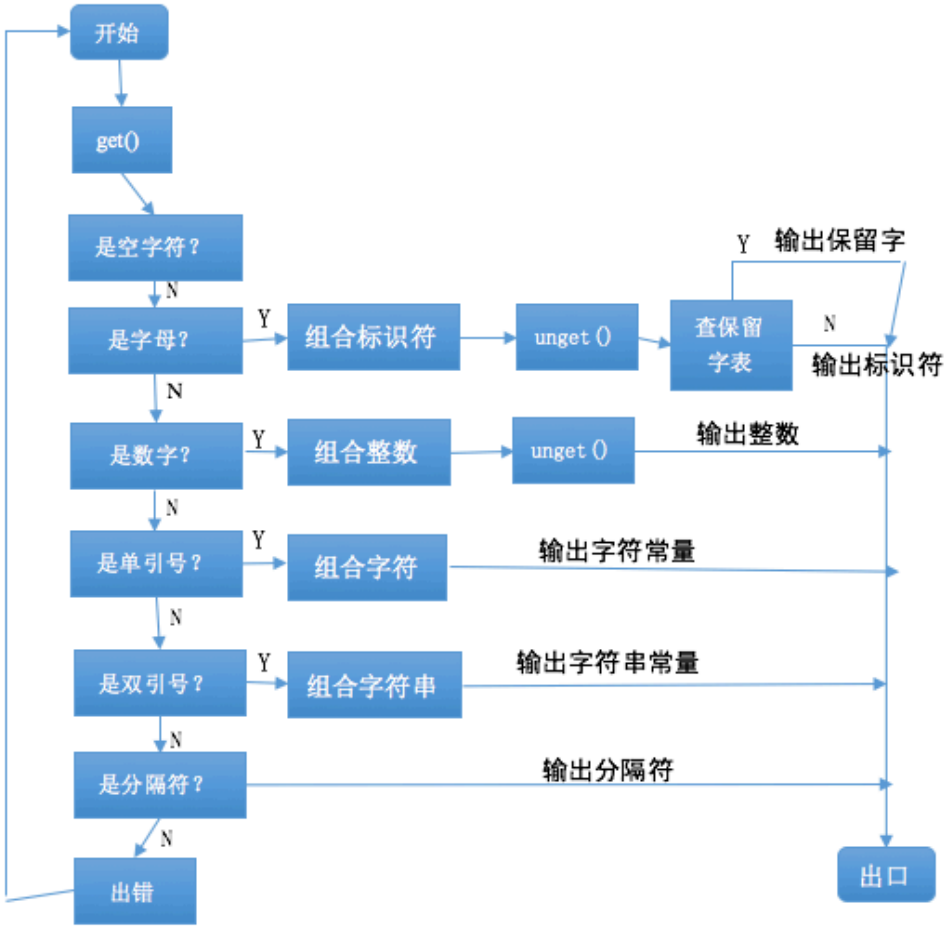
if	else
int	char
printf	scanf
while	const
switch	case
void	main
default	return

4.2 单词类别码

单词名称	类别编码	单词值	单词名称	类别编码	单词值
标识符	IDENT	内部字符串表示	<	LESSSY	<
无符号整数	UNSIGNED	整数	<=	LESSEQSY	<=
字符串常数	CHARS	字符串值	>	MORESY	>
字符常数	SYM	字符值	>=	MOREEQSY	>=
if	IF	if	!=	NEQSY	!=
else	ELSE	else	==	EQUSY	==
int	INT	int	:	COLON	:
char	CHAR	char	;	SEMI	;

printf	PRINTF	printf	,	COMMA	,
scanf	SCANF	scanf	(	LPARENT	(
while	WHILE	while	)	RPARENT	)
const	CONST	const	[	LBRACK	[
switch	SWITCH	switch	]	RBRACK	]
case	CASE	case	{	LBRACE	{
void	VOID	void	}	RBRACE	}
main	MAIN	main	=	ASSIGN	=
default	DEFAULT	default	*	MULSY	*
return	RETURN	return	/	DIVSY	/
+	ADDSY	+	-	MINUSSY	-

4.3 算法设计



5. 语法分析器

语法分析器采用递归下降分析方法，对文法中每一个非终结符逐个分析。每一个非终结符编写为一个函数，在函数内部完成相应文法的语法成分分析和识别。

根据《编译技术》教材，在采用递归下降分析方法时，应先消除左递归。

我抽到的 C0 扩充文法中已经提取了左因子，不存在左递归现象。故可以直接进入下一步的分析。

语法分析只要对每一个非终结符逐一分析，进入判断条件即可，唯一需要解决的是回溯问题，这里采取预读字符的方法解决。

例如，变量定义和有返回值函数定义需要读到第三个字符才能判断出，该条语句的语法归属。所以，在 functDef_y()和 functDef_n()函数中，处理成已经假设预读了两个字符的情况，即在前两次 getsym 之后队列指针不回退。这样，在每次调用这两个函数之前，要事先预读两个字符。

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

类似的还有 funct_main()函数，处理为已经预读了 3 个字符，所以其他函数若调用该函数时，需要先预读两个字符。

6．符号表管理方案

符号表采用一个符号表进行统一管理，不增加索引表。符号表定义为一个结构数组。

具体见下图。

```
/* symbol item 符号表项 */
struct item_table{
    string name;
    type_table type;
    int size;
    string value;           //for constant
    string functBelong;     //belong to which function, if is a local
variable
    int position;
    int address;            //reserved address
    int offset;
    bool isArray;
    bool isGlobal;
    bool isConst;
    bool isFunct;
    int parNum;
};
item_table tmp_tab;
vector <item_table> symTab;
```

6.1 符号表的结构与内容

① 名字（标识符）的种类：如简单变量、函数、过程、数组、标号、参数等。

② 类型：如整型，实型，字符型，指针等。一般用编码表示。

③ 大小：所占的字节数。

④ 值：标识符所代表的数值，仅针对于常量。

⑤ 所属函数名：仅针对于局部变量。

⑥ 地址：标识符所分配单元的保留地址，只对于全局变量。

⑦ 位移量：标识符所分配单元的偏移量，只对于局部变量。

⑧ 是否是数组。

⑨ 是否是全局的。

⑩ 是否是常量。

⑪ 是否是函数。

⑫ 参数个数：函数中的参数个数。

6.2 符号表的数据结构

```
/* type */
enum type_table{
  VOID,
  INT,
  CHAR,
  PAR_INT,
  PAR_CHAR,
  ERR
};

/* symbol item 符号表项 */
struct item_table{
 string name;
 type_table type;
  int size;
 string value;            //for constant
 string functBelong;      //belong to which function,
if is a local variable
 int position;
 int address;             //reserved address
 int offset;
 bool isArray;
 bool isGlobal;
 bool isConst;
 bool isFunct;
 int parNum;
};
```

6.3 符号表管理

6.3.1 插入符号表

① 如插入的为函数，在符号表中查找是否存在同名函数。若存在，

则插入失败；若不存在，则将其插入符号表。

② 如插入的不为函数，先查全局再查局部，设立一个 isGlobal 的 bool 类型全局变量来标注遇到的常变量属性。在全局中查找只需要查看该变量是否是否被定义过，若被定义过，则报重复定义的错误，若未被定义，则插入符号表。在局部中查找，要在变量或常量所属的函数名下的符号表进行查找，设立一个 functName 的全局变量，用来记录当前所处的函数的函数名，若在该函数中未找到，插入符号表，否则，报重复定义的错误。

6.3.2 查找符号表

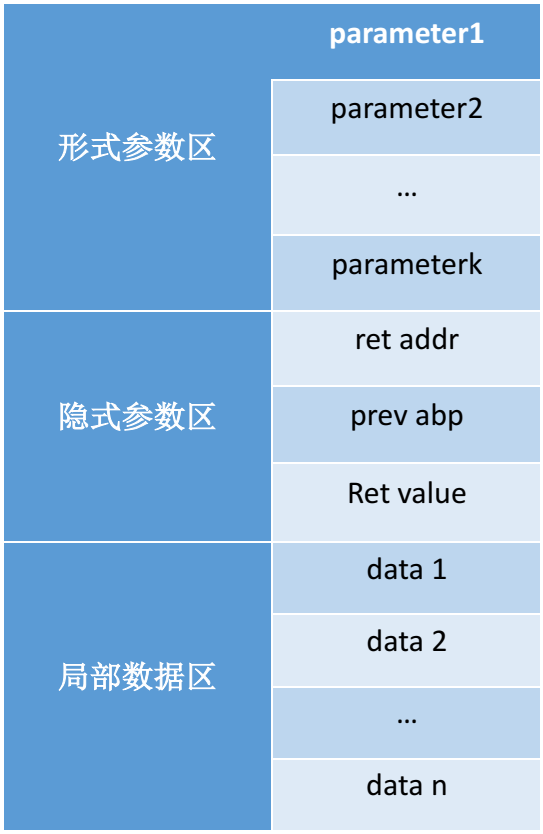
① 在全局中查找：

```
/* search global variable */
item_table* searchGlo_var(string name){
    vector<item_table>:: iterator i;
    for(i=symTab.begin(); i!=symTab.end(); i++){
        if(name == i->name && i->functBelong=="")
            return &*i;     //如果找到了，重复定义，返回这个item_table
    }
    return nullptr;         //如果没找到，返回空指针
}
```

② 在局部中查找：

```
/* search variable in function */
item_table* searchFunct_var(string functName, string varName){
    vector<item_table>:: iterator i;
    for(i=symTab.begin(); i!=symTab.end(); i++){
        if(functName == i->functBelong && varName == i->name){
            return &*i;     //如果找到了，重复定义，返回这个item_table
        }
    }
    return nullptr;         //如果没找到，返回空指针
}
```

7. 存储分配方案



8. 解释执行程序*

无解释执行程序。

9. 四元式设计*

类型	编号	四元式	说明
算数运算	0	add rs rt rd	加法：rd=rs+rt

	1	sub rs rt rd	减法：rd=rs-rt, rs 可以是 0
	2	mul rs rt rd	乘法：rd=rs*rt
	3	div rs rt rd	除法：rd=rs/rt
比较运算	5	lt rs rt label	如果 rs<rt，跳转
	6	let rs rt label	如果 rs<=rt，跳转
	7	bt rs rt label	如果 rs>rt，跳转
	8	bet rs rt label	如果 rs>=rt，跳转
	9	beq rs rt label	如果 rs==rt，跳转
	10	bne rs rt label	如果 rs!=rt，跳转
跳转	11	j rs _ _	无条件跳转
读写	13	read rt _	读 rt
	14	write rt note _	写 rt，str 字符串对于标识符：int 整数，char 字符

定义	15	const int/char value name	定义常量
	16	int/char _ _ name	定义变量
	17	array int/char len name	定义数组
	18	global	标志全局常量/变量
	19	text	text 段开始
赋值	20	assign rt _ rd	把 rt 赋值给 rd
	21	[]= rs rt rd	rd[rs] = rt
	22	=[] rs rt rd	rt = rd[rs]
调用函数	23	call funct _ _	调用无返回值 funct 函数
	24	callre rs _ funct	函数 funct 的返回值存在 rs 中
参数进栈	25	push rs _ _	把 rs 进栈
设置 label	26	set label _ _	设置 label

返回语句	27	return rs _ _	返回 rs，rs 可有可无
函数开始	28	begin funct _ _	函数开始
函数结束	29	end funct _ _	函数结束

10. 目标代码生成方案*

在生成目标代码时，需要先组建一个临时符号表，用来存储临时变量和它们在内存中的位置。其次需要设计一个运行栈，每个函数有一个运行栈，栈顶所处位置是$sp 寄存器的所处位置，地址是 0x7fffeffc。Data 段的起始在 0x00400000。为简化操作，所有的参数在内存中传递。

生成目标代码的操作大致如下文所述，更详细部分请见程序 main.cpp 第 2722 行至 3971 行。

10.1 函数调用生成方案

在中间代码中，若读到第一个操作符是 push，则把参数加入参数表。

若读到的是 call，表明函数调用开始。将栈顶指针下移符号表中 offset 偏移量的大小，将需要保存的寄存器保存到内存中。保存结束后，偏移量下移到下一函数定义开始的地方。将参数按照存储顺序 sw 到对应区域，保存对应的常量，计算当前函数的偏移量大小。该偏移量大小的计算方式是，参数个数乘以 4 加上函数内部常量变量所需的空间大小。

下一步，跳转到函数名，生成 jal 语句。函数执行结束之后，跳转到函数调用的位置，将$sp 加上函数的偏移量，popstack。

若该函数有返回值，则把$v0 寄存器中的值赋值给该变量。至此，函数调用结束。

10.2 运算类指令生成方案

对于运算类指令来讲，首先要判断操作数是否为临时变量。如果是，则查询参数符号表，查看表中是否有该临时变量。若有，则 lw $s1 寄存器。若无，则将 offset 加 4，分配空间，然后 lw $s1 寄存器。

若不是临时变量，先查找该函数对应的符号表，然后查找全局符号表，找到对应的地址，lw 到$s1 寄存器。

10.3 比较类指令生成方案

对于比较类指令，同样的方法把两个需要比较的操作数加载到寄存器当中，然后根据比较符号生成 OP A B label，比如 beq A B label1。

10.4 赋值语句生成方案

对于单纯赋值语句 B=1，找到 A，加载到寄存器中，找到 B 的地址，sw。

对于数组赋值语句 A[B]=C，找到 C 的地址，加载到$s1 寄存器中，找到 A 的地址，加载到$s2 寄存器中，然后

li $t1 B

mul $t2 $t2 $t1

此时$s2 中就是 A[B]的地址，sw 即可。

11. 优化方案*

11.1 基本块划分算法

输入：中间代码语句序列（四元式序列)。

输出：基本块序列，每条中间代码属于且仅属于一个基本块。

① 首先确定入口语句的集合。

a. 整个语句序列的第一条语句属于入口语句。

b. 任何能由条件/无条件跳转语句转移到第一条语句属于入口语句。

c. 紧跟在跳转语句之后的第一条语句属于入口语句。

② 每个入口语句直到下一个入口有，或者程序结束，它们之间的所有语句都属于同一个基本块。

11.2 通过构建 DAG 图消除局部公共子表达式算法

输入：基本块内的中间代码。

输出：完成局部公共子表达式删除后的 DAG 图。

① 首先建立结点表，该表记录了变量名和常量值，以及它们当前 所对应的 DAG 图中的结点的序号。该表初始状态为空。

② 从第一条中间代码开始，按照以下规则建立 DAG 图。

③ 对于形如 z=x op y 的中间代码，其中 z 为记录计算结果的变量名，

x 为左操作数，y 为右操作数，op 为操作符；首先在节点表中寻找 x，如果找到，记录下 x 当前所对应的节点号 i；如果未找到，在 DAG 图中新建一个叶节点，假设其节点号仍为 i，标记为 x（如 x 为变量名，则该标记更改为 x0）；在节点表中增加新的一项（x,i），表明二者之间的对应关系。右操作数 y 与 x 同理，假设其对应节点号为 j。

④ 在 DAG 图中寻找中间节点，其标记为 op，且其左操作数节点号为 i，右操作数节点号为 j。如果找到，记录下其节点号 k；如果未找到，在 DAG 图中新建一个中间节点，假设节点号仍为 k，并将节点 i 和 j 分别与 k 相连，作为其左子节点和右子节点。

⑤ 在节点表中寻找 z，如果找到，将 z 所对应的节点号更改为 k；如果未找到，在节点表中新建一项（z,k），表明二者之间的对应关系。

⑥ 对输入的中间代码序列依次重复上述步骤③-⑤。

11.3 从 DAG 图导出中间代码的启发式算法

输入：DAG 图

输出：中间代码序列

① 初始化一个放置 DAG 图中间节点的队列。

② 如果 DAG 图中还有中间节点未进入队列，则执行步骤③，否则执行步骤⑤。

③ 选取一个尚未进入队列，但其所有父节点均已进入队列的中间节点 n，将其加入队列；或选取没有父节点的中间节点，将其加入

队列。

④ 如果 n 的最左子节点符合步骤③的条件，将其加入队列；并沿着当前节点的最左边，循环访问其最左子节点，最左子节点的最左子结点等，将符合步骤③条件的中间节点依次加入队列；如果出现不符合步骤③条件的最左子结点，执行步骤②。

⑤ 将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列。

12. 出错处理

将出错信息进行分类，对每一种出错信息进行编码并报告。简单的语法错误，会显示正确的应为什么。复杂的语法错误，会进行跳读处理，，汇报错误信息，恢复继续进行检测。语义错误，汇报出错信息，恢复继续进行检测。

12.1 错误列表

错误编号	错误信息	出错原因
1	Can’ t find integer	找不到整数
2	Undeclared identifier	未声明的标识符
3	Identifier expected	此处应为标识符

4	Char expected	此处应为字符
5	Type identifier expected	此处应为类型标识符
6	[expected	此处应为[
7	] expected	此处应为]
8	Array must have size	数组应定义大小
9	Expression expected	此处应为表达式
10	(expected	此处应为(
11	) expected	此处应为)
12	Factor expected	此处应为因子
13	Term expected	此处应为项
14	If expected	此处应为 if
15	While expected	此处应为 while
16	= expected	此处应为=
17	Scanf expected	此处应为 scanf

18	Printf expected	此处应为 printf
19	: expected	此处应为:
20	Constant expected	此处应为常量
21	Case expected	此处应为 case
22	Defaullt expected	此处应为 default
23	Switch expected	此处应为 switch
24	{ expected	此处应为{
25	} expected	此处应为}
26	Return expected	此处应为 retur
27	Statement expected	此处应为语句
28	; expected	此处应为;
29	Void expected	此处应为 void
30	Illegal integer	不合法的整数
31	Illegal character	不合法的字符

32	End of a line	读到文件行末尾
33	Dupilicate definition	重复定义
34	Cant find definition	未定义
35	Function cnnot be assigned	函数名不能被赋值
36	The number of parameters does not match	参数个数不匹配
37	Scanf: Input invalid	Scanf 内容无效
38	Void function mustn't return a value	Void 函数返回了一个值

三．操作说明

1．运行环境

编译器编写环境：Codeblocks。目标代码运行环境：Mars

2．操作步骤

安装 Codeblocks。安装完成之后运行 Codeblocks。

点击 File->Open，找到编译器的项目工程文件，打开。

点击编译并运行按钮，弹出控制台，将测试文件直接拖入控制台，显示

测试文件的路径。

程序显示：Please set the optimization on-off. On:1. Off:other key.

若选择对中间代码进行优化，则输入 1。等待程序执行。

运行程序，控制台会输出对于程序的语法分析。若程序中有不符合文法的错误，会在控制台中进行报错。

四．测试报告

1．测试程序及测试结果

1.1 测试打印全覆盖 ASCII 码值的字符串

```
void test(){
      printf("+-*/ !%#$&,().-
/0123456789:;<>=?@ABCDEFGHIJKLMNOPQRSRUVWXYZabcdEFGHIJKLMNOPQR
STUVWXYZ");
}

void main()
{
      int choice;
      scanf(choice);
      if(choice<9){
            switch(choice){
                  case 7: test();
                  default: printf("overflow");
            }
      }
}
```

输入：7

输出：

+-*/!%#$&,().-

/0123456789:;<>=?@ABCDEFGHIJKLMNOPQRSRUVWXYZabcdEFGHIJKLMNOP

QRSTUVWXYZ

1.2 测试数值比较和 case-switch 语句

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

void test(){
      scanf(i);
      scanf(j);
      if(i<j)
            printf("i<j");
      if(i==j)
            printf("i=j");
      if(i>j)
            printf("i>j");
}

void main()
{
      int choice;
      scanf(choice);
      if(choice<9){
            switch(choice){
                  case 5: test();
                  default: printf("overflow");
            }
      }
}
```

输入：5

回车换行后输入 10 10

输出：i=j

输入：5

回车换行后输入-1 10

输出：i<j

输入：5

回车换行后输入 99 10

输出：i>j

1.3 测试数值运算和打印

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

void test(){
      const int _var_X=123,_var_Y=456;
      printf(_var_X+_var_Y);
      printf("\n");
      printf(_var_X-_var_Y);
      printf("\n");
      printf(_var_X*_var_Y);
}

void main()
{
      int choice;
      scanf(choice);
      if(choice<9){
            switch(choice){
                  case 6: test();
                  default: printf("overflow");
            }
      }
}
```

输入：6

输出：

579

-333

56088

1.4 测试递归

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

int fibonacci (int n){
    if(n==1) return (1);
    if(n!=2) return (+fibonacci(n+-1)+fibonacci(n-2)+0);
    return (1);
}

void main()
{
     int choice;
     scanf(choice);
     if(choice<10){
          switch(choice){
               case 9:
                {
                     scanf(number);
                     fibo = fibonacci(number);
                     printf(fibo);
                }
                default: printf("overflow");
          }
     }
}
```

输入：9

回车换行后，输入：7

输出：13

输入：9

回车换行后，输入：4

输出：3

1.5 正确测试程序五

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

int compute(){
      int n;
      scanf(tmp);
      n=a+b*c-(tmp+3)*2;
      printf(n);
}

void main()
{
      int choice;
      scanf(choice);
      if(choice<10){
            switch(choice){
             case 2: compute();
                  default: printf("overflow");
            }
      }
}
```

输入：2

回车换行后，输入：3

输出：-9

1.6 错误程序 1

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

void test(){
     printf("+-*/                                          !%#$&,().-
/0123456789:;<>=?@ABCDEFGHIJKLMNOPQRSRUVWXYZabcdEFGHIJKLMNOPQR
STUVWXYZ");
}

void main()
{

     scanf(choice);
     if(choice<9){
          switch(choice){
               case 7: test();
               default: printf("overflow");
          }
     }
}
```

错误原因：Undeclared identifier. 未声明的标识符。

1.7 错误程序 2

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

void test(){
    scanf(i);
    scanf(j);
    if(i<j)
        printf("i<j");
    if(i==j)
        printf("i=j");
```

```
    if(i>j)
        printf("i>j");
}

void main()
{
    int choice;
    scanf(choice);
    if(choice<9){
        switch(choice){
            case 5: test()
            default: printf("overflow");
        }
    }
}
```

错误原因：; expected.缺少分号。

1.8 错误程序 3

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

void test(){
    const int _var_X=123,_var_Y=456;
    printf(_var_X+_var_Y);
    printf("\n");
    printf(_var_X-_var_Y);
    printf("\n");
    printf(_var_X*_var_Y);
    return (1);
}

void main()
{
    int choice;
    scanf(choice);
    if(choice<9){
```

```
        switch(choice){
            case 6: test();
            default: printf("overflow");
        }
    }
}
```

错误原因：Void function mustn't return a value.无返回值函数返回了一个值。

1.9 错误程序 4

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

int fibonacci (int n){
    if(n==1) return (1);
    if(n!=2) return (+fibonacci(n+-1)+fibonacci(n-2)+0);
    return (1);
}

void main()
{
    int choice;
    scanf(choice);
    if(choice<10){
        switch(choice){
            case 9:
            {
                scanf(number);
                fibo = fibonacci(number,number);
                printf(fibo);
            }
            default: printf("overflow");
        }
    }
}
```

错误原因：The number of parameters does not match.参数个数不匹配。

1.10 错误程序 5

```
const int a=+1, b=2, c=1;
const char ch1='Y',ch2='N';

int i,j,tmp,fibo,number;
int array[10];
char word[10];

      int compute(){
    int n;
    scanf(tmp);
    n=a+b*c-(tmp+3)*2;
    printf(n);
}

void main()
{
    int choice;
    scanf(choice);
    if(choice<10){
        switch(choice){
            case 2 compute();
            default: printf("overflow");
        }
    }
}
```

错误原因：] expected. 缺少右中括号。

2．测试结果分析

上文的测试程序较全面的覆盖了:

全局常量定义，全局变量定义，临时常量定义，临时变量定义，函数传值，函数递归调用，赋值语句，基本运算，大小关系比较，switch-case 语句，数组元素赋值，全局常量和变量的引用等。

覆盖率大范围地覆盖了文法。

五．总结感想

编译课程设计这门课程带给了我不小的收获。首先，通过自己动手设计一个编译器，将原理课上许多抽象难懂的概念付诸于实践，可以更系统地理解编译器的整体结构和设计过程。

其次，这门课程提高了我的代码能力、调试能力和抗压能力，最为重要的是，它真正教会我如何全面的思考。在语法分析部分经常会出现逻辑的漏洞，都是通过不断的调试和逻辑的完善，才能向前一步。

在进行编译课程设计生成中间代码和目标代码这两个最难最关键步骤的同时，我们还面临着数据库最终 project，移动计算的安卓 App 开发，软件工程，大数据 Hadoop 的大作业。刚开始做如此庞大的工作量的时候，内心是十分崩溃的，而编译还遇到了工作量最大最难的部分。经过了连续一周在新主楼刷夜，经常四五点钟回到宿舍，早上起来还要继续第二天的课程，对于身体的消耗极大。好在功夫不负有心人，最终在经历前三次测试的时候效果都十分理想。

在进行编译器设计期间，我发现在正式写代码之前，一定要有一个清晰的构思，这样写代码的效率和正确率都十分可观。从词法分析开始，每次进行编程之前，我都会在笔记本上写下这一部分的逻辑框图，在大脑清晰的条件下尽可能地思考全面需要注意的问题，和易出现的错误，往往会起到事半功倍的效果。

在编译课程设计期间，也十分感谢我的室友们。在与她们讨论的过程中，我规避了一些易出现的问题。并且，在容错处理阶段也做到了尽可能得全面详尽。

最后的一点遗憾是，由于编译器最终的优化阶段正值各科的期末考试，

时间十分有限，所以最后优化只做了很少一部分，不足以达到申优的标准，内心十分遗憾。希望在编译课程结束之后，可以向做好优化的同学请教他们的思路，在寒假继续完成。