姓名：白文强　　学号：20191060064　专业：计算机科学与技术

一、实验目的

(1) 理解编译程序的基本逻辑过程；

(2) 理解作用域管理和符号表的组织与实现；

(3) 理解活动记录的基本设计，过程的调用和返回的实现；

二、实验内容

1. 实现减法(-)、除法(/)（40%）

将 expr 的定义扩展为：

expr→expr + term　　expr - expr　term

term→term * factor　　term / factor　　factor

2. 实现 while 语句（40%）

while 循环的语法可以通过 stmt 增加如下候选式：

stmt→while bexpr do stmt

其中，bexpr 为真执行 do 后面的语句，为假退出循环。

3、实现数组（20%）

将 vardecl 和 variable 的定义扩展为如下的规则：

vardef→ id | id[num: num]

variable→id | id[expr]

例如，int a[2:5]定义一个下标从 2 到 5 的 4 个元素的整型数组。数组元素可以在表达式中被引用。

三、实验分析与设计

（一）实现减法与除法

乘除法“*”，“/”和加减法“+”，“-”满足左结合，且乘除优先于加减法。将一开始的 expr 定义改写为如下的产生式：

expr→expr + term | term

term→term * factor | factor

factor→(expr) | variable | num | letter | true | false

为了实现减法与除法，需要将 expr 增加两项产生式，即：

expr→expr – term

term→term / factor

为了完成这两项功能，需要在单字符的符号表 skind 中添加减号(-)与除号(/)，在分析 expr 的程序中，需要增加判断当前 token 是否为减号即 subtoken，若是，则要使用 gettoken()获取下一个 token，分析 term 语法，一直循环，直到通过 gettoken()的 token 的类型不再是 subtoken。

在分析 term 时，需要增加判断当前 token 是否为除号即 divtoken，若是，则要使用 gettoken()获取下一个 token，随后进行 factor 的分析，一直循环，直到当前 token 的两类型不再是 divtoken。

（二）实现 while 语句

while 语句的文法如下：

stmt→while bexpr do stmt。

翻译模式：

stmt→while bexpr do stmt1

{

bexpr_rep=addrLabel(NewLabel());

stmt_next=addLabel(NewLabel());

stmt.code=(LAB, bexpr_rep, NULL,NULL)

|| bexpr.code

|| (JPC, bexpr.addr, NULL, stmt_next)

|| stmt1.code

```
        || (JUMP, NULL, NULL, bexpr_rep)
        || (LAB, stmt_next, NULL, NULL)
}
```

为了实现对 while 语句的分析，需要在 reservedword 中增加 while 与 do 两个保留关键字，将符号分别设置为 whiletoken 与 dotoken。

在分析 stmt 文法时，需要增加判断当前 token 是否为 whiletoken，如果是，则进入 WHILEstmt()函数对 while 语句进行分析，函数声明如下。

```
int WHILEstmt(int lev, int *tx, int *off)
```

在函数中，使用 gettoken()获取一个新的 token，随后进入 bexpr 的语法分析程序，并返回一个具有中间代码分量的结构的结构体指针。由于 while 的条件语句的值必须为布尔类型，即值为 true 或 false。对返回的结构体的值进行判断，如果其中的 type 字段不是布尔类型，那么则表示 while 后的条件语句是不可判断真假的类型，无法进行后面的运算，说明该语句出现问题，此时要进行报错处理。

分析过 bexpr 后，接下来需要判断当前 token 的类型，while 语句中必须包含 do 关键字，如果当前 token 不是 do，那么则说明，while 语句缺少 do 语句，进行报错处理。

如果当前 token 是 do，则需要继续对 stmt 进行分析。

随后，需要设置两个标号，分别表示 while 语句入口以及 while 语句结束后跳转地址，接着进行中间代码生成工作。

（三）实现数组

为了实现数组，需要添加一个标识符类型 array 来表示该符号是数组类型，同时，为了支持数组的定义，还需要增加两个符号‘[’和‘]’，分别用 lbracket 和 rbracket 表示。

由于数组变量需要存储起始下标，因此需要对符号表进行修改，增加起始下标与结束下标：start_index, end_index

```
struct tablestruct
{
    char name[al];          //符号的名字
```

```
        enum idform form;     //标识符的类型  ,可以通过符号的数据类型进行区分
        enum datatype type; //符号的数据类型
        int level;              //符号所在的层
        int address;            //符号的地址
        int start_index;
        int end_index;
    };
```

同时增加插入 array 变量的函数，enter_array()，将数组变量插入符号表时，调用该函数完成，而不是原来的 enter()，在 enter_array()中，大部分与 enter()函数相同，只是增加了根据数组元素的个数计算存储地址偏移的功能。

```
if (tk == inttype)
    (*off) = (*off) + 2 * (end - start + 1);
else if (tk == booltype || tk == chartype)
    (*off) = (*off) + 1 * (end - start + 1);
```

由于数组的名字也是一个 id，为了判断该 id 是不是数组类型的变量，需要在 vardef()函数中进行修改。需要多读一个 token，判断其是否是 lbracket 即‘[’，如果是，则说明该 id 是数组，否则是普通变量。

采用多读一个 token 来判断该 id 是不是数组变量的方式具有一个缺陷是难以将代码中缺少左中括号即‘[’的情况查找出来，我的解决方式是如果多读的 token 不是‘[’，在 else 语句中判断该 token 是否为 num 类型，如果是，则说明数组定义或访问缺少‘[’。

如果是数组，将 form 设置为 array，接下来对‘num:num]’进行分析，如果出现错误，则进行报错。否则，使用 enter_array()函数将该符号插入到符号表中。

对数组元素进行访问时，由于数组的下标不是从 0 开始，而是由定义时指定，那么对应 array 指针中的地址的计算方式为：array_index = (i – start_index) * type_width。

在对数组元素进行赋值时，通过 addrvar()函数获取的内容是元素的地址、

类型等信息，在代码运行时，需要通过地址找到该元素在存储单元中的位置，因此，需要修改 sourceOperandGen()与 thirdOperandGen()函数，增加对数组类型的判断，在其中需要对寄存器 bp 进行保护，即将其压入栈中。

四、实现与结果分析

（一）减除的实现

用编译器对下面的程序进行分析与运行：

```
main()
{   int a,b;
    read(a,b);
    b=2+3;
    {
        int x,y;
        if a<b-2 then x=5+6/3; else x=6;
        write(x,y);
    }
    return;
}
```

```
请输入分析的文件名:minus_div.txt
List intermediate code?(Y/N)y
Optimize the intermediate code?(Y/N)y
Run object code?(Y/N)y
the identifiers in block 0:
main type=    form=procedure  lev=0 address=1 start=0 end=0
--------------
the identifiers in block 1:
a type=int  form=variable  lev=1 address=0 start=0 end=0
b type=int  form=variable  lev=1 address=2 start=0 end=0
--------------
the identifiers in block 2:
x type=int  form=variable  lev=2 address=4 start=0 end=0
y type=int  form=variable  lev=2 address=6 start=0 end=0
--------------
intermediate code:
[0]  (JUMP,-,-,L1)
[1]  (MENTRY,9,8,L1)
[2]  (READ,-,-,a)
[3]  (READ,-,-,b)
[4]  (ADD,2,3,T1)
[5]  (ASS,T1,-,b)
[6]  (SUB,b,2,T2)
[7]  (LTC,a,T2,T3)
[8]  (JPC,T3,-,L2)
[9]  (DIV,6,3,T4)
[10]  (ADD,5,T4,T5)
[11]  (ASS,T5,-,x)
[12]  (JUMP,-,-,L3)
[13]  (LAB,-,-,L2)
[14]  (ASS,6,-,x)
[15]  (LAB,-,-,L3)
[16]  (WRITE,-,-,x)
[17]  (WRITE,-,-,y)
[18]  (RET,-,-,-)
start SIMPLE!
Input an int value: 2
Input an int value: 3
output an int value:  7
output an int value:  0
```

（二）对 while 语句的实现

用编译器对下面的程序进行分析与运行：

```
main()
{
```

```
    int a,b;
    read(a,b);
    {
       while a<b do a=a+1;
       write(a,b);
    }
    return;
}
```

该程序实现的功能是：如果 a 是小于 b 的，那么将 a 每次加 1，直到 a 与 b 相等。

```
请输入分析的文件名:while.txt
List intermediate code?(Y/N)y
Optimize the intermediate code?(Y/N)y
Run object code?(Y/N)y
the identifiers in block 0:
main type=    form=procedure  lev=0 address=1 start=0 end=0
--------------
the identifiers in block 1:
a type=int  form=variable  lev=1 address=0 start=0 end=0
b type=int  form=variable  lev=1 address=2 start=0 end=0
--------------
the identifiers in block 2:
--------------
intermediate code:
[0]  (JUMP,-,-,L1)
[1]  (MENTRY,3,4,L1)
[2]  (READ,-,-,a)
[3]  (READ,-,-,b)
[4]  (LAB,-,-,L2)
[5]  (LTC,a,do,T1)
[6]  (JPC,T1,-,L3)
[7]  (ADD,a,1,T2)
[8]  (ASS,T2,-,a)
[9]  (JUMP,-,-,L2)
[10]  (LAB,-,-,L3)
[11]  (WRITE,-,-,a)
[12]  (WRITE,-,-,b)
[13]  (RET,-,-,-)
start SIMPLE!
Input an int value: 1
Input an int value: 6
output an int value:  6
output an int value:  6
```

（三）对数组的实现

使用编译器对下面的程序进行分析:

```
main ()
{   int x[1:3];

    read(x[1]);
    read(x[2]);
    read(x[3]);
```

```
    write(x[1],x[2],x[3]);
    return;
}
```

该程序实现的功能是：输入一个数组，并将其输出。

```
请输入分析的文件名:array.txt
List intermediate code?(Y/N)y
Optimize the intermediate code?(Y/N)y
Run object code?(Y/N)y
the identifiers in block 0:
main type=    form=procedure  lev=0 address=1 start=0 end=0
--------------
the identifiers in block 1:
x type=int  form=variable  lev=1 address=0 start=1 end=3
--------------
intermediate code:
[0]  (JUMP,-,-,L1)
[1]  (MENTRY,36,6,L1)
[2]  (SUB,1,1,T1)
[3]  (MUL,T1,2,T2)
[4]  (ADD,0,T2,T3)
[5]  (READ,-,-,T3)
[6]  (SUB,2,1,T4)
[7]  (MUL,T4,2,T5)
[8]  (ADD,0,T5,T6)
[9]  (READ,-,-,T6)
[10]  (SUB,3,1,T7)
[11]  (MUL,T7,2,T8)
[12]  (ADD,0,T8,T9)
[13]  (READ,-,-,T9)
[14]  (SUB,1,1,T10)
[15]  (MUL,T10,2,T11)
[16]  (ADD,0,T11,T12)
[17]  (WRITE,-,-,T12)
[18]  (SUB,2,1,T13)
[19]  (MUL,T13,2,T14)
[20]  (ADD,0,T14,T15)
[21]  (WRITE,-,-,T15)
[22]  (SUB,3,1,T16)
[23]  (MUL,T16,2,T17)
[24]  (ADD,0,T17,T18)
[25]  (WRITE,-,-,T18)
[26]  (RET,-,-,-)
start SIMPLE!
Input an int value: 1
Input an int value: 2
Input an int value: 3
output an int value:  1
output an int value:  2
output an int value:  3
```

```
[6]  SUB ax ax bx
[7]  ST ax 8 bp
[8]  LD ax 8 bp
[9]  LDC bx 2
[10]  MUL ax ax bx
[11]  ST ax 10 bp
[12]  LDC ax 0
[13]  LD bx 10 bp
[14]  ADD ax ax bx
[15]  ST ax 12 bp
[16]  LDC ax 2
[17]  PUSH bp
[18]  PUSH ax
[19]  LD ax 12 bp
[20]  ADD bp bp ax
[21]  POP ax
[22]  ST ax 0 bp
[23]  POP bp
[24]  LDC ax 10
[25]  ST ax 6 bp
[26]  LD ax 6 bp
[27]  LDC bx 5
[28]  SUB flag ax bx
[29]  JNG 75
[30]  LD ax 6 bp
[31]  LDC bx 7
[32]  SUB flag ax bx
[33]  JNL 55
[34]  LDC ax 1
[35]  LDC bx 1
[36]  SUB ax ax bx
[37]  ST ax 14 bp
[38]  LD ax 14 bp
[39]  LDC bx 2
[40]  MUL ax ax bx
[41]  ST ax 16 bp
[42]  LDC ax 0
[43]  LD bx 16 bp
[44]  ADD ax ax bx
[45]  ST ax 18 bp
[46]  LDC ax 0
[47]  PUSH bp
[48]  PUSH ax
```

```
[48]  PUSH ax
[49]  LD ax 18 bp
[50]  ADD bp bp ax
[51]  POP ax
[52]  ST ax 0 bp
[53]  POP bp
[54]  JUMP 75
[55]  LDC ax 1
[56]  LDC bx 1
[57]  SUB ax ax bx
[58]  ST ax 20 bp
[59]  LD ax 20 bp
[60]  LDC bx 2
[61]  MUL ax ax bx
[62]  ST ax 22 bp
[63]  LDC ax 0
[64]  LD bx 22 bp
[65]  ADD ax ax bx
[66]  ST ax 24 bp
[67]  LDC ax 1
[68]  PUSH bp
[69]  PUSH ax
[70]  LD ax 24 bp
[71]  ADD bp bp ax
[72]  POP ax
[73]  ST ax 0 bp
[74]  POP bp
[75]  LDC ax 1
[76]  LDC bx 1
[77]  SUB ax ax bx
[78]  ST ax 26 bp
[79]  LD ax 26 bp
[80]  LDC bx 2
[81]  MUL ax ax bx
[82]  ST ax 28 bp
[83]  LDC ax 0
[84]  LD bx 28 bp
[85]  ADD ax ax bx
[86]  ST ax 30 bp
[87]  PUSH bp
[88]  LD ax 30 bp
[89]  ADD bp bp ax
[90]  LD ax 0 bp
[91]  POP bp
[92]  OUT ax 0
[93]  MOV top bp
```

```
[88]  LD ax 30 bp
[89]  ADD bp bp ax
[90]  LD ax 0 bp
[91]  POP bp
[92]  OUT ax 0
[93]  MOV top bp
[94]  POP bp
[95]  POP pc
start SIMPLE!
output an int value:  1

Process returned 0 (0x0)   execution time : 8.208 s
Press any key to continue.
```

由此可见，编译器对减法、除法、while 语句、数组都实现了正确的编译和运行。