《编译技术》课程设计文 档

学号：\_\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_\_\_\_\_

2016年 1月 7日

目录

[一．需求说明 2](#_Toc440106943)

[1．文法说明 2](#_Toc440106944)

[（1）获取的文法 2](#_Toc440106945)

[（2）对文法的改写和扩充 2](#_Toc440106946)

[2．目标代码说明 3](#_Toc440106947)

[3. 优化方案\* 5](#_Toc440106948)

[二．详细设计 5](#_Toc440106949)

[1．程序结构 5](#_Toc440106950)

[2．类/方法/函数功能 6](#_Toc440106951)

[3．调用依赖关系 8](#_Toc440106952)

[4．符号表管理方案 9](#_Toc440106953)

[5．存储分配方案 10](#_Toc440106954)

[6. 解释执行程序\* 10](#_Toc440106955)

[7. 四元式设计\* 10](#_Toc440106956)

[8. 目标代码生成方案\* 11](#_Toc440106957)

[9. 优化方案\* 12](#_Toc440106958)

[（1）基本块内部的公共子表达式删除（DAG图） 12](#_Toc440106959)

[（2）全局寄存器分配（活跃变量分析，建立冲突图，着色算法） 12](#_Toc440106960)

[（3）合理利用临时寄存器（临时寄存器池） 12](#_Toc440106961)

[10. 出错处理 13](#_Toc440106962)

[三．操作说明 13](#_Toc440106963)

[1．运行环境 13](#_Toc440106964)

[2．操作步骤 13](#_Toc440106965)

[四．测试报告 14](#_Toc440106966)

[1．测试程序及测试结果 14](#_Toc440106967)

[2．测试结果分析 20](#_Toc440106968)

[五．总结感想 20](#_Toc440106969)

## 一．需求说明

### 1．文法说明

（1）**获取的文法**

扩充PL/0文法（无实型），具体文法见文档结尾附录。

（2）对文法的改写和扩充

对原文法改写的原则是（1）消除二义性（2）简化非终结符的关系，方便用递归下降子程序法编写语法、语义分析程序。

|  |  |
| --- | --- |
| 原文法规则 | 改写的文法规则或规定 |
| <常量定义>  ::=    <标识符>＝ ~~<常量>~~ ~~<常量>~~     ::=    [+| -] <无符号整数>|<字符> | 合并为：<常量定义>  ::=    <标识符>＝ [+| -] <无符号整数>|<字符> |
| <语句>     ::=   <赋值语句>|<条件语句>|<当循环语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for循环语句>|<空>  <复合语句> ::=   begin<语句>{; <语句>}end | 规定<空>只能出现在复合语句的最后一个子句上，即复合语句最后一个子句后面可以有分号，也可以没有分号。 |
| <赋值语句>   ::=  <标识符> := <表达式>| ~~<函数标识符>~~ := <表达式> | <标识符>'['<表达式>']':= <表达式> ~~<函数标识符>~~  ::=  <标识符> | 合并为：<赋值语句> ::=  <标识符>['['<表达式>']']:= <表达式>  <标识符>的语义由符号表协助处理 |
| <因子>       ::=   <标识符>|<标识符>'['<表达式>']'|<无符号整数>| '('<表达式>')' | ~~<函数调用语句>~~  ~~<函数调用语句>~~  ::=   <标识符>[<实在参数表>]  <实在参数表>  ::= '(' ~~<实在参数>~~ {, ~~<实在参数>~~}')' ~~<实在参数>~~ ::=   <表达式> | 合并为：<因子>       ::=   <标识符>['['<表达式>']'|<实在参数表>]|<无符号整数>| '('<表达式>')'  <实在参数表>  ::= '(' <表达式> {, <表达式>}')'  <标识符>的语义由符号表协助处理 |

其他规定或实现细节如下：

（1）char类型的变量或常量，在表达式中用字符的ASCII码对应的整数参加运算，表达式都为整型，不能赋值给字符型变量，否则报错，函数或过程传递参数时类型不一致也报错。

（2）形式参数段中变量类型只能为基本类型，不能是数组名。

（3）for循环语句步长为1，

若for i:=1 to 10  ，则循环执行10次，i的结果应该是11  
               若for i:=1 downto 10   ，则循环不执行，  i应该被赋值为1

（4）read语句连续读入多个项目时，以回车分隔；write语句每输出一个项目，跟一个回车。

（5）标识符区分大小写字母

（6）数组的下标从0开始

### 2．目标代码说明

要生成的目标代码是32位X86汇编代码。代码生成时用到的主要指令及含义列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 操作码 | 操作数1 | 操作数2 | 含义 |
| mov | R1 | R2 | R1=R2 |
| add | R1 | R2 | R1=R1+R2 |
| sub | R1 | R2 | R1=R1-R2 |
| inc | R1 |  | R1=R1+1 |
| dec | R1 |  | R1=R1-1 |
| neg | R1 |  | R1=-R1 |
| imul | R1 | R2 | R1=R1\*R2 |
| idiv | R1 |  | eax=[edx:eax]/R1 |
| cdq |  |  | 将eax的符号位扩展到edx |
| jmp | L |  | 无条件转移至L |
| je | L |  | 若相等，转移至L |
| jne | L |  | 若不相等，转移至L |
| jg | L |  | 若大于，转移至L |
| jge | L |  | 若大于等于，转移至L |
| jl | L |  | 若小于，转移至L |
| jle | L |  | 若小于等于，转移至L |
| cmp | R1 | R2 | R1-R2,改变标志寄存器 |
| push | R1 |  | R1入栈 |
| pop | R1 |  | 弹出栈顶元素给R1 |
| pushad |  |  | 将通用寄存器入栈 |
| popad |  |  | 弹出栈顶元素给通用寄存器 |
| pushf |  |  | 标志寄存器入栈 |
| popf |  |  | 弹出栈顶元素给标志寄存器 |
| call | A |  | 调用过程A |
| ret | n |  | 过程返回，同时出栈n个元素 |

另外，还要用到一些伪指令，主要是数据定义伪指令，包括BYTE、WORD、DWORD、DUP、PROC、NEAR32、ENDP、END、ADDR、OFFSET等。

生成的汇编程序文件结构如下：

.386

.model flat, stdcall

option casemap : none

include \masm32\include\windows.inc

include \masm32\include\user32.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

include \masm32\include\msvcrt.inc

includelib \masm32\lib\user32.lib

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

includelib \masm32\lib\msvcrt.lib

.STACK 4096

.DATA

……

.CODE

……

\_start:

……

push 0

call ExitProcess

END \_start

其中.386说明程序使用80386处理器指令集，.model flat,stdcall说明内存模式为平坦模式，语言模式为stdcall，option casemap : none说明不区分大小写，过程ExitProcess用于结束程序。

生成的整个汇编程序由三部分组成，.STACK部分为堆栈，用于充当运行栈；.DATA部分定义了一些存放在内存上的数据；.CODE部分是程序的主要代码，首先是最内层的PL/0源程序翻译成的代码，往后层次依次降低，最后是最外层的源程序翻译成的代码。

### 3. 优化方案\*

需要完成的优化要求有：

（1）基本块内部的公共子表达式删除，方案是通过将中间代码转换成DAG图，再从DAG图导出中间代码；

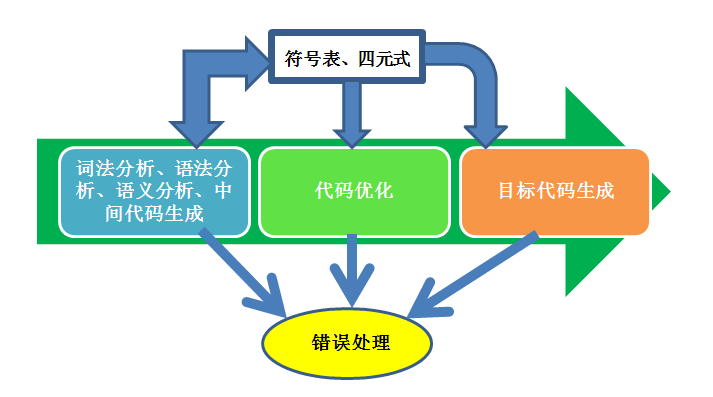
（2）数据流分析，方案是活跃变量分析或利用定义-使用链建网来构建冲突图；

（3）全局寄存器分配，方案是引用计数发或图着色算法；

（4）合理利用临时寄存器，方案是在代码生成时使用临时寄存器池分配临时寄存器。

## 二．详细设计

### 1．程序结构



程序结构大体如上图所示，采用一遍扫描方式。其中语法分析、语义分析和中间代码生成一起进行，采用递归下降子程序法实现，调用词法分析程序进行工作。由语法分析部分生成的符号表在语义分析、中间代码生成部分查询，由于是栈式符号表，故在一遍分析生成中间代码完成后符号表就不存在了，其中的定义信息转移到了四元式中，供代码优化和代码生成程序使用。程序的各个部分均进行不同类型的错误检查并调用错误处理程序输出错误信息。

### 2．类/方法/函数功能

由于函数过多，这里只列举功能主要的函数。

与语法语义分析、中间代码生成有关的函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void getsym() | 读单词 |
| void constdeclaration() | 分析常量声明 |
| void vardeclaration() | 分析变量声明 |
| void procdeclaration() | 分析过程声明 |
| void funcdeclaration() | 分析函数声明 |
| void argument\_list(int index\_0) | 分析实参列表 |
| void factor(ITEM &x) | 分析因子 |
| void term(ITEM &x\_1) | 分析项 |
| void expression(ITEM &x1) | 分析表达式 |
| void assign\_state(int ident\_index\_1) | 分析赋值语句 |
| void if\_state() | 分析条件语句 |
| void do\_while\_state() | 分析当循环语句 |
| void for\_loop\_state() | 分析for循环语句 |
| void read\_state() | 分析读语句 |
| void write\_state() | 分析写语句 |
| void composition\_state() | 分析复合语句 |
| void statement() | 分析语句 |
| void sub\_program() | 分析分程序 |
| void program(char \*filename) | 分析程序 |

与符号表、过程表、字符串表有关的函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| int lookup(char \*str) | 逆序查表，若查到返回索引，否则返回-1 |
| void set\_item(KIND kind, TYPE type) | 向符号表加入一个表项 |
| void copy\_item(ITEM &x1,ITEM x2) | 复制一个符号表表项 |
| void set\_bitem() | 向过程表加入一个表项 |
| void set\_sitem() | 向字符串表加入一个表项 |

与四元式有关的函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void gen\_code(int prog\_level,OPERATOR opt,ITEM op1,ITEM op2,ITEM op3) | 产生一条四元式 |
| void initial\_cx() | 初始化个层次四元式索引 |
| void get\_maxl() | 得到分程序最大层次 |

错误处理函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void error(char\* errmsg) | 输出错误位置和错误信息，统计错误个数 |
| void jump() | 跳读至下一个逗号、分号或程序结尾 |

与优化有关的函数、

|  |  |
| --- | --- |
| 函数名 | 功能 |
| int bblocks\_dividing\_1(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE],int lev, int start, int end) | 划分基本块函数1 |
| void bblocks\_dividing\_2(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE],int lev, int max\_index) | 划分基本块函数2 |
| void Q2DAG(int lev,int start,int end) | 四元式导出DAG图 |
| void DAG2Q(int lev) | DAG图导出四元式 |
| void get\_DefUse(int lev,int procNo,int max\_bblock\_index) | 计算def[B]和use[B] |
| void LiveVariableAnalysis(int lev,int procNo,int max\_blocks\_index,int procStart,int procEnd) | 进行活跃变量分析，得到冲突图 |
| void ColorDistribution(int lev,int procNo) | 用图着色算法进行全局寄存器分配 |
| void optimize() | 优化主函数 |

与目标代码生成有关的函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void gen\_asmfile(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE], int cx[MAX\_LEVEL + 1]) | 生成目标程序文件 |

与debug有关的函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void print\_token(); | 打印词法分析结果 |
| void print\_table(); | 打印当前栈式符号表 |
| void print\_btab(); | 打印过程表 |
| void print\_stab(); | 打印字符串表 |
| void print\_qfmt(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE], int cx[MAX\_LEVEL + 1]); | 打印四元式 |

### 3．调用依赖关系

void program(char \*filename)调用以下函数，

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void getsym() | 读单词 |
| void constdeclaration() | 分析常量声明 |
| void vardeclaration() | 分析变量声明 |
| void procdeclaration() | 分析过程声明 |
| void funcdeclaration() | 分析函数声明 |
| void argument\_list(int index\_0) | 分析实参列表 |
| void factor(ITEM &x) | 分析因子 |
| void term(ITEM &x\_1) | 分析项 |
| void expression(ITEM &x1) | 分析表达式 |
| void assign\_state(int ident\_index\_1) | 分析赋值语句 |
| void if\_state() | 分析条件语句 |
| void do\_while\_state() | 分析当循环语句 |
| void for\_loop\_state() | 分析for循环语句 |
| void read\_state() | 分析读语句 |
| void write\_state() | 分析写语句 |
| void composition\_state() | 分析复合语句 |
| void statement() | 分析语句 |
| void sub\_program() | 分析分程序 |

void optimize()调用以下函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| int bblocks\_dividing\_1(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE],int lev, int start, int end) | 划分基本块函数1 |
| void bblocks\_dividing\_2(QFRT code[MAX\_LEVEL + 1][2 \* MAX\_SOURCE\_FILE],int lev, int max\_index) | 划分基本块函数2 |
| void Q2DAG(int lev,int start,int end) | 四元式导出DAG图 |
| void DAG2Q(int lev) | DAG图导出四元式 |
| void get\_DefUse(int lev,int procNo,int max\_bblock\_index) | 计算def[B]和use[B] |
| void LiveVariableAnalysis(int lev,int procNo,int max\_blocks\_index,int procStart,int procEnd) | 进行活跃变量分析，得到冲突图 |
| void ColorDistribution(int lev,int procNo) | 用图着色算法进行全局寄存器分配 |
| void optimize() | 优化主函数 |

int main(int argc, const char \*argv[])调用void program(char \*filename)和void optimize()

### 4．符号表管理方案

符号表由结构数组实现，结构体中包含标识符名称（name），属性（kind），类型（type），层次(lev)，若为常量则须填值（val），变量和参数须填其在活动记录中的相对地址（val），函数或过程须填过程表索引（val），数组须填元素类型(array\_type)和个数(array\_len)。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | name | kind | type | lev | val | array\_type | array\_len |
| 1 |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |

typedef enum{

K\_NULL=0, CONST, VAR, PROC, FUNC, PARA, VPARA,

STRING, IMM, REG, TEMP

} KIND;

typedef enum{

NUL=0,INT , CHAR, ARRAY

} TYPE;

typedef struct {

char name[MAX\_IDENT\_LEN];

KIND kind;

TYPE type;

int lev;

int val;

TYPE array\_type;

int array\_length;

} ITEM;

符号表的栈式结构和向上查表的方法支持内层变量覆盖外层变量，每当分程序结束时弹出该分程序的符号表。

此外，本程序中还设计了过程表用于存放过程的信息，包括局部变量个数，形参个数，临时变量个数，参数类型等信息；字符串表用于存放字符串和标号。这些表格是静态的，在栈式符号表消失后，为代码生成提供必要的信息。

### 5．存储分配方案

本编译器在目标程序的.DATA段维护DISPLAY区，每次过程调用自动更新对应层的DISPLAY地址值，故活动记录中不需存DISPLAY区的信息。ebp指向当前活动记录基地址，esp指向栈顶，运行栈（X86的堆栈向下（低地址）生长）结构如下：

|  |
| --- |
| 返回值 |
| pre\_ebp |
| 实参区 |
| 返回地址 |
| 8个通用寄存器和标志寄存器（寄存器保护） |
| 局部变量区 |
| 临时变量区 |
|  |

在函数或过程调用前先预留出返回值位置，将当前ebp入栈，传入实参，然后进行call语句，压入返回地址和所有寄存器，再留出局部变量和临时变量的空间。函数返回时的过程相反，先将返回值填至返回值的位置，再释放保存的寄存器，将pre\_ebp送给ebp，进行ret语句。

### 6. 解释执行程序\*

高难度不需要写此项。

### 7. 四元式设计\*

四元式格式及含义见下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 操作符(operator) | 操作数1  (operand1) | 操作数2  (operand2) | 操作数3  (operand3) | 功能说明  (notation) |
| assign | A | B |  | 赋值(：=):A=B |
| add | A | B | C | 有符号加(+):A=B+C |
| sub | A | B | C | 有符号减(-):A=B-C |
| mul | A | B | C | 有符号乘(\*):A=B\*C |
| div | A | B | C | 有符号整数除法(/)：A=B/C |
| neg | A | B |  | 取反:A=-B |
| arg | A |  |  | 传入实参A |
| call | A | (B) |  | 调用过程A或调用函数A，返回值存入B |
| proc | A |  |  | 声明过程A |
| ret | A |  |  | 过程返回，A为过程 |
| arrayl | A | B | C | 左值数组([]=):A[B]=C |
| arrayr | C | A | B | 右值数组(=[]):C=A[B] |
| write | A | (B) |  | 将A(和B)输出到控制台 |
| read | A |  |  | 从控制台读入字符存入A |
| cmp | A | B |  | A-B |
| jmp | LABEL |  |  | 无条件转移到标号LABEL处 |
| je | LABEL |  |  | 等于(=)则转移 |
| jne | LABEL |  |  | 不等于(<>)则转移 |
| jg | LABEL |  |  | 大于(>)则转移 |
| jge | LABEL |  |  | 大于等于(>=)则转移 |
| jl | LABEL |  |  | 小于(<)则转移 |
| jle | LABEL |  |  | 小于等于(<=)则转移 |
| loopup | A | B | LABEL | 若A自增1，若小于等于B，转移至LABEL处 |
| loopdown | A | B | LABEL | 若A自减1，若大于等于B，转移至LABEL处 |
| label | LABEL |  |  | 在当前语句处设置标号LABEL |

typedef enum{ //操作符

null=0, assign, add, sub, mul, div, neg, arg, call, push, pop, proc, ret,

arrayl, arrayr, write,read,cmp,jmp, je,jne, jg, jge,jl,jle,label,loopup,loopdown

} OPERATOR;

typedef struct{ //四元式操作数

KIND kind;

TYPE type;

int lev;

int val;

}OPERAND;

typedef struct{ //四元式

OPERATOR opt;

OPERAND operand[3];

}QFRT;

### 8. 目标代码生成方案\*

先生成内层的中间代码，再生成外层的中间代码，这样可以保证过程在调用之前被声明。目标代码生成部分没有特殊的数据结构，主要是四元式操作符和X86汇编指令的对应和翻译，唯一有些麻烦的就是变量的寻址问题，由于可以调用外层变量，于是需要先计算好变量的地址，引用传递参数时也要计算好地址，特别是使用外层的引用参数时，计算有些复杂。

### 9. 优化方案\*

（1）基本块内部的公共子表达式删除（DAG图）

这是所做优化中唯一会改变中间代码的优化，保证正确性很重要。基本块内部的公共子表达式删除包括由四元式生成DAG图和由DAG图导出四元式两方面的算法，具体算法步骤和书上基本一致，这里不再赘述。再次仅说明两点不同。

首先是书上由四元式生成DAG图的算法中，赋值符号是不产生结点的，只改变相应变量对应的结点序号，这样做在遇到连续的赋值语句（比如交换两个数的代码）时得不到正确的结果，因此，我的实现方法是将赋值运算符也产生结点，但是其右孩子（而不是像取反操作符一样是左孩子）是第二个操作数，由此能够解决所遇到的局部公共子表达式删除问题，能够保证正确性。

第二点是由DAG图产生四元式的时候，对应同一结点有多个变量，选择哪一个作为操作数是一个问题。我的做法是选用任意一个该结点对应的临时变量，而当有其他局部变量存在时，在生成的四元式末尾统一为所有未赋值的局部变量赋值，以保证正确性。

（2）全局寄存器分配（活跃变量分析，建立冲突图，着色算法）

采用活跃变量分析建立局部变量之间的冲突图，然后用图着色算法分配全局寄存器

这里选用ebx,esi,edi作为全剧寄存器进行分配。

首先进行活跃变量分析，书上的算法如下，用数组的一位表示集合的一个元素，进行集合之间的运算非常方便。

•输入：程序流图，且基本块的use集合和def集合已经计算完毕

•输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]

•方法：

1.将包括代表流图出口基本块Bexit在内的所有基本块的in集合，初始化为空集。

2.根据方程out[B] = ∪B的后继基本块P in[P]，in[B] = use[B]∪ (out[B] – def[B])，为每个基本块B依次计算集合out[B]和in[B]。如果计算得到某个基本块的in[B]与此前计算得出的该基本块in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止。

构建冲突图，这里采用了较粗略的方法：当局部变量在同一个基本块的入口处活跃时，认为它们冲突。

采用图着色的启发式算法进行全局寄存器分配。实现也比较简单。

（3）合理利用临时寄存器（临时寄存器池）

这里选用eax,edx作为临时寄存器进行分配。

• 进入基本块时，清空临时寄存器池

• 为当前中间代码生成目标代码时，无论临时变量还是局部变量（亦或全局变量和静态变量），如需使用临时寄存器，都可以向临时寄存器池申请

• 临时寄存器池接到申请后，

– 如寄存器池中有空闲寄存器，则可将该寄存器标识为被该

申请变量占用，并返回该空闲寄存器

– 如寄存器池中没有空闲寄存器，则选取一个在即将生成代

码中不会被使用的寄存器写回相应的内存空间，标识该寄

存器被新的变量占用，返回该寄存器

• 在基本块结尾，或者函数调用发生前，将寄存器池中所有被占用的临时寄存器写回相应的内存空间，

清空临时寄存器池

### 10. 出错处理

错误检查在词法分析、语法分析、语义分析和中间代码生成的过程中进行，不同具体情况下跳读的部分不尽相同，跳读的最大距离是：声明部分出错后跳至下一个逗号处，其他部分出错后跳至下一个分号处，每次出错时调用错误处理函数error，直接输出错误信息和错误位置（行号、列号）。

void error(char\* errmsg){

printf("LEN:%d COL:%d %s\n", line, pos-line\_start+1, errmsg);

errnum++;

}

## 三．操作说明

### 1．运行环境

(1)Visual Studio 2013 Express

(2)masm32v11r

### 2．操作步骤

（1）在VS2013 Express中打开工程，运行，按控制台提示输入源文件路径和文件名，然后控制台会输出编译信息（是否通过编译、发现错误的位置和文字描述），程序运行结束。（生成的汇编程序12171060.asm存放在C:\目录下）

（2）按下WIN+R，键入”cmd”，打开命令提示符，键入”cd C:\masm32\bin”切换到编译链接程序所在路径。

（3）分别键入”ml –c –coff C:\12171060.asm”和”link –subsystem:console 12171060.obj”对汇编程序进行编译和链接。

（4）键入”12171060”运行生成的可执行程序12171060.exe。

版本二

修改环境变量path=\*;c:\masm32\bin

（1）在VS2013 Express中打开工程，运行，按控制台提示输入源文件路径和文件名，然后控制台会输出编译信息（是否通过编译、发现错误的位置和文字描述），程序运行结束。（生成的汇编程序12171060.asm存放在C:\目录下）

（2）按下WIN+R，键入”cmd”，打开命令提示符，键入”C:”切换当前路径到C盘。

（3）分别键入”ml –c –coff 12171060.asm”和”link –subsystem:console 12171060.obj”对汇编程序进行编译和链接。

（4）键入”12171060”运行生成的可执行程序12171060.exe。

## 四．测试报告

### 1．测试程序及测试结果

【正确程序1】

const SIZE1=8,SIZE2=-3,TRUE='T',FALSE='F';

var a,b,c:integer;

function opt(p:integer):integer;

var x:integer;

begin

x:=a+p;

opt:=x+6

end;

begin

a:=2;

b:=a+3;

c:=opt(b);

write(c)

end.

【测试结果1】

输出13

【正确程序2】

const SIZE=8,TRUE='T',FALSE='F';

var i,k:integer;

K:array[50]of integer;

procedure Quick(s,t:integer);

var i,j:integer;

procedure Swap(var x,y:integer);

var temp:integer;

begin

temp:=x;

x:=y;

y:=temp

end;

begin

if s<t

then begin

i:=s;

j:=t+1;

do

begin

do

i:=i+1

while (K[s]-K[i])\*(t-i)>0;

do

j:=j-1

while (K[j]-K[s])\*(j-s)>0;

if i<j

then Swap(K[i],K[j])

end

while i<j;

Swap(K[s],K[j]);

Quick(s,j-1);

Quick(j+1,t)

end

end;

function Check:char;

var i:integer;

begin

Check:=TRUE;

for i:=0 to SIZE-2 do

if K[i]>K[i+1]

then Check:=FALSE

end;

begin

for i:=0 to SIZE-1 do

begin

read(k);

K[i]:=k

end;

if Check=TRUE

then write("The input array is already in ascending order:")

else begin

Quick(0,SIZE-1);

write("The ordered array is:")

end;

for i:=0 to SIZE-1 do

begin

write(" ",K[i])

end

end.

【测试结果2】

输入 98 0 -3 89 -13 0 3 -7

输出 The ordered array is:-13 -7 -3 0 0 3 89 98

【正确程序3】

const MAX=4;

var i,j,k:integer;

K:array[10]of integer;

begin

k:=0;

for i:=1 to MAX do

begin

k:=k+i

end;

write(k);

for j:=20 downto k do

begin

write(j)

end;

write(j+10);

i:=67;

for i:=10 to 3 do

begin

write("errer!")

end;

write(i)

end.

【测试结果3】

输出10 20 19 18 17 16 15 14 13 12 11 10 19 67

【正确程序4】

const K='k',A='a',B='b';

var a:char;

b,c:integer;

begin

a:=K;

b:=3;

if b-3>0

then a:=A

else

a:=B;

write(a)

end.

【测试结果4】

输出b

【正确程序5】

var a:array[4]of integer;

begin

a[0]:=3;

a[1]:=4;

a[2]:=a[0]+8;

a[3]:=a[0]+a[1];

write("array");

write(a[0]);

write(a[2]);

write(a[3])

end.

【测试结果5】

输出 array 3 11 7

【错误程序1】

var x:char;

y:integer;

function foo(b:char;c:integer):char;

begin

foo:=a+57

end;

begin

x:=foo(y,y);

x:=foo();

x:=foo(x);

if y=1 then

x:=foo(x,y,x)

else

y:=0

end.

【测试结果6】

标识符未定义（因子），

实参与形参类型不一致，

实参表括号内没有参数，

实参个数过少

实参个数过多

【错误程序2】

const a=3,begin=4,c=24,a=7,d='p';

begin

write(d)

end.

【测试结果7】

常量标识符非法，

同层常量标识符重复定义

【错误程序3】

const MAX=4;

var i,j,k:integer;

K:array[10]of integer;

begin

k:=0;

for i:=1 to MAX do

begin

k:=k+i

end;

write(k);

for j:=20 down k do

begin

write(j)

end;

write(j+10);

i:=67;

for i:=10 to 3 do

begin

write("errer!")

end;

write(i);

end.

【测试结果8】

for循环语句中缺少关键字to或者downto

【错误程序4】

var a,b:char;

begin

a='s';

write(b);

end.

【测试结果9】

赋值语句中赋值符号写成等号

【错误程序5】

const SIZE=-7;

var A:array[-2]of char;

B:array[10]of integer;

x:integer;

C:array[x+2]of integer;

begin

x:=2+'a'+6;

B[10]:=12;

x:=B[-4];

x:=B[SIZE];

SIZE:=13;

write("d")

end.

【测试结果10】

数组上界应为无符号整数

数组上界应为无符号整数

因子不能为字符

数组的整数或常量下标越界（数组元素为左值）

数组的整数或常量下标越界（数组元素为右值）

不能改变常量的值

因子不能为字符

### 2．测试结果分析

|  |  |
| --- | --- |
| 测试程序 | 语法成分和语义覆盖情况 |
| 【正确程序1】 | 赋值语句，函数调用语句，有返回值函数调用，上层变量调用 |
| 【正确程序2】 | 覆盖所有语法成分，递归调用，变量形参 |
| 【正确程序3】 | for循环语句（to,downto），写语句，常量声明，复合语句 |
| 【正确程序4】 | if条件语句，常量定义，写语句 |
| 【正确程序5】 | 数组元素引用，写语句(字符串，表达式)，赋值语句 |

## 五．总结感想

这次编译课设是我第一次写由许多文件组成的大型程序，充分锻炼了编程能力。同时我也体会到，写大型文件时组织清晰的重要性。在一些编译原理算法的实现上，我也体会到具体实践时，还是会碰到一些和书本上不一样的问题，需要自己想办法去解决。总的来说，此次编译课设的时间有些紧张，主要还是我编程不熟练造成的。

另外，我也深切的体会到了提前对程序有个总体设计的重要性。我一开始写编译器时思路有些不清楚，优化也没有仔细设计，导致编程时遇到许多问题，对于自己实现的方法没有了信心，好多东西还得推倒重来，浪费了很多时间。

其实一些上课讲的编译算法当时没有理解透彻，自己实现，自己去思考了之后才有所掌握，比如临时寄存器分配，上课只会讲一个小例子，而我们自己实现时，却要考虑到多种情况和正确性问题。这门课给我的感受就是，只有真正自己实现了一个编译器，才会对编译有所了解。

附录：【获取的文法】

<程序>       :: =   <分程序>.  
<分程序>      ::=   [<常量说明部分>][<变量说明部分>]{[<过程说明部分>]| [<函数说明部分>]}<复合语句>  
<常量说明部分>  ::=   const<常量定义>{,<常量定义>};  
<常量定义>     ::=    <标识符>＝ <常量>  
<常量>      ::=    [+| -] <无符号整数>|<字符>  
<字符>         ::=  '<字母>' | '<数字>'  
<字符串>       ::= "{十进制编码为32,33,35-126的ASCII字符}"  
<无符号整数>   ::=   <数字>{<数字>}  
<标识符>      ::=   <字母>{<字母>|<数字>}  
<变量说明部分>  ::=   var <变量说明> ; {<变量说明>;}  
<变量说明>    ::=   <标识符>{, <标识符>} : <类型>  
<类型>      ::=   <基本类型>|array'['<无符号整数>']' of <基本类型>  
<基本类型>    ::=   integer | char   
<过程说明部分>  ::=  <过程首部><分程序>{; <过程首部><分程序>};  
<函数说明部分>  ::=  <函数首部><分程序>{; <函数首部><分程序>};  
<过程首部>    ::=   procedure<标识符>[<形式参数表>];  
<函数首部>    ::=   function <标识符>[<形式参数表>]: <基本类型>;  
<形式参数表>    ::= '('<形式参数段>{; <形式参数段>}')'  
<形式参数段>     ::=   [var]<标识符>{, <标识符>}: <基本类型>     
<语句>  ::=   <赋值语句>|<条件语句>|<当循环语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for循环语句>|<空>  
<赋值语句>    ::=  <标识符> := <表达式>| <函数标识符> := <表达式> | <标识符>'['<表达式>']':= <表达式>  
<函数标识符>      ::=  <标识符>   
<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>']'|<无符号整数>| '('<表达式>')' | <函数调用语句>  
<函数调用语句>  ::=   <标识符>[<实在参数表>]  
<实在参数表>   ::= '(' <实在参数> {, <实在参数>}')'  
<实在参数>   ::=   <表达式>   
<加法运算符>   ::=   +|-  
<乘法运算符>   ::=   \*|/  
<条件>       ::=   <表达式><关系运算符><表达式>  
<关系运算符>  ::=   <|<=|>|>= |=|<>  
<条件语句>     ::=   if<条件>then<语句> | if<条件>then<语句>else<语句>  
<当循环语句>  ::=   do<语句> while<条件>  
<for循环语句>  ::=   for <标识符>  := <表达式> （downto | to） <表达式> do <语句> //步长为1  
<过程调用语句>  ::=  <标识符>[<实在参数表>]  
<复合语句>  ::=   begin<语句>{; <语句>}end  
<读语句>     ::=   read'('<标识符>{,<标识符>}')'  
<写语句>    ::=   write'('<字符串>,<表达式>')'|write'('<字符串>')'|write'('<表达式>')'  
<字母>     ::=   a|b|c|d…x|y|z |A|B…|Z  
<数字>      ::=   0|1|2|3…8|9

附加说明

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符区分大小写字母

（3）赋值语句中<函数标识符> := <表达式> 作为函数的返回值，其类型应与返回类型一致，此语句后面的语句可继续执行

（4）写语句中的字符串原样输出，表达式只有单个字符类型的变量或常量按字符输出，其他表达式均按整型输出

（5）数组的下标从0开始

（6）带var的参数为变量形参，实参与该类形参传递数据时是传地址