《编译技术》课程设计文 档

学号： 14231006

姓名： 田宇

2016年12月30日

### 目录

一 需求说明 4

1文法说明 4

2目标代码说明 13

3优化方案 14

二 详细设计 15

1程序结构 15

2类/方法/函数说明 15

2.1词法分析部分 15

2.2语法分析部分 18

2.3词义分析及代码生成部分 21

3调用依赖关系 22

3.1整体分析 22

3.2词法分析 22

3.3语法分析 23

4符号表管理方案 23

4.1数据结构 23

4.2管理算法 24

5存储分配方案 24

5.1存储组织 24

5.2存储管理 24

5.3运行栈结构 25

7四元式设计 25

7.1四元式结构 26

7.2四元式操作符说明 26

8目标代码生成方案 27

8.1头部 27

8.2变量声明 27

8.3指令选择 27

9优化方案 28

9.1常数合并和传播 28

9.2消除局部公共子表达式 28

10出错处理 29

10.1出错处理方案 29

10.2错误提示 29

三 操作说明 31

1运行环境 31

2操作步骤 31

四 测试报告 33

1测试程序及测试结果 33

1.1测试test\_c1 33

1.2测试test\_c2 33

1.3测试test\_c3 35

1.4测试test\_c4 36

1.5测试test\_c5 37

1.6测试test\_c6 38

1.7测试test\_e7 38

1.8测试test\_e8 39

1.9测试test\_e9 40

1.10测试test\_e10 40

1.11测试test\_e11 41

1.12测试test\_e12 42

2测试结果分析 42

五 总结感想 45

## 一．需求说明

### 1．文法说明

1.1 ＜加法运算符＞ ::= +｜-

作用：将加法运算符定义为“+”或“-”。其中，“+”表示相加，“-”表示相减。

限定条件：双目运算符。若左操作数缺省，则表示右操作数为有符号数，“+”表示正数，“-”表示负数。一般不允许右操作符缺省。

示例：（设a、b为无符号数）

①a+b：a加上b；

②a-b：a减去b；

③+a：正数，其数值为a；

④-b：负数，其绝对值为b。

1.2 ＜乘法运算符＞ ::= \*｜/

作用：将乘法运算符定义为“\*”或“/”。其中，“\*”表示乘，“/”表示除。

限定条件：双目运算符。一般不允许操作数缺省。

示例：（设a、b为无符号数）

①a\*b：a乘b；

②a/b：a除b。

1.3 ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

作用：将关系运算符定义为“<”或“<=”或“>”或“>=”或“!=”或“==”。其中，“<”表示小于，“<=”表示小于或等于，“>”表示大于，“>=”表示大于或等于，“!=”表示不等于，“==”表示等于。

限定条件：双目运算符。一般不允许操作数缺省。由两个符号组合形成的关系运算符（如：<=，>=，!=，==）中间不允许插入其他任何字符，必须连在一起使用。

示例：（设a、b为无符号数）

①a<b：a小于b；

②a<=b：a小于或等于b；

③a>b：a大于b；

④a>=b：a大于或等于b；

⑤a!=b：a不等于b；

⑥a==b：a等于b。

1.4 ＜字母＞ ::= \_｜a｜．．．｜z｜A｜．．．｜Z

作用：将字母定义为“\_”（下划线）或[a-z]或[A-Z]。其中，[a-z]为从“a”到“z”的小写字母，[A-Z]表示从“A”到“Z”的大写字母。

限定条件：单字母。不区分大小写。

示例：

①a：字母a；

②A：字母A；

③\_：（字母）下划线。

1.5 ＜数字＞ ::= ０｜＜非零数字＞

作用：将数字定义为“0”或非零数字。

限定条件：单数字。

示例：

①0：数字0；

②1：数字1。

1.6 ＜非零数字＞ ::= １｜．．．｜９

作用：将非零数字定义为[1-9]。其中，[1-9]表示从“1”到“9”的数字。

限定条件：单数字。

示例：9（非零数字9）。

1.7 ＜字符＞ ::= '＜加法运算符＞'

｜'＜乘法运算符＞'

｜'＜字母＞'

｜'＜数字＞'

作用：将字符定义为包含加法运算符、乘法运算符、字母和数字的集合的元素。其中，加法运算符为“+”或“-”，乘法运算符“\*”或“/”，字母为“\_”（下划线）或[a-z]或[A-Z]，数字为[0-9]。

限定条件：单字符。不区分字母的大小写。

示例：

①’+’：字符’+’；

②’\*’：字符’\*’；

③’\_’：字符’\_’；

④’a’：字符’a’；

⑤’0’：字符’0’。

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

作用：将字符串定义为若干字符连接形成的有序组合。

限定条件：组成字符为十进制编码32、33、35-126的ASCII字符。字符串可为空。

示例：

①””：空字符串；

②”abc”：字符串abc。

1.9 ＜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＞

作用：将程序定义为说明块、若干定义块和主函数的组合。其中，说明块包括常量说明和变量说明，定义块为有返回值函数定义或无返回值函数定义。

限定条件：允许说明块缺省。定义块可缺省，也可有若干个。一般不允许主函数缺省。

示例：

const int max=100, min=0;

int in\_1, in\_2, out;

int add(int a, int b) { return(a+b); }

void print(int res) { if(res<=max) { if(res>min) { printf(res); printf(“, ”); } } return; }

void main() { scanf(in\_1, in\_2); out=add(int\_1, int\_2); print(out); return;}

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

作用：将常量说明定义为由不少于一条“const＜常量定义＞;”组成的语句块。

限定条件：关键字const表明为常量。

示例：

①const int a=1;：常量a为整数1；

②const int a=1; const char c=’d’;：常量a为整数1，常量c为字符’d’。

1.11 ＜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＞}

作用：将常量定义定义为类型（int或char）和不少于一条定义语句的顺序组合。其中，int表明为整型常量（整数），chat表明为字符型常量（字符）。

限定条件：若干条定义语句之间用逗号,分隔。每处常量定义只允许出现int、char中的一种类型。

示例：

①int a=3：a被定义为整数3；

②char a=’c’：a被定义为字符’c’。

1.12 ＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

作用：将无符号整数定义为若干数字的有序组合。其中，数字包括0和非零数字。

限定条件：无符号整数的首数字不为0。

示例：

①1：无符号整数1；

②10：无符号整数10。

1.13 ＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

作用：将整数定义为有符号（“+”或“-”）整数或0。其中，“+”表明正整数，“-”表明负整数。

限定条件：若符号缺省且数值不为0，则表明正整数。

示例：

①0：整数0；

②1：正整数；

③+2：正整数2；

④-2：负整数-3。

1.14 ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

作用：将标识符定义为若干字母和数字的有序组合。

限定条件：不允许为空。首字符必须为字母。不区分字母大小写。

示例：

①a：标识符a；

②\_007：标识符\_007；

③a1A2：标识符a1A2；

④\_abc56：标识符\_abc56。

1.15 ＜声明头部＞ ::= int ＜标识符＞| char ＜标识符＞

作用：将声明头部定义为类型（int或char）和标识符的顺序组合。

限定条件：每处声明头部只允许出现int、char中的一种类型。。

示例：

①int a：声明整型a；

②char a：声明字符型a。

1.16 ＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

作用：将变量说明定义为若干条变量定义。

限定条件：变量定义不少于一条。若干条变量定义用分号;分隔。

示例：

①int a, b；：整型变量a，整型变量b；

②char String[10];：字符型变量数组String（数组维数为10）。

1.17 ＜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＞‘]’)}

作用：将变量定义定义为类型标识符和若干标识符或标识符[无符号整数]的顺序组合。其中，标识符[无符号整数]表示数组定义，无符号整数表明数组维数。

限定条件：至少有一个标识符或标识符[无符号整数]。若干标识符或标识符[无符号整数]之间用逗号,分隔。

示例：

①int a, b：整型变量a，整型变量b；

②char String[10]：字符型变量数组String，数组维数为10。

1.18 ＜常量＞ ::= ＜整数＞|＜字符＞

作用：将常量定义为整数或字符。

限定条件：整数包括正整数、负整数和0。

示例：

①0：整数常量；

②’a’：字符常量；

③-1：整数常量。

1.19 ＜类型标识符＞ ::= int | char

作用：将类型标识符定义为int或char。

限定条件：类型标识符为固定字符串。

示例：

①int：整型标识符；

②char：字符型标识符。

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

作用：将有返回值函数定义定义为声明头部、(参数)和{复合语句}的有序组合。其中，声明头部中的类型即为函数返回值的类型。

限定条件：参数外部用左右小括号()括起来，复合语句外部用左右大括号{}括起来。

示例：

①int add(int a, int b) {int sum; sum=a+b; return(sum);}：add函数，计算两个整型形参值的和，返回值为整型；

②char sub(char a, int b) {char res; res=a-b; return(res);}：sub函数，计算两个整型形参的差，返回值为浮点型；

③char get() {return ‘d’;}：get函数，返回值为字符型。

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

作用：将无返回值函数定义定义为void、标识符、(参数)和{复合语句}的有序组合。其中，void表明该函数无返回值。

限定条件：参数外部用左右小括号()括起来，复合语句外部用左右大括号{}括起来。

示例：

①void print() {printf(“Hello World!”); return;}：print函数，打印字符串“Hello World!”；

②void read(int tmp) { scanf(tmp); return;}：read函数，读入一个整数tmp。

1.22 ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

作用：将复合语句定义为若干常量说明、若干变量说明和语句列的顺序组合。

限定条件：常量说明和变量声明允许缺省。

示例：

①const int a=1, b=2; int sum; sum=a+b;：已知常量a、b，计算其和并赋值给变量sum；

②int a; a=233;：将233赋给整型变量a；

③return(0);：返回值赋为0。

1.23 ＜参数＞ ::= ＜参数表＞

作用：将参数定义为参数表。

限定条件：无。

示例：

①int a：整型参数a；

②int b, char c：整型参数b，字符型参数c；

③：空，参数缺省。

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

|＜空＞

作用：将参数表定义为若干个类型标识符和标识符的有序组合或空。

限定条件：参数表允许缺省。若干个若干个类型标识符和标识符的有序组合之间用逗号,分隔。

示例：

①int a：整型参数a；

②int b, char c：整型参数b，字符型参数c；

③：空，参数缺省。

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

作用：将主函数定义为void、main、()和{复合语句}，其中，void表明无返回值，main表明主函数。

限定条件：无返回值。参数缺省。

示例：

void main() {printf(“Hello World!”);}

1.26 ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

作用：将表达式定义为符号（“+”或“-”）、项、若干个加法运算符和项的顺序组合的有序组合。

限定条件：符号允许缺省。加法运算符和项的顺序组合允许缺省。

示例：（设a、b、c为变量或常量）

①a：表达式a；

②+a：表达式+a，同a；

③-a+b-c：表达式-a+b-c。

1.27 ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

作用：将项定义为因子、若干乘法运算符和因子的顺序组合的有序组合。

限定条件：乘法运算符和因子的顺序组合允许缺省。

示例：（设a、b、c为变量或常量）

①a：项a；

②a\*b\*c：项a\*b\*c。

1.28 ＜因子＞ ::= ＜标识符＞｜＜整数＞｜＜字符＞

｜＜标识符＞‘[’＜表达式＞‘]’｜‘(’＜表达式＞‘)’

｜＜有返回值函数调用语句＞

作用：将因子定义为标识符或标识符[表达式]或整数或实数或字符或又返回值函数调用语句或(表达式)。其中，标识符[表达式]表明数组元素。

限定条件：无。

示例：

①a：因子a；

②a[1+1]：因子a[2]；

③666：因子666；

④’a’：因子’a’，计算时其值为对应ASCII编码；

⑤(a+b)：因子(a+b)。

1.29 ＜语句＞ ::= ＜条件语句＞|＜情况语句＞｜＜循环语句＞

｜‘{’＜语句列＞‘}’｜＜空＞;

｜＜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＞;

｜＜返回语句＞;

作用：将语句定义为条件语句或循环语句或{语句列}或有返回值函数调用语句或无返回值函数调用语句或赋值语句或读语句或写语句或空或返回语句。

限定条件：语句允许缺省。

示例：

①：条件语句；

②while(i>0) i=i-1;：循环语句；

③：赋值语句；

④scanf(a);：读语句；

⑤printf(“Hello World!”)：写语句；

⑥：空语句；

⑦return(0);：返回语句；

⑧scanf(a);if(a==0) a=1; else a=0;a=1+1;；语句列

⑨switch(a){case 1:a=a+1;case 2:a=a+2;default:a=0;}：情况语句

1.30 ＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞

|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

作用：将赋值语句定义为标识符=表达式或标识符[表达式]=表达式。其中，标识符[表达式]表示数组元素。赋值运算符=为双目运算符，将右操作数的值赋给左操作数。

限定条件：左右操作数类型一致。一条赋值语句中赋值运算符=仅出现一次，即不允许连续赋值（连等）。

示例：

①a=1+1：a被赋值为2；

②a[1+1]=’z’：字符数组元素a[2]被赋值为’z’。

1.31 ＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

作用：将条件语句定义为if(条件)、语句、else(语句)的顺序组合。当条件为逻辑“真”（值不为0）时执行else前的语句，否则执行else后的语句。

限定条件：else(语句)允许缺省，等同于else后语句为空。

示例：

①if(a==0) a=1;：当a的值为0时将a赋值为1；

②if(a==0) a=1; else a=0;：当a的值为0时将a赋值为1，否则将a赋值为0。

1.32 ＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞

｜＜表达式＞

作用：将条件定义为表达式、关系运算符、表达式的顺序组合或单独的表达式。其中，顺序组合值或表达式值为0表明条件为逻辑“假”，否则为逻辑“真”。

限定条件：无。

示例：（设a、b为变量或常量）

①a==b：若a与b的值相等则“真”，否则“假”；

②a：若a的值不为0则“真”，否则“假”。

1.33 ＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

作用：将循环语句定义为while语句，即while语句定义为while(条件)和语句的有序组合。

限定条件：无

示例：（设a和i为变量）

①while(i>0) i=i-1;：当i的值大于0时执行循环体（i自减一）；

②while(a)a=0;：当a的值不为0时执行循环体（将a赋值为0）。

1.34 ＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’‘{’＜情况表＞［＜缺省＞］‘}’

作用：将情况语句定义为switch语句。其中包含情况表（若干case语句）和缺省（default）语句，缺省语句是可以省略的。当switch后的表达式值满足情况表中的某种情况时执行其后的语句（执行到下一条case语句时跳出），否则执行default（若有）后的语句。

限定条件：switch后表达式的值为常量。

示例：（设a为常量）

①switch(a){case 1:a=a-1;case 2:a=a-2;default:a=0;}：判断a的值，当a的值为1时将a的值减1，当a的值为2时将a的值减2，否则将a赋值为0；

②switch(a){case 1:a=a-1;case ‘c’:a=a-2;}：判断a的值，当a的值为1时将a的值减1，当a的值为字符’c’所对应的ASCII码时将a的值减2，否则不进行任何操作。

1.35 ＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

作用：将情况表定义为若干条情况子语句。

限定条件：情况子语句不少于一条，每条情况子语句代表不同情况（case后常量值不同）。

示例：（设a为变量）

①case 1:a=a-1;case 2:a=a-2;：当switch后的表达式的值为1时将a的值减1，当switch后的表达式的值为2时将a的值减2；

②case ‘c’:a=a-2;：当switch后的表达式的值为字符’c’所对应的ASCII码时将a的值减2。

1.36 ＜情况子语句＞ ::= case＜常量＞：＜语句＞

作用：将情况子语句定义为一条case语句。其中，case后为一常量，即若符合该常量所代表的情况，则执行其后的语句。

限定条件：常量为整数或字符（不同情况子语句中的常量值不同）。若为字符，则在比较时自动转换为其对应的ASCII码值。

示例：（设a为变量）

①case 1:a=a-1;：当switch后的表达式的值为1时将a的值减1；

②case ‘c’:a=a-2;：当switch后的表达式的值为字符’c’所对应的ASCII码时将a的值减2。

1.37 ＜缺省＞ ::= default : ＜语句＞

作用：将缺省定义为一条default语句。当case语句中没有满足表达式值的情况时执行default后的语句。

限定条件：default语句在switch情况语句中至多有一条，可省略。

示例：

default:a=a-1;：当switch后的表达式的值不等于任何一条case语句中的常量值时将a的值减1。

1.38 ＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

作用：将有返回值函数调用语句定义为标识符和(值参数表)的有序组合。

限定条件：标识符为已定义的有返回值函数名。值参数表允许缺省。

示例：（设func1和func2为已定义的有返回值函数）

①func1();：调用无参函数func1；

②func2(int a);：调用有参（a）函数func2。

1.39 ＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

作用：将无返回值函数调用语句定义为标识符和(值参数表)的有序组合。

限定条件：标识符为已定义的无返回值函数名。值参数表允许缺省。

示例：（设func1和func2为已定义的有返回值函数）

①func1();：调用无参函数func1；

②func2(a);：调用有参（参数a）函数func2。

1.40 ＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

作用：将值参数表定义为若干表达式的有序组合或空。

限定条件：值参数表允许缺省。若干表达式之间用逗号,分隔。

示例：（设a、b为变量或常量）

①：空，值参数缺省；

②a：值参数a；

③a, a+b：值参数a，值参数a+b。

1.41 ＜语句列＞ ::= ｛＜语句＞｝

作用：将语句列定义为若干条语句的集合。

限定条件：语句列允许为空。

示例：

①：语句列为空；

②scanf(a);if(a==0) a=1; else a=0;a=1+1;：语句列（含读语句、条件语句和赋值语句）。

1.42 ＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

作用：将读语句定义为scanf和(不少于一个标识符)的顺序组合。

限定条件：若干个标识符之间用逗号,分隔。

示例：

①scanf(a)：将读入的值赋给a；

②scanf(a,b)：将读入的两个值按次序分别赋给a和b。

1.43 ＜写语句＞ ::= printf ‘(’＜字符串＞,＜表达式＞‘)’

| printf ‘(’＜字符串＞‘)’

| printf ‘(’＜表达式＞‘)’

作用：将写语句定义为printf和(字符串, 表达式)或(字符串)或(表达式)的有序组合。其中，(字符串)原样输出，(表达式)输出其值。

限定条件：(字符串, 表达式)中确定了输出顺序。

示例：（设a为整型变量或常量）

①printf(“The result is: ”, a)：打印字符串“The result is: ”并在其末尾输出a值；

②printf(“Hello World!”)：打印字符串“Hello World!”；

③printf(a)；打印a值。

1.44 ＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

作用：将返回语句定义为return和(表达式)的有序组合。

限定条件：(表达式)允许缺省。

示例：

①return：无返回值，函数结束标志；

②return(a)：将a作为返回值；

附加说明：

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符；

（2）标识符不区分大小写字母；

（3）写语句中的字符串原样输出；

（4）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句；

（5）数组的下标从0开始。

### 2．目标代码说明

本程序生成的目标代码为x86汇编程序，适用intel 80386指令集：

|  |  |
| --- | --- |
| 指令 | 含义 |
| mov | move数据传送指令 |
| add | add加指令 |
| neg | negate求补（求反） |
| sub | subtract减指令 |
| mul | 无符号乘 |
| cdq | Convert Double to Quad数据扩展指令  把edx扩展为eax的高位  （标记被除数） |
| idiv | 有符号除 |
| jge | 条件跳转指令  有符号大于等于则跳转 |
| jg | 条件跳转指令  有符号大于则跳转 |
| jle | 条件跳转指令  有符号小于等于则跳转 |
| jl | 条件跳转指令  有符号小于则跳转 |
| jz | 条件跳转指令  为0则跳转 |
| jnz | 条件跳转指令  不为0则跳转 |
| cmp | compare比较指令 |
| jmp | 无条件跳转指令 |
| invoke | 函数调用指令 |
| ret | 函数返回指令  相当于pop+jmp |
| proc | 定义子过程的伪指令  起始标志 |
| endp | 定义子过程的伪指令  结束标志 |
| end | 指定主过程  end后面的过程名是入口的子过程名 |
| local | 声明局部变量 |

### 3. 优化方案\*

进行代码优化的主要目的是要获得更高效的目标程序，使目标代码更加简短，以尽量减少存储空间和运行时间，同时要确保源程序功能不变。

在局部范围内可能做的优化包括常数表达式的计算、操作符的某些运算律的使用以及公共子表达式的检测等。还可以进行更大范围的优化，包括对多次出现的相同子表达式进行一次性计算、去除在循环体内不变的语句并将它们移到循环外等。还可以进行与机器有关的优化，如变量的寄存器分配等。

在下面的详细设计里将介绍本程序采用的两种优化方法，常量合并和传播以及消除局部公共子表达式。

## 二．详细设计

### 1．程序结构



### 2．类/方法/函数功能

2.1词法分析部分

词法分析部分的功能是扫描源程序字符，按文法中给定的词法规则识别并输出不同的单词或符号，同时进行词法检查。

2.1.1函数描述

|  |  |
| --- | --- |
| 函数名 | 功能描述 |
| errorSym | 进行词法分析过程中的错误处理  打印错误提示信息 |
| getChar | 读字符过程  每调用一次读进一个字符放在CHAR中  把读指针指向下一个字符 |
| isAscii | 判断字符串常量中的字符是否为合法字符 |
| isSpace | 判断CHAR是否为空格 |
| isNull | 判断CHAR是否为空字符 |
| isLetter | 判断CHAR是否为字母或下划线 |
| isDigit | 判断CHAR是否为数字 |
| isComma | 判断CHAR是否为逗号 |
| isSemicolon | 判断CHAR是否为分号 |
| isSquote | 判断CHAR是否为单引号 |
| isDquote | 判断CHAR是否为双引号 |
| isAssign | 判断CHAR是否为等号 |
| isBang | 判断CHAR是否为叹号 |
| isPlus | 判断CHAR是否为加号 |
| isMinus | 判断CHAR是否为减号 |
| isTimes | 判断CHAR是否为乘号 |
| isDivi | 判断CHAR是否为除号 |
| isLparen | 判断CHAR是否为左括弧 |
| isRparen | 判断CHAR是否为右括弧 |
| isLbrack | 判断CHAR是否为左方括号 |
| isRbrack | 判断CHAR是否为右方括号 |
| isLbrace | 判断CHAR是否为左花括号 |
| isRbrace | 判断CHAR是否为右花括号 |
| isGtr | 判断CHAR是否为大于号 |
| isLss | 判断CHAR是否为小于号 |
| isColon | 判断CHAR是否为冒号 |
| clearToken | 清空TOKEN字符数组 |
| catToken | 每次调用把当前CHAR和TOKEN连接 |
| reserver | 查找保留字  若未找到则设置当前TOKEN类别码为标识符  否则设置当前TOKEN类别码为对应关键字 |
| Symbol | 词法分析子程序  识别并设置当前读入单词的类别码 |

2.1.2关键算法



2.2语法分析部分

在语法分析过程中，将根据语法规则将单词进一步组合成文法中给定的语法类或语法成分，如变量声明、表达式、语句、函数、过程等。

2.2.1函数描述

|  |  |
| --- | --- |
| 函数名 | 功能描述 |
| errorGram | 进行语法分析中的错误处理  打印错误提示信息 |
| enterTab | 填表  将标识符信息填入符号表中 |
| Factor | 因子分析子程序  识别因子这一语法成分 |
| Term | 项分析子程序  识别项这一语法成分 |
| Expression | 表达式分析子程序  识别表达式这一语法成分 |
| transStr | 将整数转化为对应字符串  等价于itoa函数 |
| transNum | 将字符串转化为对应整数  等价于atoi函数 |
| initNext | 初始化类别符槽  读入三个类别符 |
| newNext | 更新类别符槽  向前读入一个类别符 |
| constDef | 常量定义分析子程序  识别常量定义语句这一语法类 |
| constDecl | 常量声明分析子程序  识别常量声明语句这一语法类 |
| varDef | 变量定义分析子程序  识别变量定义语句这一语法类 |
| varDecl | 变量声明分析子程序  识别变量声明语句这一语法类 |
| Parameter | 参数分析子程序  识别参数这一语法成分 |
| Condition | 条件分析子程序  识别条件这一语法成分 |
| Compound | 复合语句分析子程序  识别复合语句这一语法类 |
| preCase | 情况语句分析子程序  识别情况语句这一语法成分 |
| Case | 情况表分析子程序  识别情况表这一语法成分 |
| singleCase | 情况子语句分析子程序  识别情况子语句这一语法成分 |
| Default | 缺省分析子程序  识别缺省这一语法成分 |
| preStat | 判断语句起始类别符是否合法 |
| Statlist | 语句列分析子程序  识别语句列这一语法成分 |
| Constant | 常量分析子程序  识别常量这一语法成分 |
| Statement | 语句分析子程序  识别语句这一语法类 |
| callFunct | 函数调用语句分析子程序  识别函数调用语句这一语法类 |
| valParam | 值参数分析子程序  识别值参数这一语法成分 |
| fewParams | 判断值参数个数是否合法 |
| noParam | 判断无参函数调用是否未传参 |
| callProc | 过程调用语句分析子程序  识别过程调用语句这一语法类 |
| valFunct | 有返回值函数定义分析子程序  识别有返回值函数定义语句这一语法类 |
| novalFunct | 无返回值函数定义分析子程序  识别无返回值函数定义语句这一语法类 |
| mainFunct | 主函数分析子程序  识别主函数这一语法成分 |
| Program | 语法分析子程序  识别程序这一语法成分 |

2.2.2关键算法

本程序采用递归下降分析法。其主要做法是对文法中的每个产生式左部都构造出一个子程序，以完成该产生式左部所对应的语法成分的分析和识别问题。某个产生式左部的语法分析子程序的功能是用该产生式的右部去匹配输入串。

语法分析程序输出一颗语法分析树。在该语法分析树中，树的叶节点就是单词符号，而一个非叶节点代表一个语法成分。













2.3语义分析及代码生成部分

语义分析程序通过语法分析程序识别出来的语法成分进一步进行语义分析处理，识别它们的含义，以确定源程序的意义。语义分析程序的功能还包括生成中间代码，本程序中生成的中间代码形式是四元式，与目标机器无关，使用它便于代码优化和编译程序的移植。

代码生成程序是将源程序的中间形式（四元式）转换为x86汇编语言。

2.3.1函数描述

|  |  |
| --- | --- |
| 函数名 | 功能描述 |
| errorMean | 进行语义分析中的错误处理  打印错误提示信息 |
| enterTab | 填表  将标识符信息填入符号表中 |
| getType | 取符号表某表项的类型 |
| getTabtop | 取栈顶指针 |
| genStrlab | 为字符串常量生成一个标识符 |
| genNewlab | 生成跳转标号 |
| genVarlab | 为中间变量生成一个标识符 |
| genQuat | 生成四元式  记录操作符、操作数、结果和标号 |
| locTab | 查表  根据标识符名称查符号表并返回对应表索引 |
| newKind | 确定中间变量的种类  常量之间的计算结果仍为常量  其他计算结果均为变量 |
| newType | 确定中间变量的类型  整型、字符型参与的计算，结果均为为整型 |
| printHead | 打印目标代码头部  x86包含路径、库文件等信息 |
| printQuat | 打印四元式 |
| finishBuild | 在目标文件中打印完整的目标代码  包括.data和.code部分 |
| genAsm | 生成目标代码  根据四元式对应表项，用x86指令描述操作 |

2.3.2关键算法

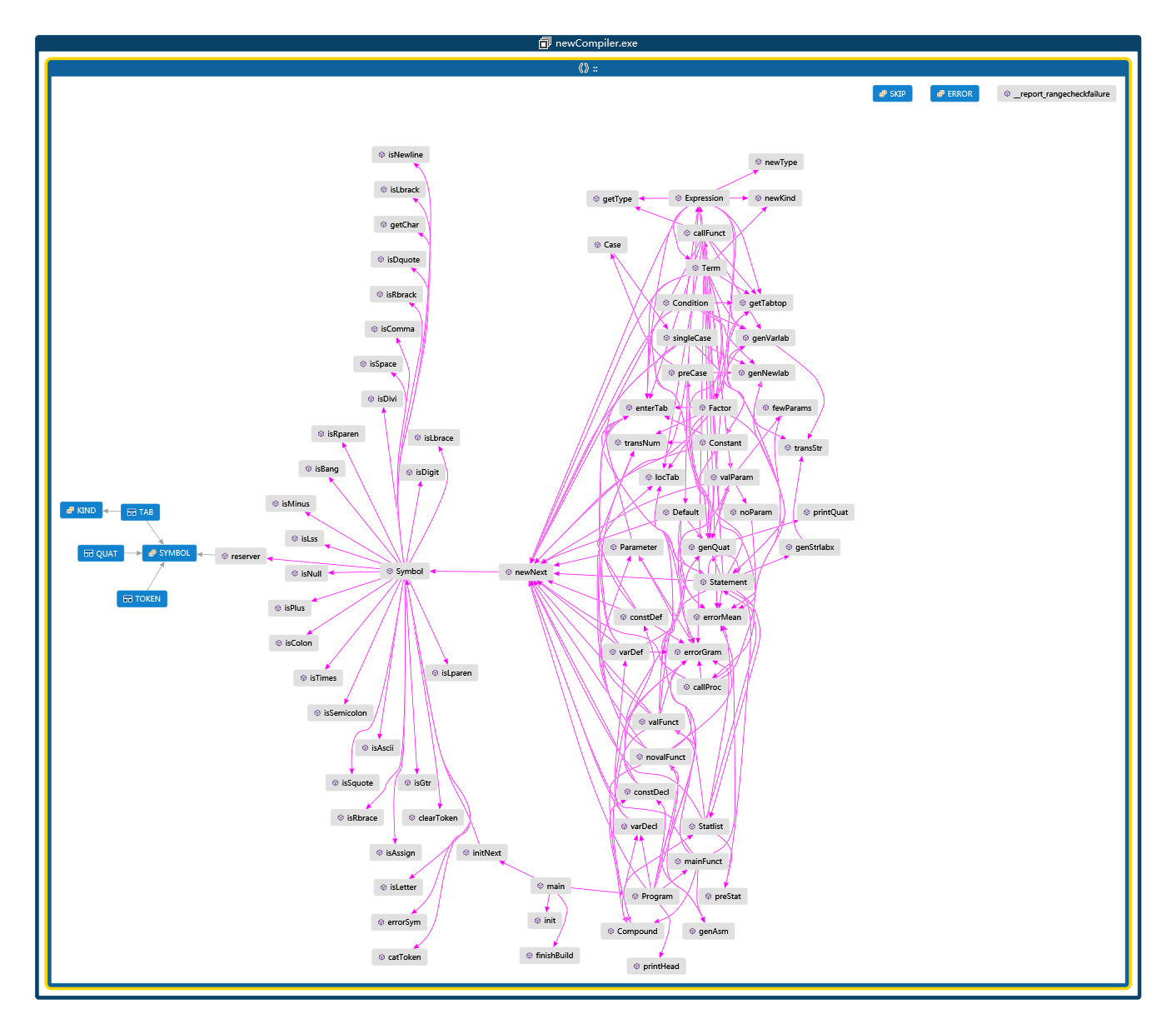
本程序使用语法制导翻译方法。在这种方法中，将产生式所包含的语义用一个或多个子程序（语义动作）来描述，并将这些子程序名插入产生式的相应位置，从而形成翻译文法；进一步，可分析文法中每个符号的语义，并将这些语义以属性的形式附加到相应的符号上，再根据产生式所包含的语义给出符号间属性的求值规则，从而形成属性翻译文法。

翻译文法的自顶向下翻译分析输入文法的每个产生式以及其中每个符号的意义，并在适当位置插入动作符号，这些动作符号与完成所需的语义动作的语义子程序相对应。这样就将输入文法扩展为相应的翻译文法。

使用语法制导翻译方法可以为源程序产生中间代码。

### 3．调用依赖关系

3.1整体分析



3.2词法分析



3.3语法分析



### 4．符号表管理方案

4.1数据结构

符号表是用于保存每个标识符及其属性信息的数据结构，在符号表的表项中登录标识符的属性。本程序中符号表由一系列行组成，每一行均包含名字域和特性域。

名字域用来存放标识符的符号串或指向标识符字符串的指针。特性域包括多个域，分别表示标识符的有关信息，特别地，对于数组要考虑其上界（维数），对于函数要考虑形参个数、函数返回值类型等。

|  |  |
| --- | --- |
| 代码 | 说明 |
| struct TAB  {  char name[IDMAX + 1];  enum KIND kind;  enum SYMBOL type;  int value;  int arrdim;  };  struct TAB tab[TABMAX];  Int TAB\_index; | 标识符名称  标识符中类  标识符类型  值  数组维数或形参个数  符号表  符号表索引 |

4.2管理算法

采用栈式符号表。当到达标识符声明时，将该标识符有关信息记录到符号表中。当程序执行到分程序结尾时，将在这个分程序中声明的所有标识符记录移出堆栈。

4.2.1插入操作

插入前首先要检查是否有重复的名字声明（按推入顺序的逆序检查正在编译的分程序中所有已声明的名字记录，以保证变量的最小作用范围），源程序允许存在重复的名字声明，但是它们不能出现在同一分程序中。插入时将新的标识符信息记录到栈顶单元。

4.2.2查表操作

对符号表进行从顶到底的线性检索，确保找到最近出现的标识符名字。

4.2.3定位操作

在分程序表的顶端将产生一个新的分程序索引项，并赋值为当前符号表栈栈顶的索引，由此建立起一个指向当前分程序的第一个记录单元的指针。

|  |  |
| --- | --- |
| 代码 | 说明 |
| int btabx[LEVELMAX];  int BTABX\_index; | 分程序表  分程序表索引 |

4.2.4重定位操作

将分程序表的顶端表项的值设置为符号表顶端第一个空闲存储单元的索引，从而重写覆盖刚刚编译完成的分程序在栈式符号表中的所有记录。

### 5．存储分配方案

5.1存储组织

|  |  |  |
| --- | --- | --- |
| 地址 | 内存区 | 说明 |
| ↓ | 栈区 | 向下增长  保存局部变量 |
| ↑ | 堆区 | 向上增长 |
|  | 静态区 | 未初始化全局变量  已初始化全局变量、静态变量、常量 |
|  | 代码区 | 可执行代码 |

5.2存储管理

在程序的执行过程中，程序中数据的存取是通过对应的存储单元进行的。在程序执行以前，首先将用x86汇编语言编写的程序输送到内存的某个固定区域上，并预先给变量和数据分配相应的内存地址。

5.2.1静态存储分配

对于源程序中出现的简单变量、字符串常量等数据，在编译时就能确定其存储空间的大小，因而在编译时就可以为它们分配固定的存储空间，从而在目标程序运行时直接使用这些存储单元作为它们的数据空间。

特别地，本程序使用常量合并和传播的优化方法，不在全局变量声明区声明常量，在使用时直接以立即数形式出现。

5.2.2动态存储分配

对于源程序中出现的子过程等，程序对其数据区的需求在编译时是未知的，但是当程序在运行中进入一个子过程时，其所需的数据区大小必须是已知的。在动态存储分配方案中，每当程序运行进入一个子过程时就为其分配一个新的数据区。因此，数据对象的多次出现也是允许的，如子过程的递归调用。

在本程序中使用运行栈来实现动态存储分配策略。程序内的每一个子过程都有自己的数据区，在程序运行中，当子过程被调用时，可以从运行栈中请求一个空间作为它的数据区，并保留该空间直至执行完整个子过程为止，即执行完毕后从运行栈中弹出。同时，从子过程被调用到它运行结束之间，还可以通过过程或函数调用进入其他子过程，此时也可按前述方法将其数据区压入运行栈，当程序返回到开始调用的地方时，运行栈的结构和内容立即恢复到调用之前的情况。

5.3运行栈结构

X86汇编中与堆栈相关的3个寄存器是：SS，ESP，EBP。ESP寄存器中的内容作为堆栈的当前指针，使用SS:ESP指向堆栈单元。EBP寄存器中的内容作为堆栈的基指针，SS:EBP指向的地址作为基准地址，在函数（子程序）内部，可以使用[EBP+立即数]的形式来取得主程序传递的参数，使用[EBP-立即数]的形式来访问局部变量。

|  |  |  |
| --- | --- | --- |
| 指针 | 数据 | 说明 |
| 帧指针EBP→  栈指针ESP→ | argn  ...  argi | 参数 |
| Ret addr | 返回地址 |
| Last EBP | 上一个EBP地址 |
| Local var | 局部变量的保存区 |

运行栈是子程序运行时所需的基本空间。进入子程序时分配，地址向下生长；从子程序返回时，当前运行栈将被“移除”；递归调用的同一个子程序，每次调用都将获得不同的运行栈空间。

### 7. 四元式设计\*

四元式是一种源程序的中间表示形式，便于对程序进行优化处理，其每条指令具有如下形式的4个域：

＜操作符＞＜操作数1＞＜操作数2＞＜结果＞

其中，＜操作符＞表示四元式的功能；＜操作数1＞和＜操作数2＞分别表示第一个和第二个操作数；＜结果＞表示计算的结果，该结果通常是一个临时变量，编译程序可以为该变量分配一个寄存器或一个主存地址。

7.1四元式结构

在本程序中，添加一个新的＜标号＞域，标记四元式所在位置，用于跳转指令。

|  |  |
| --- | --- |
| 代码 | 说明 |
| struct QUAT  {  enum SYMBOL op;  int operand\_1;  int operand\_2;  int result;  int labx;  };  Struct QUAT quat[QUATMAX];  int QUAT\_index; | 操作助记符  第一个操作数  第二个操作数  结果  标号  四元式表  四元式表索引 |

7.2四元式操作符说明

|  |  |
| --- | --- |
| 操作助记符 | 说明 |
| ADD | 加 |
| SUB | 减 |
| MUL | 乘 |
| DIV | 除 |
| JGE | 大于等于跳转 |
| JG | 大于跳转 |
| JLE | 小于等于跳转 |
| JL | 小于跳转 |
| JZ | 等于0跳转 |
| JNZ | 不等于0跳转 |
| ASSIGN | 赋值 |
| NEG | 取补（求负） |
| CMP | 比较大小 |
| JMP | 无条件跳转 |
| RFUNC | 有返回值函数调用 |
| NFUNC | 无返回值函数调用 |
| LDPAR | 加载参数 |
| NOP | 空（打印标号） |
| SCANF | 读整数或读字符 |
| PRINTFE | 打印表达式的值 |
| PRINTFS | 打印字符串 |
| RET | 函数或过程结束（返回值） |
| PROC | 函数或过程定义 |
| END | 函数或过程定义结束 |
| MPROCE | main过程定义 |
| MEND | main过程定义结束 |
| LOCARRY | 局部数组变量定义 |
| LDARY | 加载数组元素 |

### 8. 目标代码生成方案\*

代码生成模块面向x86体系结构生成目标代码，其主要任务包括：从中间代码到目标代码转换过程中所进行的指令选择；目标代码地址空间的划分，目标体系结构上的存储单元的分配和指派；目标代码序列的生成等。

8.1头部

.386

.model flat, stdcall

include F:\masm32\include\msvcrt.inc

includelib F:\masm32\lib\msvcrt.lib

8.2变量声明

将全局变量声明在.data后，全局静态变量不必声明（使用时直接转换为立即数形式）。局部变量使用local伪指令声明在子过程内部。

8.3指令选择

从中间代码到目标代码转换过程中，考虑为输入的中间代码选择语义等价且运行效率较高的目标指令集合（最终生成的目标指令序列放在.code后）。

|  |  |
| --- | --- |
| 四元式操作符 | X86指令组合 |
| PLUS | mov+add |
| MINUS | mov+neg+sub |
| TIMES | mov+mul |
| DIVI | mov+cdq+idiv |
| LSS | jge |
| LEQ | jg |
| GTR | jle |
| GEQ | jl |
| NEQ | jz |
| EQL | jnz |
| ASSIGN | mov+add |
| NEG | mov+neg |
| CMP | mov+cmp |
| JMP | jmp |
| RFUNC | invoke |
| NFUNC | invoke |
| NOP |  |
| SCANF | invoke |
| PRINTFE | mov+invoke |
| PRINTFS | invoke |
| RET | mov+ret |
| PROC | proc+local |
| END | ret+endp |
| LDARY | mov+add |
| MPROC | proc+local |
| MEND | ret+endp+end |
| LOCARRAY | local |
| LOCVAR | local |

### 9. 优化方案\*

9.1常数合并和传播

常数合并是在编译时计算出值的表达式用其相应的值代替，即若在编译时，编译程序能知道这一个表达式的所有操作数的值，则此表达式就可以由其计算出的值替代。

常数传播是在编译时已知的变量值来代替程序正文中对这些变量的引用。

9.2消除局部公共子表达式

输入：中间代码语句序列。

输出：消除局部公共子表达式后的中间代码序列。

方法：

①划分基本块，每条中间代码属于且仅属于一个基本块。

首先确定入口语句的集合：整个语句序列的第一条语句属于入口语句；任何能由条件或无条件跳转语句转移到的第一条语句属于入口语句；紧跟在跳转语句之后的第一条语句属于入口语句。

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

②对于每个基本块内的中间序列，通过构建DAG图消除局部公共子表达式。

首先建立结点表，记录变量名和常量值，以及它们当前所对应的DAG图中结点的序号。置其初始状态为空。

从第一条中间代码开始，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。首先在结点表中寻找x，如果找到，记录下x当前所对应的结点号i；如果未找到，在DAG图中新建一个叶结点，假设其结点号为i，标记为x（若x为变量名，该标记更改为x0）；在结点表中增加这一新结点。右操作数与左操作数同理，假设其对应结点号为j。在DAG图中寻找中间结点，其标记为op，且左操作数结点号为i，右操作数结点号为j。如果找到，记录下其结点号k；如果未找到，在DAG图中新建一个中间结点，假设其结点号为k，并将结点i和j分别与k相连，作为其左子结点和右子结点。在结点表中寻找z，如果找到，将z所对应的结点号更改为k；如果未找到，在结点表中新建一项。

数组、指针及函数调用较复杂，不依据以上规则构建DAG图。

③从DAG图重新导出中间代码。

初始化一个放置DAG图中间结点的队列。

若DAG图中还有中间结点未进入队列，则选取一个尚未进入队列，但其所有父结点均已进入队列的中间结点n加入队列，或选取没有父结点的中间结点加入队列。如果n的最左子结点符合前述条件，将其加入队列，并沿着当前结点的最左边，循环访问其最左子结点、最左子结点的最左子结点等重复操作。

若DAG图中所有中间结点均已进入队列，则将中间结点队列逆序输出，便得到中间结点的计算顺序，将其整理成中间代码序列。

### 10. 出错处理

编译的各个阶段都能检测到源程序中的错误，而且，当检查到一个错误之后必须立即对该错误进行处理，除了正确报告出错信息外，还应使编译工作能够继续进行下去，继续对源程序中可能存在的其他错误进行检测。

10.1出错处理方案

编译程序在检查出错误以后，一方面要向用户报告错误信息，另一方面要对错误做适当处理，以便分析能继续下去。

10.1.1错误更正

当编译程序检查出错误以后，根据语言的文法和对程序意图做某种猜测，对这个错误进行改正。但是，有时源程序的意图很难从发现的错误中体现出来，所以要正确地改正错误是件困难的事，同时编译过程的复杂性也会增加。在本程序中没有显式地进行错误更正。

10.1.2错误局部化处理

当编译程序检查出错误以后，编译程序就暂停对后面的符号进行分析，跳过错误所在的语法单位，尽可能把错误限制在一个局部的范围内，避免程序扩散和影响程序其他部分的分析和检查。

在词法分析时，跳过出错的单词，找到下一个新的单词并继续往下分析。在用递归下降法进行语法分析过程中，检查出错误时，先报告错误信息，然后跳过错误所在的短语或语句，找到下一个新的短语或新的语句，继续往下分析。

10.2错误提示

10.2.1词法分析部分

|  |  |  |
| --- | --- | --- |
| 错误编号 | 错误助记符 | 含义 |
| 0 | other | 未定义的错误类型 |
| 1 | LONGID | 标识符过长 |
| 2 | KEYSAME | 标识符与关键字重名 |
| 3 | ZEROBEG | 非零整数以0开头 |
| 4 | LONGINT | 整数过长 |
| 5 | CHARCONS | 单引号内非字符常量 |
| 6 | SQLACK | 缺少右侧单引号 |
| 7 | ILSTR | 字符串常量中出现非法字符 |
| 8 | LONGSTR | 字符串常量过长 |

10.2.2语法分析部分

|  |  |  |
| --- | --- | --- |
| 错误编号 | 错误助记符 | 含义 |
| 0 | other | 未定义的错误类型 |
| 9 | CASELACK | 缺少case |
| 10 | IDLACK | 缺少标识符 |
| 11 | INTLACK | 缺少整型常量 |
| 12 | ASGLACK | 缺少赋值符号 |
| 13 | CHARLACK | 缺少字符型常量 |
| 14 | ILTYPE | 非法类型标识符 |
| 15 | ARRAYIDX | 数组变量非法下标 |
| 16 | SEMILACK | 缺少分号 |
| 17 | PARENLACK | 缺少括弧 |
| 18 | BRACKLACK | 缺少方括号 |
| 19 | BRACELACK | 缺少花括号 |
| 20 | COLONLACK | 缺少冒号 |
| 21 | DFTLACK | 缺少default项 |
| 22 | ILPRINT | 非法printf格式 |
| 23 | MAINLACK | 缺少主函数 |
| 24 | PROGEND | 程序未结束 |
| 25 | ILFACTOR | 未定义因子类型 |
| 26 | ILRETYPE | 无函数返回值类型 |
| 27 | ILSTAT | 非法的语句类型 |

10.2.3语义分析部分

|  |  |  |
| --- | --- | --- |
| 错误编号 | 错误信息 | 含义 |
| 0 | other | 未定义的错误类型 |
| 28 | PARACONT | 参数个数不匹配 |
| 29 | UNDEF | 使用未定义的标识符 |
| 30 | CONSASG | 给静态变量赋值 |
| 31 | REDEF | 重复定义标识符 |
| 32 | RETVAL | 过程返回值 |
| 33 | SCANFPAR | scanf传参错误 |
| 34 | PRINTFPAR | printf传参错误 |

## 三．操作说明

### 1．运行环境

本程序使用CodeBlocks 13.12编写。

生成的目标代码的编译环境配置使用Windows 7的MASM32。需要安装MASM32以得到汇编头文件（include）和库文件（lib）。

### 2．操作步骤

本程序使用MASM32编译工具建立、汇编、链接源程序文件和运行生成的可执行文件。

安装好MASM32后，打开“资源管理器”或“我的电脑”，进入MASM32所在文件夹，双击QEDITOR.EXE的图标，进入MASM32的集成开发环境。

①输入源程序。

使用菜单栏上的菜单File -> Save或快捷工具栏中的代表存盘功能的磁盘状图标，第一次需要输入文件名，注意要指定扩展名.asm。

②汇编源程序。

使用菜单栏上的菜单Project -> Assemblier ASM file。这时系统会打开一个DOS窗口运行相关的批处理文件。结束后会弹出一个名为“\masm32\bin\asmbl.txt”的窗口报告结果。

若汇编不成功，则会显示错误信息。这时应按这些信息修改源程序中相应行，存盘后重新汇编。直至窗口显示如下成功信息：

|  |
| --- |
| Assembling: F:\masm32\out.asm  驱动器 F 中的卷没有标签。  卷的序列号是 0006-0B32  F:\masm32 的目录  2016/12/28 19:15 9,119 out.asm  2016/12/28 20:03 4,626 out.obj  2 个文件 13,745 字节  0 个目录 200,662,761,472 可用字节 |

③链接目标文件。

使用菜单栏上的菜单Project -> Link OBJ file。这时系统会打开一个DOS窗口运行相关的批处理文件，并弹出名为“\masm32\bin\lnk.txt”的窗口报告结果如下：

|  |
| --- |
| 驱动器 F 中的卷没有标签。  卷的序列号是 0006-0B32  F:\masm32 的目录  2016/12/28 19:15 9,119 out.asm  2016/12/28 20:04 4,608 out.exe  2016/12/28 20:03 4,626 out.obj  3 个文件 18,353 字节  0 个目录 200,662,753,280 可用字节 |

④运行生成的可执行文件。

使用菜单栏上的菜单Project -> Run program。

## 四．测试报告

### 1．测试程序及测试结果

1.1测试test\_c1

int index\_fibo;

int Fibonacci(int n)

{

if (n == 0)

{

return(0);

}

if (n == 1)

{

return(1);

}

return(Fibonacci(n-1)+Fibonacci(n-2));

}

void main()

{

printf("Fibonacci..");

printf("Please input the index of Fibonacci: ");

scanf(index\_fibo);

printf("Expected result: ", Fibonacci(index\_fibo));

return;

}

测试结果：

Fibonacci..Please input the index of Fibonacci:

|  |  |  |
| --- | --- | --- |
| 序号 | 输入 | 输出 |
| 1 | 0 | Expected result: 0 |
| 2 | 1 | Expected result: 1 |
| 3 | 2 | Expected result: 1 |
| 4 | 3 | Expected result: 2 |
| 5 | 4 | Expected result: 3 |
| 6 | 5 | Expected result: 5 |

1.2测试test\_c2

void printPrime(int n)

{

int i, j;

int array[100];

if (n < 2) printf("No prime found..");

else if (n > 100) printf("Unexpected large area..");

else

{

i = 2;

while (i <= n)

{

array[i] = 1;

i = i + 1;

}

i = 2;

while (i <= n)

{

if (array[i] == 1)

{

j = i + i;

while (j <= n)

{

array[j] = 0;

j = j + i;

}

}

i = i + 1;

}

printf("Expected primes are:");

i = 2;

while (i <= n)

{

if (array[i] == 1) printf(" ", i);

i = i + 1;

}

}

return;

}

void main()

{

int area;

printf("Prime..");

printf("Please input the upper bound: ");

scanf(area);

printPrime(area);

}

测试结果：

Prime..Please input the upper bound: 10

|  |  |  |
| --- | --- | --- |
| 序号 | 输入 | 输出 |
| 1 | 0 | No prime found.. |
| 2 | 1 | No prime found.. |
| 3 | 2 | Expected primes are: 2 |
| 4 | 10 | Expected primes are: 2 3 5 7 |
| 5 | 20 | Expected primes are: 2 3 5 7 11 13 17 19 |

1.3测试test\_c3

int steps, no\_disk;

void move(int n, char \_from, char \_to)

{

steps = steps + 1;

printf("Step ", steps);

printf(": move disk ", n);

printf(" from ", \_from);

printf(" to ", \_to);

printf("..");

}

void Hanoi(char \_a, char \_b, char \_c, int n)

{

if (n > 10)

{

printf("Too many disks..");

}

else if (n < 1)

{

printf("No disk found..");

}

else

{

if (n != 1)

{

Hanoi(\_a, \_c, \_b, n - 1);

move(n, \_a, \_c);

Hanoi(\_b, \_a, \_c, n - 1);

}

else move(n, \_a, \_c);

}

}

void main()

{

steps = 0;

printf("Hanoi..");

printf("Please input the number of disks within [1,10]: ");

scanf(no\_disk);

Hanoi('A', 'B', 'C', no\_disk);

}

测试结果：

Hanoi..Please input the number of disks within [1,10]:

|  |  |  |
| --- | --- | --- |
| 序号 | 输入 | 输出 |
| 1 | 1 | Step 1: move disk 1 from A to C.. |
| 2 | 2 | Step 1: move disk 1 from A to B..  Step 2: move disk 2 from A to C..  Step 3: move disk 1 from B to C.. |

1.4测试test\_c4

const int c1 = 1;

const int c2 = -100;

const char c3 = 'a';

int r1;

int r2;

char r3;

int foo1(int var1,int var2)

{

r1 = var1 + var2;

return (r1);

}

void foo2(int var1, int var2)

{

r2 = var1 - var2;

}

char foo3(char var)

{

r3 = var+2;

return (r3);

}

void main(){

int v1;

int v2;

char v3;

printf("Input v1: ");

scanf(v1);

printf("Input v2: ");

scanf(v2);

v1 = foo1(v1,c1);

foo2(v2,c2);

v3 = foo3(c3);

printf(v1);

printf(r2);

printf(v3);

}

测试结果：

Input v1:

Input v2:

|  |  |  |
| --- | --- | --- |
| 序号 | 输入 | 输出 |
| 1 | 233  666 | 234766c |
| 2 | 555  748 | 556848c |

1.5测试test\_c5

int var[2];

void foo2(int x)

{

const int c1 = 3,c2 = 4;

printf("foo2: ");

printf(c1);

printf(c2);

}

void foo1(int y)

{

const int c3 = 1,c4 = 2;

printf("foo1: ");

printf(c3);

printf(c4);

foo2(c3);

}

void main()

{

var[0] = 5;

var[1] = 6;

foo1(var[0]);

printf("main: ");

printf(var[0]);

printf(var[1]);

}

测试结果：

foo1: 12foo2: 34main: 56

1.6测试test\_c6

int choice;

void main()

{

printf("Please choose a number from [1,4]: ");

scanf(choice);

switch(choice)

{

case 1:

{

printf("Game 1: Hanoi..");

}

case 2: printf("Game 2: Fibonacci..");

case 3:

{

printf("Game 3: Prime..");

}

case 4:

{

printf("Game 4: Guess character..");

}

default:printf("Error: invalid chosen number..");

}

return;

}

测试结果：

Please choose a number from [1,4]:

|  |  |  |
| --- | --- | --- |
| 序号 | 输入 | 输出 |
| 1 | 0 | Error: invalid chosen number.. |
| 2 | 1 | Game 1: Hanoi.. |
| 3 | 2 | Game 2: Fibonacci.. |
| 4 | 3 | Game 3: Prime.. |
| 5 | 4 | Game 4: Guess character.. |
| 6 | 7 | Error: invalid chosen number.. |

1.7测试test\_e7

void message()

{

printf("Hello World!");

}

void main()

{

char

message();

}

测试结果：

ERROR\_GRAMMER\_9\_ILLEGAL STATEMENT

1.8测试test\_e8

void foo2(int x)

{

const int v1 = 3,v2 = 4;

printf("foo2: ");

printf(v1);

printf(v2);

char

}

void foo1(int y)

{

const int r1 = 1,r2 = 2;

printf("foo1: ");

printf(r1));

printf(r2)

foo2(r1);

}

void main()

{

c1 = 5;

c2 = 6;

foo1(c1);

printf("

");

printf(c1);

printf(c2);

测试结果：

ERROR\_GRAMMER\_9\_ILLEGAL STATEMENT

ERROR\_GRAMMER\_14\_SEMICOLON LOST

ERROR\_GRAMMER\_14\_ILLEGAL STATEMENT

ERROR\_MEANING\_19\_UNDEFINED IDENTIFIER

ERROR\_MEANING\_20\_CONST VALUE CHANGE

ERROR\_MEANING\_20\_UNDEFINED IDENTIFIER

ERROR\_MEANING\_21\_CONST VALUE CHANGE

ERROR\_MEANING\_21\_UNDEFINED IDENTIFIER

ERROR\_SYMBOL\_22\_TOO LONG STRING

ERROR\_SYMBOL\_23\_TOO LONG STRING

ERROR\_GRAMMER\_24\_PAREN LOST

ERROR\_GRAMMER\_24\_SEMICOLON LOST

ERROR\_GRAMMER\_24\_ILLEGAL STATEMENT

1.9测试test\_e9

const int const1 = 1, const2 = -100;

const float const3 = 74.0;

int var1, var2;

void main()

{

int id;

int number;

int choice;

scanf( choice );

while ( choice )

{

switch(choice)

{

case 1: printf("first ");

case 2: printf("second ");

case 3: printf("third ");

default: printf("hh ");

}

scanf(choice);

}

}

测试结果：

ERROR\_GRAMMER\_2\_ILLEGAL TYPE

ERROR\_SYMBOL\_2\_OTHER

1.10测试test\_e10

void foo\_temp(int a)

{

if(a > 1)

{

a = a - 1;

foo\_temp(a);

}

else

printf(a);

}

void main()

{

foo\_temp(4);

return ();

}

测试结果：

ERROR\_GRAMMER\_15\_ILLEGAL FACTOR KIND

ERROR\_GRAMMER\_16\_PAREN LOST

ERROR\_MEANING\_16\_PROCEDURE RETURN VALUE

1.11测试test\_e11

const int id=0,ID=1;

char Main;

int result;

int foo(int v1,int v)

{

result = v1 + v2;

return (result);

}

char swap(char ch)

{

result=ch;

;;

ch=id;

}

void main()

{

printf(swap);

scanf(foo);

result=id\*/id+1;

}

测试结果：

ERROR\_SYMBOL\_2\_IDENTIFIER NAMED AS RESERVER

ERROR\_MEANING\_2\_REDEFINED IDENTIFIER

ERROR\_MEANING\_7\_UNDEFINED IDENTIFIER

ERROR\_MEANING\_17\_WRONG PRINTF PARAMETER

ERROR\_MEANING\_18\_WRONG SCANF PARAMETER

ERROR\_GRAMMER\_19\_ILLEGAL FACTOR KIND

1.12测试test\_e12

int choice;

void main()

{

printf("Please choose a number: ");

scanf(choice);

switch(choice)

{

case 1: return;

}

default:printf("Error: invalid chosen number..");

return;

}

测试结果：

ERROR\_GRAMMER\_10\_ILLEGAL STATEMENT

### 2．测试结果分析

2.1测试test\_c1

①测试覆盖：

全局变量声明、有返回值函数定义、条件语句、有返回值函数调用、返回语句、读语句、写语句、赋值语句、主函数

②主要测试功能：

递归函数的调用

③测试结果：

满足预期

2.2测试test\_c2

①测试覆盖：

无返回值过程定义、局部变量声明、数组、条件语句、无返回值过程调用、循环语句、返回语句、赋值语句、读语句、写语句、主函数

②主要测试功能：

数组的赋值

③测试结果：

满足预期

2.3测试test\_c3

①测试覆盖：

全局变量声明、无返回值过程定义、局部变量声明、表达式、条件语句、无返回值过程调用、赋值语句、读语句、写语句、主函数

②主要测试功能：

子过程间的调用

③测试结果：

满足预期

2.4测试test\_c4

①测试覆盖：

全局变量声明、无返回值过程定义、有返回值过程定义、局部变量声明、表达式、无返回值过程调用、有返回值过程调用、赋值语句、读语句、写语句、主函数。

②主要测试功能：

负数以及字符参与计算、输出字符型变量

③测试结果：

满足预期

2.5测试test\_c5

①测试覆盖：

全局数组变量声明、无返回值过程定义、局部静态变量声明、表达式、无返回值过程调用、赋值语句、读语句、写语句、主函数

②主要测试功能：

输出数组元素

③测试结果：

满足预期

2.6测试test\_c6

①测试覆盖：

全局变量声明、情况语句、读语句、写语句、主函数

②主要测试功能：

执行情况语句

③测试结果：

满足预期

2.7测试test\_e7

①测试报错类型：

局部变量声明不完整

②测试结果：

满足预期

2.8测试test\_e8

①测试报错类型：

局部变量声明位置错误、缺少分号、标识符未定义、字符串常量错误

②测试要求：

尽可能多地准确地报告错误信息

③测试结果：

满足预期

2.9测试test\_e9

①测试报错类型：

类型标识符使用错误、符号未定义

②测试要求：

尽可能多地准确地报告错误信息

③测试结果：

满足预期

2.10测试test\_e10

①测试报错类型：

返回语句括号内未传参、主函数出现返回值

②测试要求：

尽可能多地准确地报告错误信息

③测试结果：

满足预期

2.11测试test\_e11

①测试报错类型：

不区分大小写的标识符与保留字重名、不区分大小写的标识符重复定义、使用未定义的变量、写函数传参错误、读函数传参错误、函数调用格式错误

②测试要求：

尽可能多地准确地报告错误信息

③测试结果：

满足预期

2.12测试test\_e12

①测试报错类型：

情况语句中缺省语句位置错误

③测试结果：

满足预期

## 五．总结感想

差不多已经到编译课程设计的最后阶段了，我所能想到的所有关于这一过程的关键词就是“选择”和“坚持”了。

一开始出于对课程安排的综合考虑选择了中难度文法，但是在更深层次地学习了编译原理的理论知识以及逐步完成编译器设计的过程中，我发现自己还是有比较充足的时间和精力挑战一下高难度文法，所以在正式进行阶段测试的前一天我通过课程网站更改了文法。值得一提，这是一个很棒的决定。虽然在更换文法时我基本已经设计完成了原来的编译器，而且测试日期已经近在眼前，在这时更换文法无疑是个挑战，但是本着让自己快速进步的迫切目标，我坚持了下来，而且在完成新编译器的过程中，我有机会学习到了更多有用而重要的知识，比如x86汇编，在此之前我是不够了解它的。每个当下的选择都是极为重要的，实在不能放过每个应该要把握住自己的节点。

所以，我必须要强调的是，在最终结果浮出水面之前，我们或许没法判断哪个是最佳选项，那么我们所能做的就是坚定地踏实地努力地在我们已经选好的道路上勇往直前。尽可能多地把握锻炼自己的机会，将自己的价值最大化，磨砺自己，充实自己，拓宽视野。绝对不要轻易后撤，因为所有的放弃，所有的失败，都源于自己的狭隘。

在完成设计的过程中，理论知识是基础，是重要的东西，没有根基的人，将来走的任何一条道路都比那些基础深厚的人来得辛苦和单薄。这是不容置否的，在设计初期，总是很难入手，一定要深刻理解课上讲过的知识点，将其对应到编程实践中，比如语法分析时涉及递归下降分析法和语义分析时采用语法制导翻译等，这样会大大降低实现的难度。

另外，我要特别提到讨论的重要性。毕竟一个人的能力和眼界是有限的，如果一直一个人埋头写码，而不经常与同学们讨论，那么基本上可以断定最终完成的编译器一定是不完善的。就我自己而言，在完成文法解读作业时经常和舍友讨论，这使我在设计初期考虑得比较全面，在后期调试时就避免了大幅改动代码可能造成的一系列问题。

有人说，世界没有终点，只有里程碑。但行好事，莫问前程，那份柳暗花明的喜悦和必然的抵达往往取决于我们在山穷水尽之中的修持。

最后，庆幸自己的“选择”和“坚持”，感谢老师们为这门课程付出的心血，辛苦！