**《编译技术》课程设计大纲及指导书**

**实验名称：**

根据给定的文法完成一个小型编译器。

**实验目的：**

本课程设计实验项目的目标是通过实际构造一个完整的编译器，加深对编译器实现原理的理解，了解完整编译系统的构造方法。

**实验类型：**

验证型、提高型

**实验学时：**

50学时

**实验描述：**

根据各难度的题目的说明，选择一个难度系数，将获得一份随机分配的该难度的文法。根据获取的文法实现编译器：

1. **难度系数**： 1

**目标**：完成编译器及解释执行程序，解释执行程序对编译器产生的PCODE能解释执行，产生运行结果

**文法**：PL/0文法（与教材上的文法略有差别）

**优化**：无

**中间代码**：无

**目标码**：PCODE

**评分因子**：0.7

1. **难度系数：** 2

**目标：**完成编译器及解释执行程序，解释执行程序对编译器产生的PCODE能解释执行，产生运行结果

**文法：**扩充PL/0文法

**优化：**无

**中间代码：**无

**目标码：**PCODE

**评分因子**：0.85

1. **难度系数：** 3

**目标：**实现编译器，生成MIPS汇编

**文法：** 扩充C0文法(无实型)

**优化：**基本块内部的公共子表达式删除（DAG图）；

       全局寄存器分配（引用计数或着色算法）；  
          数据流分析（通过活跃变量分析，或利用定义-使用链建网等方法建立冲突图）；

其它优化自选；

代码生成时合理利用临时寄存器（临时寄存器池），并能生成较高质量的目标代码；

**中间代码：**四元式（有统一格式要求）

**目标码：**32位MIPS汇编

**评分因子**：1

**实验要求：**

1. 每人独立完成各次作业，按时提交7项阶段成果和1项最终成果

2. 采用递归子程序法进行语法分析，所有代码要求手工编程实现

3. 可以有图形用户界面，或将被编译的源代码文件以带路径的文件名方式从控制台传入

4. 最终成果需提交一份WORD文档（一个文件），包括以下内容：

   1) 需求说明：包括文法及其分析说明、目标代码说明、优化方案说明（3）

   2) 详细设计：包括程序结构、类、方法/函数的功能描述和调用依赖关系、符号表管理方案、存储分配方案、解释执行程序结构及过程 （1、2） 、四元式设计（3）、目标代码生成方案设计（3） 、优化方案设计（3） 、出错处理方案

   3) 操作说明：包括程序安装、环境配置；程序启动后，编译、运行、结果显示等操作的说明

   4) 测试报告：包括至少10个测试程序的源程序及测试结果（无需截屏）（其中5个是有错误的文件，如有扩充，还需另外提供5个测试程序，扩充后的语言必须包含原来的语言）、测试结果分析

   5) 总结感想：完成本实验项目后的感想及建议

5. 用C/C++语言实现

6. 开发环境

     VS2010或CodeBlocks13.12

Mars4.5

7. 提交完整的项目/工程文件，在集成开发环境下可编译运行

8. 目录命名及组织

阶段成果以”学号\_姓名\_任务说明”命名提交的文件（或文件夹），压缩为"学号.zip"，提交该zip文件；

最终成果组织为如下三个文件夹：

源代码   文档 测试程序  
  将上述三个文件夹放入以如下规则命名的文件夹中： 学号\_姓名 [\_申优]（如15061001\_×××\_申优, 15061001\_×××），再将上述文件文件夹压缩为"学号.zip"，提交该zip文件。

**考核：**

1. 检查文档内容是否齐全、正确。

2. 将对每份作业的源程序进行编译，编译不通过不能得分；不能正常运行，不能得分。

3. 每份作业将依次用4个测试程序进行测试，根据编译产生的目标代码运行的结果进行评分（测试程序包括考虑大小写、函数调用、不同作用域的同名变量、递归调用等方面)

4. 回答现场提问或按要求修改、演示程序；难度系数为3的作业还需提交指定测试程序产生的中间代码、目标代码和运行结果

5. 申优的同学交一篇文章，以指导如何完成课程设计为主要内容，着重说明在课程设计中遇到的各个困难点及所采用的解决方案，要求内容具体，阐述清楚，题目和内容组织自定。注意，文章与前面要求提交的技术文档不同。提交作业的同时提交文章。未提交申优文章视为放弃申优。

6. 若发现程序、测试程序或文档雷同，按作弊处理。

**作业获取和提交**  
  1. 访问教学辅助平台，以学号登录，获取和提交作业。  
  2. 每人最多能三次获取题目（同一难度最多允许两次），以最后一次为准进行考核（不保留前一次的记录，不能返回到前一次获取的结果），必须按最后一次获取的题目完成作业。  
   3. 请在规定的时间内上网提交作业，逾期将关闭提交通道。在时间允许范围内可多次提交作业，只保留最后一次。

**参考资料**

1. 《编译技术》第17章 第18章，PL/0编译器和Pascal-S编译器源代码

或《编译原理及编译程序构造》第十四章 第十五章 附录A 附录B  
  2. Compilers: Principles, Techniques, and Tools. By Alfred V. AHO, Ravi SETHI and Jeffrey D. ULLMAN

3. 中文版：编译原理，李建中，姜守旭译，机械工业出版社

编译原理，赵建华，郑滔，戴新宇译，机械工业出版社

4. Advanced Compiler Design and Implementation. By Steven S. Muchnick.

5. 中文版：高级编译器设计与实现，赵克佳，沈志宇译，机械工业出版社

**其他说明**

1.  首次登录请修改密码，务必填写个人联系信息。

2. 直接生成机器码的可以获得加分。

3. 其他优化，例如复制传播、循环强度削弱等，可以自行选作，成功完成都有加分。

   4. 只有难度系数为3的作业才能申优。

**实验指南：**

1. 由于要采用递归子程序法进行语法分析，对不满足条件的文法理论上要进行等价改写；在实现中也可以采用向前查看若干符号的方法。

2. 对于扩充的语法成分的语义处理可参考教材第10章和PASCAL-S源代码

**文档模板：**

见附录

（附录）

《编译技术》课程设计文 档

学号：\_\_\_\_\_\_\_15061050\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_余琦\_\_\_\_\_\_\_\_\_\_\_\_\_

2017年 11月 23日

## 一．需求说明

### 1．文法说明

＜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＞‘]’ }

＜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＞

＜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＞;｜＜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＞   ::=  for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞

＜步长＞    ::=  ＜非零数字＞｛＜数字＞｝

＜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＞

＜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＞‘)’]

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

（6）for语句先执行一次循环体中的语句再进行循环变量是否越界的测试

**文法解读部分**

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

＜加法运算符＞ ::= +｜-

＜乘法运算符＞  ::= \*｜/

＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞   ::= ０｜＜非零数字＞

＜非零数字＞  ::= １｜．．．｜９

句子示例：a1\_c\*Z0

作用条件：定义了字符的组成。此处字符只能是加减符号或乘除符号或字母（包括ascii码为95的下划线）或数字。且包含一对单引号。

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

句子示例：aaaaaa!!!!! ^}]

作用条件：由十进制编码为32,33,35-126的ASCII字符重复0-n次组成的串即是字符串，中间可以有空格，但是函数命名和标识符命名中间不能存在空格，可以有下划线。

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＞

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

句子示例：const int a=1;constb=2;

int x;int y;char z

int huffman(int x){return 1}

char binary(char z){return ‘c’ }

void main(){}

作用条件：程序的大体结构定义。

4. ＜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＞}

＜标识符＞    ::=  ＜字母＞｛＜字母＞｜＜数字＞｝

＜无符号整数＞  ::= ＜非零数字＞｛＜数字＞｝

＜整数＞        ::= ［＋｜－］＜无符号整数＞｜０

句子示例：const int a = 1;const b=2;

作用条件：常量的声明过程,常量的声明过程是分离的。

5. ＜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＞‘]’ }

＜类型标识符＞      ::=  int | char

句子示例： int a[100],b1,c；

作用条件：变量的声明过程，可声明数组且可多次声明

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

＜有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＞

句子示例：int sqrt(int a){

Return a\*a；

}

作用条件：定义有返回值的函数，可以调用此函数获得规定的返回值。

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

＜有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＞

句子示例：void ppp(int b){

b = b + 3；

}

作用条件：定义无返回值的函数，可以调用此函数来对某一数据结构进行处理。

8. ＜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＞

句子示例：switch (a+b){

Case 1 : c = 2\*a

Case 2 : c = 6\*a

Default : c = b

}

作用条件：switch语句中必定要有default 语句来跳出，且case后接的常量必定与表达式的类型一一致

9. ＜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＞‘)’

句子示例：c = Huffman(a) + 3\*b

作用条件：定义了表达式的组成。

10. ＜语句列＞   ::= ｛＜语句＞｝

＜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＞   ::=  for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞

＜步长＞    ::=  ＜非零数字＞｛＜数字＞｝

＜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＞

句子示例：

const int a = 1;

const int b = 2;

int c ;

int d;

for(c=0;c<12;c=c+1){d=d+d;}

作用条件：语句列之间需要以分号分隔。

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

句子示例：scanf(a,b)

作用条件：从控制台读取输入的字符，注意与c语言不同的是，此处没有对读取格式的要求。

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

句子示例：printf(“%d”,a=b+c)

作用条件：输出指定内容到控制台，注意与c语言不同，此处没有格式定义。

13. ＜返回语句＞   ::=  return[‘(’＜表达式＞‘)’]

句子示例：return a+b

作用条件：作为函数的返回值，用以判断程序是否正常结束或返回所需值。

### 2．目标代码说明

目标代码为mips指令代码

### 3. 优化方案\*

## 1．窥孔优化（未实现）  在采用中间代码到目标代码逐条生成的过程中，目标代码中经常会出现代码的冗余或者低效率的指令，为了消除这种指令，我们采用窥孔优化的方式处理。窥孔优化关注在目标指令的一个较短序列上，通过删除冗余代码或者替换更高效和简洁的代码来使得最终得到的目标代码是高质量的。  2．常数合并 （未实现） 如果在编译的时候，对于某些表达式的部分或者全部操作数的值，则此表达式就会被优化，将已知值的操作数进行可以进行的运算后再与变量进行运算。  3.常量替换 （未实现） 在编译之初就将所有常量替换为值，不在对常量进行取址等操作。  4.消除公共子表达式（未实现）

## 二．详细设计

【应包括但不限于以下内容】

### 1．程序结构

error.h

错误处理

### 词法分析

getsym.h

### 语法分析

grammer.h

### 语义分析

### 中间代码生成方法

midcode.h

objectcode.cpp

### 目标代码生成方法

### objectcode.cpp调用其他的四个头文件，即为主函数文件。生成的中间代码在midcode.txt,生成的目标代码在mips.txt中。

### 2．类/方法/函数功能

**getsym部分**

|  |  |
| --- | --- |
| 函数名 | 功能 |
| isSpace()等 | 判断是否为某种字符 |
| reserver() | 返回保留字的序号，若不是返回0 |
| catToken() | 将读到的字符连接起来 |
| clearToken() | 清空token数组 |
| error() | 错误处理程序 |

**语法分析与语义分析**

**grammer.cpp部分**

|  |  |
| --- | --- |
| insertSymTab() | 将函数名和标识符插入符号表 |
| init() | 初始化符号表 |
| searchSymTab() | 查找符号表，得到标签地址，标识符类型等 |
| Integer() | 处理纯数字的因子 |
| Factor() | 因子子处理程序 |
| Term() | 项子处理程序 |
| Exp() | 表达式子处理程序 |
| ConDef() | 常量定义子处理程序 |
| ConExplain() | 常量解释子处理程序 |
| VarDef() | 变量定义子处理程序 |
| VarExplain() | 变量解释子处理程序 |
| Step() | 步长子处理程序 |
| Condition() | 条件子处理程序 |
| Cycle() | 循环语句子处理程序 |
| CondiSen() | 条件语句子处理程序 |
| Assign() | 赋值语句子处理程序 |
| FuncCall1() | 有返回值函数调用子处理程序 |
| FuncCall0() | 无返回值函数调用子处理程序 |
| Scanf() | 读字符子处理程序 |
| Printf() | 打印子处理程序 |
| Return() | 函数返回子处理程序 |
| Main() | 主函数处理程序 |
| FuncDef1() | 有返回值函数定义子处理程序 |
| FuncDef0() | 无返回值函数定义子处理程序 |
| Constant() | 常量子处理程序 |
| Switch() | 情况语句子处理程序 |
| Case() | 子情况语句子处理程序 |
| Default() | 缺省语句子处理程序 |
| Sentence() | 语句子处理程序 |
| SentenceList() | 语句列子处理程序 |
| ComSen() | 复合语句子处理程序 |
| grammer() | 语法语义分析总程序 |

**中间代码生成方法**

**midcode.h部分**

|  |  |
| --- | --- |
| genMidCode() | 生成中间代码，并将结果写入midcode.txt |
| genVar() | 生成临时变量名 |
| genLabel() | 生成标签名 |

**目标代码生成**

**objectcode.cpp部分**

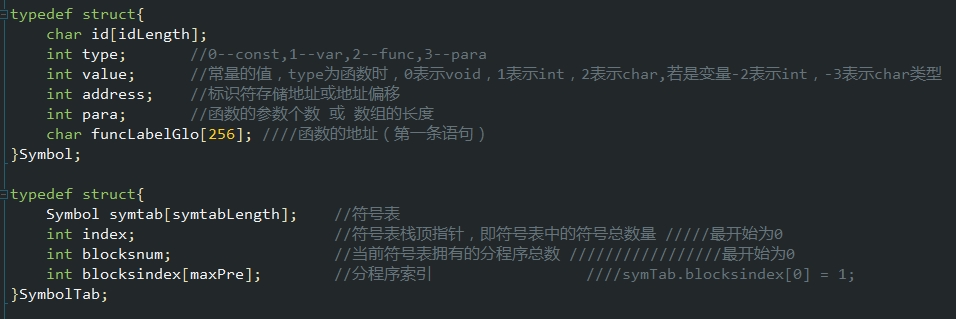
|  |  |
| --- | --- |
| isNum() | 判断是否为数字 |
| findAddr() | 在符号表中查找偏移量 |
| int\_mips() | int变量处理 |
| char\_mips() | char变量处理 |
| inta\_mips() | int变量数组处理 |
| chara\_mips() | char变量数组处理 |
| add\_mips() | 加法处理 |
| minus\_mips() | 减法处理 |
| mul\_mips() | 乘法处理 |
| div\_mips() | 除法处理 |
| compare\_mips() | 条件判断处理程序 |
| pri\_mips() | 打印程序 |
| sca\_mips() | 接收字符处理程序 |
| label\_mips() | 标签处理程序 |
| jmp\_mips() | 无条件跳转处理程序 |
| bnz\_mips() | 满足条件跳转处理程序 |
| bz\_mips() | 不满足条件跳转处理程序 |
| ass\_mips() | 赋值处理 |
| retCall() | 函数返回值赋值处理 |
| ret\_mips() | 函数返回处理 |
| func\_mips() | 函数处理 |
| funcCall\_mips() | 函数调用处理 |
| paraPush() | 参数push处理 |
| para\_mips() | 参数处理 |
| funcConst() | 函数内常量处理 |
| main() | 生成目标代码主程序 |

3．调用依赖关系

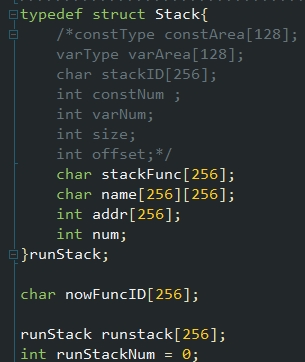
【说明各类之间的关系，方法/函数之间的调用关系】

### 符号表管理方案

语法分析中的符号表



目标代码生成中的符号表



### 5．存储分配方案

|  |
| --- |
| 栈顶 |
| return address |
| Prev abp |
| 参数区 |
| display |
| …… |
| …… |
| 主函数变量定义区 |

### 6. 解释执行程序\*

本程序不解释执行，而是生成mips代码，在mars下运行

### 四元式设计\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 语句类型 | 语句示例 | 语句三元式 | 语句type | mips |
| 函数声明 | int func1() | int,func1,label | 1 |  |
| 函数声明 | char func1() | char,func1,label | 2 |  |
| 函数声明 | void func1() | void,func1,label | 3 |  |
| 函数声明 | void main() | void,main,space | 4 |  |
| 函数定义中的参数 | int func1(int a) | para,int,a | 5 | √ |
| 函数中定义的参数 | int func1(char a) | para,char,a | 6 | √ |
| 函数调用（1） | func1() | call,func1,label | 7 | √ |
| 函数调用中的参数 | func1(a) | push,a,space | 8 | √ |
| 函数返回 | return(x); | ret,x,space | 9 | √ |
| 变量声明 | int i; | var,int,i | 10 |  |
| 变量声明 | char i; | var,char,i | 11 |  |
| 变量声明 | int a[b]; | int,a,b | 12 |  |
| 变量声明 | char a[b]; | char,a,b | 13 |  |
| 常量声明 | const int a = 1; | int,a,1 | 14 |  |
| 常量声明 | const char a = 1; | char,a,1 | 15 |  |
| 运算表达式 | a+b | result,a,b | 16 | √ |
| 运算表达式 | a-b | result,a,b | 17 | √ |
| 运算表达式 | a\*b | result,a,b | 18 | √ |
| 运算表达式 | a/b | result,a,b | 19 | √ |
| 条件判断 | a<b | result,a,b | 20 | √ |
| 条件判断 | a<=b | result,a,b | 21 | √ |
| 条件判断 | a>b | result,a,b | 22 | √ |
| 条件判断 | a>=b | result,a,b | 23 | √ |
| 条件判断 | a!=b | result,a,b | 24 | √ |
| 条件判断 | a==b | result,a,b | 25 | √ |
| 条件跳转 | 无条件跳转 | GOTO,LABEL,space | 26 | √ |
| 条件跳转 | 满足条件跳转 | BNZ,t,LABEL | 27 | √ |
| 条件跳转 | 不满足条件跳转 | BZ,t,LABEL | 28 | √ |
| 赋值语句 | a=b; | a,equal,b | 29 | √ |
| 赋值语句 | a=b[i]; | a,b,i | 30 | √ |
| 赋值语句 | a[i] = b; | a,I,b | 31 | √ |
| 标签 | label1: | label1,space,space | 32 | √ |
| 情况语句 | switch(x){} | switch,x,labeln | 33 |  |
| 情况子语句 | case(a):y=y+1; | label1,case,a | 34 | √ |
| 情况缺省 | default:y=y+1; | label1,default | 35 | √ |
| if语句 | if(x) y=y+1 | if,x,label1 | 36 | √ |
| else语句 | else y=y+1 | else,label1 | 37 | √ |
| 循环语句 | for(){} | for,t,label1 | 38 | √ |
| 读语句 | scanf(x); | scanf,x,int | 39 | √ |
| 赋值语句 | a=1; | a,equal,1 | 40 | √ |
| 写语句 | printf(x); | printf,x,int | 41 | √ |
| 写语句 | printf("x"); | printf,x,string | 42 | √ |
| 标签 |  | label | 43 | √ |
| 函数调用返回 | return(x); | I,equal,RET | 44 | √ |
| 写语句 | printf(x); | printf,x,char | 45 | √ |
| 读语句 | scanf(x); | scanf,x,char | 46 | √ |
| 函数调用（0） | func1(); | call,func1,label | 47 | √ |
| 赋值语句 | printf("x"); | tempVar1,equal,x | 48 |  |

### 8. 目标代码生成方案\*

1. 如何为中间代码选择语义等价且运行效率高的目标指令；

2. 为目标代码、局部变量、全局变量、静态变量、临时变量、常量等指定和划分地址 空间；

3. 由于源程序和中间代码中的变量和数据结构的显示定义都将消失，取而代之的是具 体的寄存器编号和内存空间地址等，因此需要准确合理的将这些寄存器和内存单元 分配给不同类型和结构的变量；

4. 生成和输出目标代码序列到文件。在这一过程中，目标代码的序列和输入的中间代 码序列在顺序上可以一致，也可以不一致，在保证语义不变的前提下，尽量选择更 高执行效率或者更小存储空间的代码序列。

### 9. 优化方案\*

1.常数合并

如果在编译的时候，对于某些表达式的部分或者全部操作数的值，则此表达式就会被优化，将已知值的操作数进行可以进行的运算后再与变量进行运算。

2.常量替换

在编译之初就将所有常量替换为值，不在对常量进行取址等操作。

### 10. 出错处理

}

#ifndef ERROR

#define ERROR

#include <stdio.h>

#include<stdlib.h>

#include<string.h>

void error(int x,int errorRow,int errorCol){

switch(x){

case 0:printf("At line(%d),column(%d):\nError(%d):An IDEN starts with a number!\n",errorRow,errorCol,x); break;

case 1:printf("At line(%d),column(%d):\nError(%d):An UNINT starts with 0!\n",errorRow,errorCol,x); break;

case 2:printf("At line(%d),column(%d):\nError(%d):A single exc!\n",errorRow,errorCol,x); break;

case 3:printf("At line(%d),column(%d):\nError(%d):An illegal letter!\n",errorRow,errorCol,x); break;

case 4:printf("At line(%d),column(%d):\nError(%d):Neither char or int after a CONST!\n",errorRow,errorCol,x); break;

case 5:printf("At line(%d),column(%d):\nError(%d):Not a IDEN after const (char|int)!\n",errorRow,errorCol,x); break;

case 6:printf("At line(%d),column(%d):\nError(%d):An illegal single quote!\n",errorRow,errorCol,x); break;

case 7:printf("At line(%d),column(%d):\nError(%d):An illegal double quote!\n",errorRow,errorCol,x); break;

case 8:printf("At line(%d),column(%d):\nError(%d):A ']' was lost!\n",errorRow,errorCol,x); break;

case 9:printf("At line(%d),column(%d):\nError(%d):An error func!\n",errorRow,errorCol,x); break;

case 10:printf("At line(%d),column(%d):\nError(%d):An error func!\n",errorRow,errorCol,x); break;

case 11:printf("At line(%d),column(%d):\nError(%d):A SEMI was lost!\n",errorRow,errorCol,x); break;

case 12:printf("At line(%d),column(%d):\nError(%d):Array subscript is not a number!\n",errorRow,errorCol,x); break;

case 13:printf("At line(%d),column(%d):\nError(%d): Not an ID after (int|char)!\n",errorRow,errorCol,x); break;

case 14:printf("At line(%d),column(%d):\nError(%d):Step is not a num!\n",errorRow,errorCol,x); break;

case 15:printf("At line(%d),column(%d):\nError(%d):A '(' was lost!\n",errorRow,errorCol,x); break;

case 16:printf("At line(%d),column(%d):\nError(%d):Not an ID after '('!\n",errorRow,errorCol,x); break;

case 17:printf("At line(%d),column(%d):\nError(%d):Not an equal in Cycle!\n",errorRow,errorCol,x); break;

case 18:printf("At line(%d),column(%d):\nError(%d):Step is not an ID!\n",errorRow,errorCol,x); break;

case 19:printf("At line(%d),column(%d):\nError(%d):A '=' is lost!\n",errorRow,errorCol,x); break;

case 20:printf("At line(%d),column(%d):\nError(%d):Not a (+|-) in step!\n",errorRow,errorCol,x); break;

case 21:printf("At line(%d),column(%d):\nError(%d):A ')' was lost!\n",errorRow,errorCol,x); break;

case 22:printf("At line(%d),column(%d):\nError(%d):Paranums are different when calling!\n",errorRow,errorCol,x);break;

case 23:printf("At line(%d),column(%d):\nError(%d):Printf ends without ')'!\n",errorRow,errorCol,x); break;

case 24:printf("At line(%d),column(%d):\nError(%d):Para without (char|int)!\n",errorRow,errorCol,x); break;

case 25:printf("At line(%d),column(%d):\nError(%d):A LBRA3 was lost!\n",errorRow,errorCol,x); break;

case 26:printf("At line(%d),column(%d):\nError(%d):Not an ID after void!\n",errorRow,errorCol,x); break;

case 27:printf("At line(%d),column(%d):\nError(%d):A default sentence was lost!\n",errorRow,errorCol,x); break;

case 28:printf("At line(%d),column(%d):\nError(%d):A ':' was lost! \n",errorRow,errorCol,x); break;

case 29:printf("At line(%d),column(%d):\nError(%d):SwitchExp and CaseExp have different types!\n",errorRow,errorCol,x); break;

default: break;

}

}

#endif

错误处理方式:

1.递归子程序发现错误立即调用error()并传递出错信号；

2.error函数通过接收到的出错信号打印出出错信息和位置信息；

3.通过出错类型型号和位置判断应该如何进行跳读；

4.继续编译，直到文件结束。

## 三．操作说明

### 1．运行环境

64位windows10及以上 vs2010及以上

### 2．操作步骤

打开objectcode.cpp运行，输入测试文件路径，回车，在mips.txt中得到目标代码。

## 四．测试报告

### 1．测试程序及测试结果

（1）

int mod(int x, int y)

{

int z;

z = x - x/y\*y;

return(z);

}

int gcd(int a, int b)

{

if (b==0) return(0);

else

if (mod(a,b)==0)

return(b);

else return(gcd(b, mod(a,b)));

}

void main(){

printf(gcd(15,24));

}

测试结果：正确输出 3，覆盖范围是函数调用，函数定义，函数传参，函数循环调用，多重传参。

（2）void defi(int value){

const char D1 = 'd',E1 = 'e',F1 = 'f',G1 = '+',H1 = '-';

const char I1 = '8';

int a,b,c,d[10],e[4],f;

int m[7],n[8],p,q,h[9];

char i,o[4],r[5],s,t,u[65],v\_\_\_;

a = value;

d[5] = value;

a = D1;

v\_\_\_ = G1;

printf("variable and const defination passed!");

}

void main(){

defi(1);

}

测试结果：正确输出variable and const defination passed!，测试了字符数组，字符的赋值。

（3）int factorial(int f) {

if (f <= 1) {

return (1);

} else {

return (f \* factorial(f-1));

}

}

void main（）

{

int x;

scanf(x);

printf(factorial(x));

}

测试结果：正确输出所输入的斐波那契数列的第几个数字。

（4）

int fibo(int n)

{

if (n<0) return(-1);

else if (n==0) return(0);

else if (n==1) return(1);

else if (n>MAX) return(-2);

else return(fibo(n-1)+fibo(n-2));

}

void main（）

{

int x;

scanf(x);

printf(fibol(x));

}

测试结果：正确输出所输入的斐波那契数列的第几个数字。

（5）void main(){

int x;

int y;

scanf(x);

scanf(y);

switch(x){

case 1: printf(1);

case 2 :printf(“test2”);

case 3 :{switch(y){

case 1: printf(“test3”);

case 3 :printf(“test4”);

default:printf(“default2”);

}

default :printf(“default1”);

}

}

}

测试结果：测试了switch语句能否正确执行，以及若switch语句之下嵌套switch语句是否正确。

int mod(int x, int y)

{

int z;

z = x - x/y\*y;

}

int gcd(int a, int b)

{

if (b==0) return(0);

else

if (mod(a,b)==0)

return(b);

else return(gcd(b, mod(a,b)));

}

void main(){

printf(gcd(15,24));

}

测试结果：输出函数定义为int返回值类型，却无return语句。

（7）void defi(int value){

const char D1 = 'd',E1 = 'e',F1 = 'f',G1 = '+',H1 = '-';

const char I1 = '8';

int a,b,c,e[4],f;

int m[7],n[8],p,q,h[9];

char i,o[4],r[5],s,t,u[65],v\_\_\_;

char d[10];

a = value;

d[5] = value;

a = D1;

v\_\_\_ = G1;

printf("variable and const defination passed!");

}

void main(){

defi(1);

}

测试结果：数组d为char类型数组，但是赋值时直接ass为int类型，报错但是依旧执行。

（8）int factorial(int f) {

if (f <= 1) {

return (‘a’);

} else {

return (f \* factorial(f-1));

}

}

void main（）

{

int x;

scanf(x);

printf(factorial(x));

}

测试结果：测试的函数式int类型，却返回’a’,报错

（9）int fibo(int n)

{

if (n<0) return(-1);

elseif (n==0) return(0);

else if (n==1) return(1);

else if (n>MAX) return(-2);

else return(fibo(n-1)+fibo(n-2));

}

void main（）

{

int x;

scanf(x);

printf(fibol(x));

}

测试结果：不存在elseif的标识符，报错

（10）void main(){

int x;

int y;

scanf(x);

scanf(y);

switch(x){

case 1: printf(1);

case 2 :printf(“test2”);

case 3 :{switch(y){

case 1: printf(“test3”);

case 3 :printf(“test4”);

default:printf(“default2”);

}

}

}

}

测试结果：缺少一个default语句，报错。

### 测试结果分析

覆盖了程序中的一些基本操作，如数组的赋值，数组的声明，数组的操作，函数的调用，函数的嵌套调用，条件语句，语句列空语句，switch语句以及各种语句之间的嵌套。

## 总结感想

课程设计开始的时候，开始我是准备选2难度的课设题目，想着能把这门课过了就行，但是我的室友们都选了难度三，想着这是北航计算机学院最后一门比较核心的课程设计了，觉得不挑战一下自己不有点遗憾，所以也算是在自己的鼓励和室友的怂恿下选了难度三的课设题目，刚开始写词法分析的时候还好，每周都能按时完成任务，但是在写语法分析的时候，由于在理论课上的疏忽，第一遍的时候没有想过预读取一个标识符，这样写出来的程序bug非常多而且难以调试，因为涉及到了fseek函数中的回退步数问题，我直到周六的晚上才发现这个问题，所以只能在第二天坐在宿舍里含泪重新写了一遍语法分析的程序，这样写出的程序有很多的bug，而我也没能及时地测出来并改正，为后来的工作加大了很多难度。后来写到语义分析，生成中间代码和目标代码的时候，出现了非常非常多的bug，当我没日没夜的改完的时候，发现我基本上等于把我的程序又重新设计了一遍。所以说程序的设计和整体把握还是非常重要的。

所以我觉得应该在每一周做完相关的程序设计的时候，都应该多用一些测试程序来测试程序的bug，尽量在前面改掉大多数的bug，即保证每一步往下面做的时候，前面的部分要基本正确，不然会给整个程序带来非常多的错误，其次在进行程序设计的时候应该少用全局变量，这在之前的数据结构作业中没有太多的要求，但是在编译器的设计中会因为很多函数的迭代吧原来预想的全局变量的值冲掉，这样就带来了很多不知名的错误。

总而言之，我的课程感想就是面对无穷无尽的bug不要气馁，一个一个的去解决就好了，办法总比困难多，另外要珍惜获得额外测试的机会，不然每一次的测试出了错就会很惊险。以张莉老师的话结尾吧，“教育就是让受教育者通过学习逐步获得自信的过程”，编译课设增强了我的debug能力，也使得我变得更加自信。