《编译技术》课程设计

文 档

学号：16231256

姓名：李天宇

2019年 1月 4日

## 一．需求说明

### 1．文法说明

文法为扩充的C0文法，其中特殊部分为while和switch，这里给出经过我改编后的文法。不难看出，改编后的文法是LL（1）文法，是可以使用地柜下降分析法来分析的文法。

＜加法运算符＞ ::= +｜-

加法运算符只能是+或-；

举例：+、-

＜乘法运算符＞ ::= \*｜/

乘法运算符只能是\*或/；

举例：\*、/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

一共有六种关系运算符（小、小等、大、大等、不等、等）；

举例：略

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

字母包含a-z、A-Z和\_(下划线符号)；

举例：略

＜数字＞ ::= ０｜＜非零数字＞

数字分是0 和 非零（1-9）；

举例：略

＜非零数字＞ ::= １｜．．．｜９

非零数字是1-9；

举例：略

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

字符指+、z、0这种，必须是一位；

举例：’+’、’-’、’c’、’5’

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

字符串是用双引号括起来的一串有限定的字符（可以使空的）；

举例：””、” !#”、”awea{}”

＜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＞

程序是 常量说明（可省）、变量说明（可省）、任意多个有无返回值函数定义（同样可省）和必不可少的唯一主函数组成；

举例：略

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

常量说明由至少一个“const<常量定义>;”组成，const、<常量定义>、;为一组；

举例： const int flag = 1, flag\_2 = -1; const char class = A;、const char grade = 7;

＜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＞}

<常量定义>是组成常量说明的部分，由int<标识符>=整数，后续逗号+<标识符>=整数循环或者char<标识符>=字符，后续逗号+<标识符>=<字符>组成；一个常量定义只能是int或者char的，不会混合出现。

举例：int flag = +6 、int flag = -6,flag2 = 0、char \_class =A, Grade = 9

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝| 0

<无符号整数>是0或者由非零数字开头的一位、多位数；

举例：0、3、76583

＜整数＞ ::= ［＋｜－］＜无符号整数＞

<整数>是可有可无的正负号和<无符号整数>组成的，其中若无正负号则默认为正；

举例：+1、-74、32、+0、0

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

<标识符>必须是字幕开头的，由字母或数字组成的一位或者多位串；

举例：name、\_test1、fun\_1

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

<声明头部>是 int或者char开头 接着<标识符>的一种东西（仅仅被用在了有返回值的函数定义上），由于这个特性，这个标识符就一定会被保存为函数名；

举例：int max、char \_class

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

<变量说明>是由至少一个的<变量定义>加分号组成的；

举例：略

＜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＞']' )} //＜无符号整数＞表示数组元素的个数，其值需大于0

<变量定义>是组成变量说明的关键部分（之一（还有分号）），由<类型标识符>开头（int或者char），接着接一个<标识符>（一堆字母和数字组成的那玩意），或者是<标识符>[无符号整数]表示要建立一个数组（该无符号整数不能是0），这里的变量定义是不能给变量赋初值的；

举例：int count, count\_2、char alpha[7]、int flags[10],count

＜常量＞ ::= ＜整数＞|＜字符＞

<常量>是一个整数或者一个字符；

举例：+99、’9’、9

＜类型标识符＞ ::= int | char

<类型标识符>简单易懂，就是int和char;

举例：略

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

<有返回值函数定义>是一个有返回值的函数的定义，其中参数表用小括号括起来，复合语句用大括号括起来；

举例: int max(<参数表>){<复合语句>}（这个int是表示返回类型的）

＜无返回值函数定义＞ ::= void <无返回值函数子句>

<无返回值函数子句> ::= ＜标识符＞'('＜参数表＞')''{'＜复合语句＞'}'

<无返回值函数定义>和“有返回值函数定义”区别仅在开头的void；

举例：void max(<参数表>){<复合语句>}

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

<复合语句>由常量说明（可省）、变量说明（可省）和必须的语句列组成（注意顺序）；

举例：略

＜参数表＞ ::= ＜参数＞{,＜参数＞}| ＜空>

<参数表>是用在函数声明里面的，可以没有（空）或者有一个或者有多个并用逗号隔开；

举例：int a、、int a, char b, int c

＜参数＞ ::= ＜类型标识符＞＜标识符＞

<参数>用来形成参数表，结构是类型标识符+标识符；

举例: int a、int b、 char c

＜主函数＞ ::= void <主函数子句>

<主函数子句> ::= main'('')''{'＜复合语句＞'}'

<主函数>是<程序>的最后一个部分也是不可省略的，以 void main ()开头之后是{<复合语句>}；

举例：void main(){<复合语句>}

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}   //[+|-]只作用于第一个<项>

<表达式>是一个或者多个项，用+-链接（第一个项前可以有+-表示第一个项的正负性质）；

举例：略

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

<项>是组成表达式的部分，一个项是多个因子\*/链接而成的；

举例：略

＜因子＞ ::= ＜标识符＞｜＜标识符＞'['＜表达式＞']'|'('＜表达式＞')'｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞

<因子>顾名思义就是因子，可以使一个标识符，或者是一个标识符[表达式]（数组元素），或是括号括起来的表达式，或是一个常量，或是一个有返回值函数调用语句（用这个返回值）；

举例：略

＜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＞;

<语句>包含了所有的语句，除了条件语句（if）、循环语句（while）、情况语句（switch）之外都需要在最后又分号表示结束（因为这三种语句毕竟还没有结束，所以也不是分号）；

举例：略

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞

<赋值语句>是给已经定义的标识符或者数组元素赋值的语句，要注意这里的分号是没有的，是在语句那里有个分号；

举例: flag = 1、name[3] = ’l’、

＜条件语句＞ ::= if '('＜条件＞')'＜语句＞

<条件语句> 是if条件语句，直接举例，我们发现这里的if语句只能执行下一个语句；

举例：if(<条件>)<语句>（条件是一个或者两个表达式通过关系运算（<），0为假其他真）

＜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

<条件>给条件语句充当判断条件或者给循环语句充当判断条件；

举例：flag+7、flag\*6 > 44

＜循环语句＞ ::= while '('＜条件＞')'＜语句＞

<循环语句>是while循环，和if语句格式相似；

举例：略

＜情况语句＞ ::= switch '('＜表达式＞')' '{'＜情况表＞＜缺省＞ '}'

<情况语句>即为switch、case组合语句，这里的switch默认执行了一个直接跳出;

举例：switch(count){<情况表><缺省>}（情况表是多个情况子，缺省就是default那里）

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

<情况表>必须至少有一个情况子语句；

举例：略

＜情况子语句＞ ::= case＜常量＞：＜语句＞

<情况子语句>用于组成情况表，一个子语句是case 常量 冒号 语句 组成的；

举例： case 4 : <语句>

＜缺省＞ ::= default : ＜语句＞|＜空＞

<缺省>用来处理不符合所有case的情况下该怎么办，格式是没有或者是default冒号语句；

举例：（空）、default : count++

＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

用来调用一个有返回值的函数，必须已经被定义才行；

举例： max(a,b)（max已被早先定义）

＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

和上一个差不多，不过被定义的应该是一个没有返回值的函数

举例：略

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

<值参数表>用来填充调用时括号内的部分，可以使空或一个或者多个表达式用逗号分隔；

举例：略

＜语句列＞ ::= ｛＜语句＞｝

<语句列>可以是空的，也可以是一个或者多个语句；

举例：略

＜读语句＞ ::= scanf '('＜标识符＞{,＜标识符＞}')'

<读语句>；

举例：scanf(name,name\_2,name\_3)

＜写语句＞ ::= printf '(' ＜字符串＞,＜表达式＞ ')'| printf '('＜字符串＞ ')'| printf '('＜表达式＞')'

<写语句>，如果想要输出字符串的话必然在表达式前面而且和表达式用逗号隔开，一次只能输出一个字符串+表达式、一个字符串、一个表达式（表达式会输出ascii的值除非是单个char变量或者常量）；

举例：printf(”hello World!”,name)、 printf(name)、 printf(”hello World！”)

＜返回语句＞ ::= return['('＜表达式＞')']

<用来返回> return和可省的表达式（需要检测类型）；

举例：return、 return(count-3)

<全局变量定义>

<有返回值函数定义子句>

上方两个函数没有具体的定义式，是为了消除文法的二义性而诞生的两个函数，特此列在这里

<函数调用语句>

上方这个语句是为了书写，语法分析时不能区分函数调用有无返回值，只能统一称为函数调用语句

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）写语句中，字符串原样输出，单个char类型的变量或常量输出字符，其他表达式按整型输出

（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

目标代码为mips汇编代码。其中我们需要使用的代码有以下几类：

1. 运算类代码：

有add，addi，sub，subi….等指令。

1. 存取类代码：

有lw，lb，sw，sb等存取指令。

1. 输入输出类代码：

输入输出主要通过syscall来实现。

### 3. 优化方案

由于时间原因，没能完成优化，这里仅仅使用了不完全的常数合并优化。

## 二．详细设计

### 1．程序结构

程序由以下部分组成：

**1.1词法分析程序：**

词法分析程序用于读入数据、可以自主去掉无用字符直接读取到下一个有程序意义的字符；如若下一个字符为非法字符，将会将词法错误类型存入相应的位置方便错误处理程序处理。

**1.2语法分析程序：**

语法分析程序通过构建相应的流程图来判断语法是否符合C0文法；如果不符合语法则会触发对应的错误处理程序来处理出现的语法错。

**1.3语义分析程序：**

混在语法分析程序的主干中，当一个合格的语句在被判定时，利用对应的动作语句来处理应当处理的部分，相当于一个语法制导的翻译程序；如果某处存在语义错误，则会触发相应的语义错误处理程序来处理出现的语义错。语义分析程序能够生成中间代码（未优化），即四元式序列，并将其保存到需要的目录下的文件中。

**1.4目标代码生成程序：**

目标代码生成程序是根据中间代码（四元式序列）直接生成MIPS汇编代码，可以线性地完成任务（即读入一个四元式，生成对应的代码，再读入四元式……）。

**1.5优化程序：**

优化程序通过读入四元式序列，输出优化后的四元式序列，这里仅仅用到了不完全的常数合并。

### 2．类/方法/函数功能

#### 2.1词法分析部分函数

词法分析部分一共有以下几种函数：

1. nextsym()函数，用来获取下一个符号，供给外部调用（即外部调用次函数获得下一符号）；
2. judge()函数，用来分类分析标识符、数字等的函数，相当于一个分类器；
3. str\_judge()、chr\_judge()、num\_judge()、rsv\_judge()，这些函数用于具体的判断string、char等语法成分；
4. 其他函数，例如isnot、isnum、islet等，是打包好的子函数供上述函数调用（简化逻辑）。

#### 2.2语法分析部分函数

由于使用递归下降分析法，故语法分析部分函数是嵌套下降的，即根据文法来向下调用。由于语法成分较多，这里不再列举所有函数。其中重要的函数就是isProgram()函数，调用此函数（在此之前先调用nextsym()）即可开始语法分析。详细内容见gramma.c与gramma.h内容。

#### 2.3语义分析和中间代码生成函数

使用语法制导的翻译技术进行语义分析和中间代码生成。与语义分析与语法分析内容在同一源文件下（即gramma.c与gramma.h），中间代码生成是通过调用中间代码生成函数（ICgen.c与ICgen.h）实现的。也就是说中间代码生成函数相当于属性翻译文法的动作符号，由于内容过多不予罗列。

#### 2.4最终代码生成函数

读取生成的中间代码并依据已知符号表，生成最终代码。值得注意的是，最终代码生成时，中间代码已经完全生成完毕。最终代码生成函数文件为translate.c与translate.h，由于内容过多不予罗列。

#### 2.5优化函数

由于时间关系，没有完成既定的优化部分，仅仅进行了不完全的常量传播优化，这种优化是通过继承属性和综合属性，在语义分析阶段进行判断的，并且在中间代码生成时已经完成。

#### 2.6错误处理函数

主要由两部分完成，即报错函数和错误恢复函数。报错函数负责报出具体错误位置和错误信息，错误恢复函数负责尽量恢复到正常的分析中，以便完成更加多的编译检查（即尽可能通过一遍编译报出潜在的所有错误）。

1. 报错函数

函数err(int i)，接受一个整形参数作为错误编号，并根据存储的错误表进行报错输出，输出行号（位置）、错误信息、英文信息。

1. 错误恢复函数

由于错误恢复函数的本质是读至下一个可接受符号，故错误恢复函数写在了词法分析函数的源文件中（直接调用nextsym()）：

\_\_nextSemicolon()读到下一个分号；

\_\_nextCommaOrSemicolon()读到下一个逗号或者分号；

\_\_nextClosingBraces()读到下一个右大括号；

\_\_nextReservedSymbol()读到下一个保留字；

\_\_nextCommaOrRightParenthesesOrSemicolon()读到下一个右小括号或者逗号或者分号。

### 3．调用依赖关系

本编译器使用C语言实现，故仅有函数调用关系，没有类的关系。

编译器内容分为5大部分，在这五大部分每部分内部的调用不再列举（例如地柜下降的调用过于庞大，麻烦也没必要在这里列出）。

**这里给出主要部分间的重要调用关系：**

1. nextsym()被语法分析调用：nextsym()函数本身可以使编译器阅读下一符号，故被语法分析调用。
2. ICgen内函数被语法分析（语义分析）调用：用于生成中间代码。
3. Symtable内函数被语法分析、代码生成（包括中间、最红）调用：用于保存、查询符号表等操作。
4. 错误函数（err）和恢复函数被语法分析（语义分析）调用：用于报错和处理语法、语义错误。

⑤Main函数调用其他部分：用于开始分析、读取文件名、打印信息等操作。

### 4．符号表管理方案

数据结构如下：

|  |
| --- |
| /\*--------------------------------------------------  符号表项定义  符号名： name  符号名长度：namelen  符号种类： kind 常量（CONSTANT）变量（VARIABLE）  符号类型： type 整型（INT） 字符型（CHAR）  是否是数组：array 非（NOTARR） 是（ARR）  数组上界： upbound 例：char[3] upbound=2 char upbound=0  常量值 value 常量值为该常量的值，变量的此处为0或后续赋值  相对于第一个参数（符号）的偏移量： offset（运行栈要用）  是否是确定的值： definite constant类型为1，其余为0  --------------------------------------------------\*/  typedef struct symbolTableTerm {  char name[IDE\_MAX\_LEN];  int namelen;  int kind;  int type;  int array;  int upbound;  int value;  int offset;  int definite;  }symterm;  /\*--------------------------------------------------  符号表定义  符号表项的组： sym[MAX\_SYM\_LEN] //最多容纳符号数量（包括参数）  函数中参数的数量: numofpara  已使用的项数： used  这些参数将在运行栈中占用的空间： usespace 初始化为0  --------------------------------------------------\*/  typedef struct symbolTable {  symterm sym[MAX\_SYM\_NUM];  int numofpara;  int used;  int usespace;  }symtable;  /\*--------------------------------------------------  函数表项定义  函数名： name  函数名长度： namelen  函数返回值类型： value 整型（INT）字符型（CHAR）无返回值（VOID）  符号表指针: stp //symbol table pointer  --------------------------------------------------\*/  typedef struct functionTableTerm {  char name[IDE\_MAX\_LEN];  int namelen;  int value;  symtable \*stp;  }funterm;  /\*--------------------------------------------------  函数表定义  函数表项的组： fun[MAX\_FUN\_NUM] //最多定义函数数量（包括main）  已登录的函数数量： used  --------------------------------------------------\*/  typedef struct functionTable {  funterm fun[MAX\_FUN\_NUM];  int used;  }funtable; |

管理算法如下：

|  |
| --- |
| funtable \*newFunTable();  创建新函数表  int searchFun(char \*name, int namelen);  搜索函数  int insertFun(char \*name, int namelen, int value);  插入函数  void showFunTable();  展示函数表  symtable \*newSymTable();  创建新符号表  int searchSym(char \*paraname, int paralen);  在当前环境搜索符号  int searchGlobalSym(char \*paraname, int paralen);  在全局符号表搜索符号  int addPara(char \*paraname, int paralen, int paratype, int definite);  给当前环境添加参数  int addSym(char \*symname, int symlen, int symkind, int symtype, int symarray, int symupbound,int symvalue, int definite);  给当前环境添加常量、变量  void showCurSym();  展示当前符号表  void showGloSym();  展示全局符号表 |

### 5．存储分配方案

对于全局定义的常量和变量，将会在data区为他们分配空间（即全局区），在mips中是存储在.data范围内的。

对于函数中的参数和定义的常量、变量，则会在栈区为他们分配空间，分配空间的位置顺序和定义他们时的位置顺序是完全相同的，而且他们在运行时相对于sp指针的偏移量是与符号表中表示偏移量的值完全相同的。

运行栈结构大致如下：

|  |
| --- |
| $ra value |
| Last Sp address =last sp |
| Const def1 |
| Const def2 |
| Variable def1 |
| $t9 |
| $t8 |
| $t7 |
| $t6 |
| $t5 |
| $t4 |
| $t3 |
| $t2 |
| $t1 |
| $t0 |
| $ra value |
| Last Sp address = last sp |
| Variable def1 |
| Parameter2 |
| Parameter1 |

解释说明：从上到下表示地址递减，即上高下低。Main函数运行前sp指向表格最上面的黑线处，Main函数运行时sp指向黄色格子下方的地址（也就是黄格子的基地址）。

**函数调用别的函数时的工作：**

1. 先将正在使用的临时寄存器保存至相应位置；
2. 将当前sp寄存器内的值存入sp-8的位置；
3. 将fp寄存器的值置为sp - (被调用函数体积 + 48)，其中48包括10个临时寄存器的保存空间、ra保存空间、sp返回保存空间（sp - 8）；
4. 以fp为基地址，将参数按照顺序依次传入 Parameter1、Parameter2处；
5. 将sp指针的值置为fp的值；
6. j到对应函数的label。

**函数被调用后的工作：**

1. 需要将自己的常量值放入运行栈中；
2. 并且存放RA寄存器的值到对应位置。

**函数返回时的工作：**

1. 将a0寄存器内的值置为返回值；
2. 将ra值置为栈中对应位置的值；
3. 将sp置为栈中对应位置的值；
4. JR $ra以返回原代码处。

**返回后的函数如需读取返回值（通常是立即使用，故不会有覆盖问题）：**

1. 取到a0处的值。根据返回值类型不同使用lb或lw指令。

总运行方式是在全局变量定义完成后生成分配main运行地址的指令和跳转到main的指令；并在这些指令之后紧接着退出指令（main返回后直接退出程序）。

注：由于mips执行操作的问题，会在必要的地方使用4Byte对齐操作。这也导致了数据段和栈区的空间有缝隙（也就是没有被利用的部分），是一种浪费存储空间的做法。

### 6. 解释执行程序

Mips代码使用Mars运行，没有解释执行程序。

### 7. 四元式设计

7.1 函数声明 int foo(int a, int b)

|  |
| --- |
| 中间代码 |
| LABEL FUN\_foo: |

7.2 常量与变量声明

①全局常量：

GLOCST constname 4(空间大小)

GLOCSTASS （给全局常量赋值）

LI t1 = 0

SW t1 constname

…

②全局变量：

GLOVARSTART

GLOVAR change1 4

GLOVAR change2 12

…

GLOVAREND

③局部常量会在函数中给运行栈对应位置赋值

LI t1 = 3

SW t1 0(sp)

…

④局部变量没有中间代码

7.3语句中的函数调用 x = num + foo(1, 2)

|  |
| --- |
| 中间代码 |
| LW t1 = num |
| SUB fp,sp,size |
| LI t2 = 1 |
| SW t2 = …(fp) |
| LI t2 = 2 |
| SW t2 = …(fp) |
| JAL FUN\_foo |
| MOVE t2 = a0 |
| ADD t1 = t1 + t2 |
| SW t1 x |

7.4函数调用（作为单个语句） foo(2, x)

|  |
| --- |
| SUB fp = sp - size |
| LI t2 = 1 |
| SW t2 = …(fp) |
| LI t2 = 2 |
| SW t2 = …(fp) |

7.5表达式 a\* (b+ c)

|  |
| --- |
| 中间代码 |
| LW t1 = a |
| LW t2 = b |
| LW t3 = c |
| ADD t2 = t2+t3 |
| MUL t1 = t1 \* t2 |

7.6赋值语句 t1=t2

|  |
| --- |
| 中间代码 |
| MOVE t1=t2 |

7.7条件语句头(if) if(x >= y)

|  |
| --- |
| 中间代码 |
| LI t1 = x（表达式） |
| LI t2 = y（表达式） |
| BLT t1 t2 IF\_i\_END |

7.8 数组变量 a[i]

|  |
| --- |
| 中间代码 |
| MUL t1 = t1 \* i  ADD t1 = t1 + sp  LW/LB t2 (t1)… |
|
|

7.9循环语句头(while)

跟if相似，只是在WHILE\_i\_END标识符前会加入跳转指令，跳回条件判断处。不再列举。

7.10情况语句（将会被处理为多个条件语句）

跟多个if相似，而且第二个表达式必定为常量型，是多个if的结合版。不再列举。

### 8. 目标代码生成方案

根据中间代码的开头字段决定后续的读入方法和生成方法。由于中间代码生成时特意为了生成目标代码预留了相应的处理措施，故可以直接翻译。详见translate.c。

### 9. 优化方案

由于时间关系，没能完成既定优化目标，仅仅进行了不完全的常量传播优化。

### 10. 出错处理

传入规划好的参数，将错误信息打印输出到错误文档中。错误信息将包含行数、中文信息、英文信息。

具体错误信息列表如下：

|  |
| --- |
| //编译时词法错、语法错、语义错-------------------------------------------------  //index 错误信息 英文信息  //1 非法字符 invalid symbol  //2 字符串内存在非法字符 invalid symbol in string  //3 缺少双引号来结束字符串 need double quatation mark to end a string  //4 字符类型内的值不合法 invalid symbol of char  //5 缺少分号 need semicolon ";"  //6 缺少等号 need equal mark "="  //7 保留字错误 wrong reserved symbol  //8 缺少类型保留字 need reserved symbol of type "int"or"char"  //9 缺少右方括号 need closing brackets "]"  //10 缺少右大括号 need closing braces "}"  //11 缺少左大括号 need left braces "{"  //12 缺少右括号 need right parentheses ")"  //13 缺少左括号 need left parentheses "("  //14 缺少标识符 need an identifier  //15 缺少左方括号 need left brackets "["  //16 缺少冒号 need colon ":"  //17 缺少逗号 need comma ","  //18 缺少main保留字 need reserved symbol "main"  //19 需要一个因子 need divisor  //20 错误符号 wrong symbol  //21 缺少main函数定义 need defination of main  //22 缺少等号或左括号 need equal mark "=" or left parentheses"("  //23 缺少等号或左方括号 need equal mark "=" or left brackets "["  //24 缺少逗号或右括号 need comma "," or right parentheses ")"  //25 缺少逗号或左方括号 need comma "," or left brackets "["  //26 缺少无符号整数 need an unsigned integer  //27 缺少整数 need an integer  //28 缺少字符 need a char  //29 缺少常量 need a constant value  //30 未定义的函数 using of a function without declaration  //31 应使用有返回值函数 should be a using of function with return value  //32 使用了未定义的标识符 using of a identifier without declaration  //33 缺少表达式 need an expression  //34 需要整型表达式 need an int expression  //35 数组下标值越界 invalid value of index for array  //36 赋值语句左右类型不同 wrong type of assignment  //37 不能给数组值或常量赋值 invalid assignment to an array or constant value  //38 Case类型与表达式不一致 different type of case and switch  //39 参数数量错误 wrong amount of parameters  //40 参数类型错误 wrong type of parameter  //41 必须定义void mian the return value of main must be void  //42 main后多余字符 too much symbols after main definition  //43 代码非正常结束，可能需要大括号等 wrong ending of the code, maybe need right braces "}"  //-----------------------------------------------------------------------------  //symbol table errors----------------------------------------------------------  //index error-message English-message  //1 函数声明过多 can't insert: declare too much functions  //2 函数重复声明 can't insert: function having the same name has already been declared  //3 参数过多 this function has too much parameters  //4 参数同名 this function has declared a parameter having the same name  //5 符号声明过多 this function has too much declare of symbols  //6 重复声明符号 this function has declared a symbol having the same name  //7 符号表初始化失败 error happened when initializing symbol tables  //----------------------------------------------------------------------------- |

出错处理函数err(int) 和 tableErr(int)接受错误序号并打印相关信息到文件。后续出错处理由语法分析自行完成（需要根据当前所处的语法成分来分析需要忽略多少信息）。

## 三．操作说明

### 1．运行环境

本编译器使用Microsoft Visual Studio 2017 Community Windows版本进行开发，使用编程语言为C语言。可以接受命令行输入的文件路径。

### 2．操作步骤

打开C0\_Compiler项目（.sln），运行程序后根据命令行内提示输入文件路径名即可编译生成中间代码、目标代码等。出错信息将会输出到命令行内。

**注意事项：**

若编译提示不安全等，需要在 项目（Co\_compile）右键属性→C/C++→预处理器→预处理器定义 这一项加入\_CRT\_SECURE\_NO\_DEPRECATE。

若命令行窗口在运行完成后直接退出，可以在 项目（Co\_compile）右键属性→连接器→系统→子系统 一栏选择“控制台(/SUBSYSTEM:CONSOLE)”，即可。

## 四．测试报告

### 1．测试程序及测试结果

**测试程序详见测试程序文件夹，测试结果在下面给出：**

#### 16231256\_测试程序1（即完美覆盖版）输出结果：

1!

1>0!

0>=0!

0<=0!

1!=0!

0==0!

5

2

x

y

z

0

1（输入）

1（输入）

flag1=1!

flag2=1!

my\_print1!

my\_print2! 5

my\_print3 d

t

m

a

0

! 8

! 10

0

0

0

1

-1

123456789

-987654321

A

b

C

164

3

2

1

#### 16231256\_测试程序2输出结果：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

#### 16231256\_测试程序3输出结果：

114514（输入）

1414514（输入）

66（输入）

233（输入）

988（输入）

-23（输入）

0（输入）

77（输入）

-55（输入）

1（输入）

-55（输出）

-23（输出）

0（输出）

1（输出）

66（输出）

77（输出）

233（输出）

988（输出）

114514（输出）

1414514（输出）

#### 16231256\_测试程序4输出结果：

12（输入）

144（输出）

#### 16231256\_测试程序5输出结果：

21（输入）

14（输入）

7（输出）

#### 16231256\_错误程序1输出结果：

line 144 : 代码非正常结束，可能需要大括号 wrong ending of the code, maybe need right braces "}"

#### 16231256\_错误程序2出结果：

line 9 : 缺少逗号或左方括号 need comma "," or left brackets "["

line 145 : 缺少分号 need semicolon ";"

line 145 : 缺少main函数定义 need defination of main

#### 16231256\_错误程序3输出结果：

line 3 : 缺少整数 need an integer

line 145 : 缺少分号 need semicolon ";"

line 145 : 代码非正常结束，可能需要大括号 wrong ending of the code, maybe need right braces "}"

#### 16231256\_错误程序4输出结果：

line 11 : 重复声明符号 this function has declared a symbol having the same name

#### 16231256\_错误程序5输出结果：

line 54 : 需要整型表达式 need an int expression

line 142 : 代码非正常结束，可能需要大括号 wrong ending of the code, maybe need right braces "}"

### 2．测试结果分析

#### 测试程序1结果分析：

根据文法语义，测试结果正确无误。并且输出按行分开，方便观察输出结果。

#### 测试程序2结果分析：

测试程序2是一个输出1~100的循环程序，根据输出期望和实际输出结果，发现输出正确无误。

#### 测试程序3结果分析：

测试程序3是个读入10个整数，对它们进行冒泡排序（升序）并输出的程序。从输入输出结果分析，程序运行输出无误。

#### 测试程序4结果分析：

测试程序4是一个输入i值，输出斐波那契数列第i项的程序。其中输入12，输出的144确实为第12项，故结果输出正确无误。

#### 测试程序5结果分析：

测试程序5是个求最大公约数的程序，使用的是更相减损术（因为文法没有求余数功能，辗转相除法实现较为麻烦），测试结果中，输入21、14，成功求出最大公约数7。另外自己测试多组数据也成功求出最大公约数，故结果正确无误。

#### 错误程序1结果分析：

错误程序1的错误是最后的main函数没有结束的右大括号，从报错输出可以看出检测出位置和错误信息，与实际错误相符合。

#### 错误程序2结果分析：

该错误程序出错在第九行，缺少了结束变量定义的符号。报错给出了正确的行号和错误信息。多余的错误信息是由于连锁报错导致的，不能算作错误处理有误。

#### 错误程序3结果分析：

该错误程序出错在第三行，整形常量定义位置，没有在等号右边给出常量值。报错给出了正确的行号和错误信息。多余的错误信息是由于连锁报错导致的，不能算作错误处理有误。

#### 错误程序4结果分析：

该错误程序出错在第十一行，重复定义了char类型变量test\_char。报错给出了正确的行号和错误信息。多余的错误信息是由于连锁报错导致的，不能算作错误处理有误。

#### 错误程序5结果分析：

该错误程序出错在第五十四行，if语句的条件使用了char变量，故给出需要整形表达式的报错。报错给出了正确的行号和错误信息。多余的错误信息是由于连锁报错导致的，不能算作错误处理有误。

#### 结果分析：

上述结果中，测试程序1覆盖了所有语法成分，其余测试程序分别实现了几个有意义的小功能来测试该文法写出的程序的使用情况。另外错误的测试程序样例中的报错均成功，故可以说我们的编译器完成了预期的使用需求（即正确编译出C0扩充文法写出的程序）。

## 五．总结感想

**记一次收获满满又满怀遗憾的课程设计**

本次编译课课程设计，我完成了一个小型编译器，目标是把扩充的C0文法编译生成MIPS汇编代码。整体代码量还是不小的。

本身来说，我做这个编译器的积极性还是很高的，一开始就一直在赶进度，所以早期的作业完成的非常早，也提交的非常早。但是后期其他课程的杂事耽搁了这边的进度，甚至自己的电脑进水被送去维修，这都导致这边的优化部分都没有能够完成，可以说是非常遗憾的了。

我在完成这个课设的过程中收获了不少。其中关于编译的原理、目标、过程的理解更加深化了。为了编写这个程序，我查书、上网查找资料、咨询老师助教、和同学讨论。这些无疑带给我了很大的收获。我不仅了解了很多的知识，也学到了和人沟通的方法、查找问题、解决问题的方法。

在实现编译器的过程中，少不了的就是调试过程。由于整个编译体系的庞大性，调试成为了一个非常复杂的问题。我通过这次课设，熟练了单步执行、逐过程执行的调试方法。另外在生成mips代码的部分，由于一开始生成的mips代码肯定是不能运行的（即有一些bug），我也锻炼了阅读汇编代码的能力。当汇编代码能执行却又执行问题时，我是通过阅读汇编代码并与实际实现方法进行比较，来检查代码生成部分BUG的，可以说是人肉翻译机了，这都增强了我对汇编语言和高级语言的理解、对它们之间转化方式的理解。

除此之外我也在讨论区和各路大佬共同讨论问题，大家齐心讨论给人一种“我不是一个人在前进”的感觉，令人心里“暖洋洋”的。

实际上，这个编译器做到这里大概有4700行左右的代码量，也是我自出生以来（笑）做过的最大的项目了，说实话给人成就感也是很大的。做这样的大项目让我深刻认识到事先设计的重要性，不然最后接口产生变化的话要更改的部分实在是太多了（真的，我试过）。好在大部分的设计沿用了最早的计划，没有导致过多的麻烦。

最后，优化部分没能完成实在是我这次课设最遗憾的部分了。本来对于优化我也和同学、助教进行了探讨，但是最后由于各种机缘巧合没有能够完成这个部分，也就放弃了。

总结一下吧，整个编译课设带给我的收获有以下几点：学会了一个编译器的基本组成部分和原理、查阅资料能力的提升、和人沟通能力的提升、小型软件设计能力的提升（包括事先设计等）、理解了时间安排的重要性（一开始做的不错，后来被别的冲了，真遗憾）。