

**实践课程报告**

**题目： ｃ- -程序编译器的实现**

**课程名称： 编译原理实践**

**专业班级： ACM1401**

**学 号： U201414805**

**姓 名： 王浩波**

**指导教师： 徐丽萍**

**报告日期： 2016年1月12日**

**计算机科学与技术学院**

**目录**

[1选题背景 1](#_Toc22716295)

[1.1任务 1](#_Toc1559740234)

[1.2目标 1](#_Toc766518300)

[1.3 源语言定义 1](#_Toc1642001620)

[2词法分析和语法分析 2](#_Toc2095416666)

[2.1单词文法描述 2](#_Toc1799954259)

[2.2语言文法描述 4](#_Toc816713620)

[2.3 词法分析器的设计 5](#_Toc1821387127)

[2.4 语法分析器设计 7](#_Toc589698599)

[2.5语法分析器实现结果展示 8](#_Toc1028081746)

[3语义分析 10](#_Toc1914310684)

[3.1语义表示方法描述 10](#_Toc1141425006)

[3.2符号表结构定义 11](#_Toc465877075)

[3.3错误类型码定义 13](#_Toc786106602)

[3.4 语义分析实现技术 13](#_Toc401311258)

[3.5语义分析结果展示 15](#_Toc629233288)

[4中间代码生成 17](#_Toc1752456026)

[4.1中间代码格式定义 17](#_Toc1606149357)

[4.2中间代码生成规则定义 18](#_Toc1510416809)

[4.4 中间代码生成结果展示 20](#_Toc1158935846)

[5目标代码生成 22](#_Toc63774146)

[5.1指令集选择 22](#_Toc1271838064)

[5.2寄存器分配算法 23](#_Toc1081754008)

[5.3 目标代码生成算法 23](#_Toc1908597979)

[5.4 目标代码生成结果展示 24](#_Toc100368916)

[5.5目标代码运行结果展示 27](#_Toc54060231)

[6结束语 29](#_Toc1921088)

[6.1 实践课程小结 29](#_Toc1265797506)

[6.2自己的亲身体会 29](#_Toc909840893)

[参考文献 30](#_Toc562237603)

# 1选题背景

1.1任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

1.2目标

本次课程实践目标是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

1.3 源语言定义

本次实验选用的文法是c语言的简单集合c--语言。

# 2词法分析和语法分析

2.1单词文法描述

一．Tokens定义如下：

1.STRUCT struct：struct关键字

2.NOTE1 "\/\/"：单行注释

3.NOTE2 "\/\\*"[^"\\*\/"]\*"\*/"：多行注释

4.FLOAT ([1-9]\*[0-9]{DOT}[0-9]\*(f\*)|(F\*))：浮点型数

5.NUM\_16 0(x|X)(([0-9])|([A-F])|([a-f]))+：16进制整数

6.NUM\_8 0[0-7]+ ：8进制整数

7.NUM\_10 0|([1-9][0-9]\*)：10进制整数

8.RETURN return：函数返回return关键字

9.IF if：if关键字

10.ELSE else：else关键字

11.WHILE while：while关键字

12：FOR for：for关键字

13.TYPE (int)|(float)：类型，分别为整形int以及浮点型float

14.ID [\_a-zA-Z][0-9|\_|a-z|A-Z]\*：标识符，以下划线或者英文字母开头，中间可以包含

数字，英文字母，下划线

1. SEMI \;：分号

16.COMMA \,：逗号

17.ASSIGNOP \=：等号

18：RELOP \>|\<|(\>\=)|(\<\=)|(\=\=)|(\!\=)：比较操作的运算符

19.PLUS " \+ "：加法运算操作符

20.MINUS " \- "：减法运算操作符

21.POS \+：取正运算操作符

22.NEG \-：取负运算操作符

23.STAR \\*：乘法运算操作符

24.DIV \/：除法运算操作符

25.AND \&\&：逻辑与

26.OR \|\|:逻辑或

27.DOT \.：点

28.NOT \!：去非运算

29.LP \(：左小括号

30.RP \)：右小括号

31.LB \[：左中括号

32.RB \]：右中括号

33.LC \{：左大括号

34.RC \}：右大括号

35.BLANK [\t\x0B\f\r\x20]+：空字符的集合

二．注释处理：

1.对于单行注释，则在匹配到“//”后，将此行中输入的所有字符的丢弃，直到读到换行符，

则将列号置1，代码如下：

{NOTE1} { char c=input();while(c!='\n')c=input();yycolumn=1;}

2.对于多行注释，则对于匹配到的“/\*任意字符串\*/”，不进行任何操作，即直接丢弃，代

码如下：

{NOTE2} {}

1. 行号列号：

使用flex内部提供的变量yylineno记录行号，定义变量yycolumn定义列号，当读到一个换行符时，yycolumn置为1，代码如下;  
 int yycolumn=1;

#define YY\_USER\_ACTION \

yylloc.first\_line=yylloc.last\_line=yylineno;\

yylloc.first\_column=yycolumn;\

yylloc.last\_column=yycolumn+yyleng-1;\

yycolumn +=yyleng;

1. 优先级：

由于关键字的组成满足id的匹配规则，所以为了避免关键字被匹配成id，应将id规则放在所有关键字规则的后面进行匹配。

1. 错误处理：

在所有匹配规则的最后，加入对于”.”的处理函数，此即为词法分析过程中的错误处理过程。

2.2语言文法描述

一. Hight-level Definitions

包含c--语言中所有的高成语法（全局变量与函数定义）。Program表示整个程序。ExtDefList表示由多个全局变量，结构体，函数定义语句ExtDef组成的语句序列。

二.Specofoers

这一部分的产生式主要与变量的类型有关，分为type类型（int或者float）以及结

构体类型，其中包括结构体的定义语句

三.Declarators

这一部分主要与变量的定义以及函数的定义有关，其中VarDec表示一个变量名或

者数组名的定义，本次实验实现的编译器中，不支持多维数组的定义与使用。FunDec

表示一个函数名的定义。

四.Statements

这一部分的产生式主要与语句有关。其中CompSt语句表示一个被大括号包括的语

句块，StmtList这表示一个有多条Stmt组成的语句序列，Stmt为一条语句或者一

个CompSt语句块。

五.Local Definitions

这一部分的产生式主要与局部变量的定义有关。DefList表示多条由Def组成的变量

定义序列，Def为一条定义语句，其中可能定义了多个变量名，用COMMA隔开，

变量的类型匹配为Specifier。

六.Expressions

这一部分的产生式主要与表达式有关。包括各种运算表达式，以及函数的调用语句

2.3 词法分析器的设计

一.语法树的定义：

语法树节点结构体的定义如下：

struct mytree{

char\*name;

int linno;

int linno2;

int childnum;

int crtype;

char \*charnum;//数值的字符串

char \*vname;//exP的名字

char\*type;

int ca;

struct mytree \*childnode;//孩子

struct mytree \*next;//父节点的子链中的下一个结点

union

{

char\* charval;

int intval;

float fltval;

};

};

其中，name为该节点的名称，childnum为该节点在语法树中的孩子节点的数目，linno为该节点所处的行号，crtype为生成该节点所使用的推导式的编号，charnum：如果该节点的值为一个整型或者浮点型，则该处存放其值的字符串形式，以便于直接输出。Vname：如果该节点为一个Exp节点，如果由id推导而得，则此处用于记录该节点的名称，如果是一个计算得到的节点，则该处记录其值的类型。Type：如果该节点为exp类型则此处记录其值的类型，ca表示是否是一个数组。

1. 词法分析器的实现：

当成功匹配到一个token时，则执行相应的在该token后编写的处理代码，主要功能为生成该token的语法树节点，以备语法分析中生成语法树时使用。并向语法分析程序返回一个该token。实现语句如下（以ID为例）：

{ID} {yylval.a=newnode("ID",0,yylineno);return ID;}

当匹配到一个token为ID类型时，则生成一个语法树节点，存储当前节点的名称以及孩子个数，行号，并向语法分析程序返回一个ID。

如果匹配失败，则匹配为“.”，输出该处的行号以及列号，还有出错的字符。

Token匹配优先级的问题可以通过设置token匹配的顺序实现，例如关键字的优先级应高于ID，则将ID的匹配放于所有的关键字的匹配之后。

2.4 语法分析器设计

语法分析程序每次从词法分析程序中读出一个token，通过判断其与当前待规约的语法规则是否匹配来判断输入的token是否满足语法要求。当输入的token可以规约到一个非终结符时，调用函数struct mytree\* newnode(char\*name ,int num,...)生成该非终结符的对应的节点，并将其子节点加入到该节点的孩子队列上，如果程序正确，则可在完成后，生成程序的语法树，通过函数dlr($$,0)打印输出语法树。

1.移进规约冲突的处理：

通过给语法中的操作符设置结合性，以及优先级来消除文法中的移进

规约冲突：

%right ASSIGNOP

%left AND OR

%left RELOP

%left STAR DIV

%right PLUS MINUS

%left DOT

%left LP RP

%left LB RB

%left LC RC

%left NOT

优先级从上往下依次升高，同意行的操作符具有相同的优先级，%right代表本操作符为

右结合，%left表示本操作符为左结合性。

2.if-else语句的处理：

使用实验指导书中提供的解决方案，在if-else语句的文法中增加一个优先级比else低的

LOW\_THAN\_ELSE：

IF LP Exp RP Stmt IFELSE { $$=newnode("Stmt",6,$1,$2,$3,$4,$5,$6);};

IFELSE:%prec LOWER\_THAN\_ELSE{$$=newnode("IFELSE",0,-1);}

|ELSE Stmt {$$=newnode("IFELSE",2,$1,$2);}

当对if语句进行规约时，如果后面由token“ELSE”，则优先使用第二条推导式移进

token“ELSE”，否则使用第一条生成IFELSE,即没有else语句的if语句，从而解决了

if-else语句文法中存在的引进规约冲突。

3.错误处理

在文法中加入推导处error的推导式，代表当出现错误时，删除还没有规约完的规则并且

需要丢到从error到出错位置之间的所有词法单元，从而保证在后续的语法分析过程中不

会受到该错误的影响。

示例：

Stmt: error SEMI {}

在规约Stmt的推导式中加入如上所示的error，则会将在规约过程中出现错误时，弹出

未规约完的规则，并将从出错位置开始到最近一个分号的所有词法单元丢弃。并由分析器

调用错误处理函数，输出出错位置以及出错原因，输出出错位置的函数如下：

int yyerror(const char\*msg)

{

printf("\nerror inln=%d,col=%d:%s",yylloc.first\_line,yylloc.first\_column,msg);

}

4.语法树输出：

使用先序遍历方式遍历语法树，并输出相应而得节点信息，孩子节点在输出时，相对于其

父亲节点增加两个空格的缩进。

2.5语法分析器实现结果展示

1.语法树输出：

使用测试程序如下：

struct complex

{

float real,image;

};

int main()

{

int i;

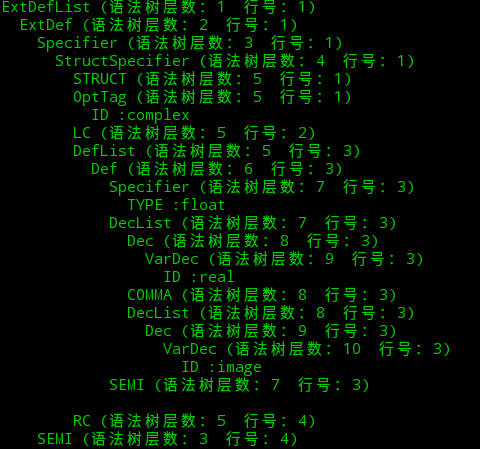
/\*safafada\*/

struct complex y;

y.image=3.5;

}

输出的相应的语法树（部分）如下：



上图部分为结构体complex的定义部分的语法树，由语法树可知结构体complex中包含了两个float类型的变量real以及image。

1. 错误处理：

使用测试程序如下：

int main()

{

int i=1;

int j=~1;

}

其中～为未定义符号，输出报错信息如下：

深度截图20170113153219

# 3语义分析

3.1语义表示方法描述

本次实验采用实验知道书上提供的语义要求，即能够支持结构体的定义和使用，以及各种函数的定义与调用，支持一维数组以及局部变量，全局变量的定义与使用。不支持多维数组的定义与使用，不支持嵌套作用域，不支持指针。

不允许程序中的两个变量重名，不允许变量与结构体，函数名相同，变量类型只支持int，float以及结构体类型。

1.对于定义语句部分：

在此部分的语义分析中，应对每一个出现的标识符，判断其是否在符号表中已经存在，存在，则为符号的重定义，如果不存在，则从specifiers节点中获取其类型信息，将其类型信息与名称共同放入符号表中待查，如果是函数，则还应获取其参数与内部变量信息，共同存入符号表。

2.对于stmt语句部分：

在此部分中，出现的标识符应为已定义的标识符，则在函数表以及变量表中判断其是否已被定义，如果未被定义，则为标识符未定义错误，如果已被定义，如果该标识符为一个函数，还应对其传入的参数的数量，类型判断是否与变量表中存储的信息相同。

3.对于exp表达式语句：

对于表达式语句，应当在语法树的节点中存入当前exp节点的类型：int，float，struct，如果该表达式中存在错误或者规约到该表达式的子表达式类型为“error”，则应将该表达式的类型修改为“error”,以便将错误的表达式从代码中剔除，实现错误处理的功能，并避免重复报错。

3.2符号表结构定义

符号表使用链表结构，分为：全局变量表，全局数组表，结构表，函数表，其中结构表中，有包含变量成员表，数组成员表，而函数表中每一个函数节点中，同样保存则该函数的局部变量表，局部数组表，以及该函数的形参表。

1.全局变量节点：

struct varnode {

char\* name;//变量名

char\* type;//变量类型

int intval;//整型变量的值

float floval;//浮点型变量的值

struct varnode \*next;//指向下一表项的指针

}

其中name指向变脸名称的字符串，type表示变量类型，intval存储整型变量的值，floval存储浮点型变量的值，next指针指向链表中下一个节点。

2.数组节点：

struct arrnode{

char \*name ;//数组名；

char \*type;//数组类型；

struct arrnode\*next;//链表的下一个表项

}

Name为数组的名称，type为数组类型

3.结构体节点：

struct strnode{

char \*name;//结构体名

struct varnode \*vhead;//变量成员的链表

struct arrnode \*ahead;//数组成员的链表

struct strnode \*next;

}

name存储结构体名，vhead表示结构体域中的变量链表，ahead为结构体域中的数组链表，next，为结构体链表中指向下一个结构体节点的指针。

4.函数节点：

struct funnode{

char\*name;//函数名

char\*retype;//返回值类型

int cannum;//参数数目

struct varnode \*vhead;//变量成员的链表

struct arrnode \*ahead;//数组成员的链表

struct cannode \*head;//参数类型表的头指针

struct funnode \*next;//函数表的相下一个节点

}

Name存储函数名，retype存储函数返回值的类型，为int，flaot或者void，cannum为函数的形参个数，vhead为函数中的局部变量链表的头指针，ahead为函数中局部数组的头指针，head为函数形参表的头指针。

5.函数的参数表：

struct cannode{

char\*type;//参数类型

char\*name;//参数名称

int ca;//是否是数组

int intval;

float floval;

struct cannode \*next;//参数表中的下一个参数

}

Type为形参类型，name为参数名称，ca为1表示该参数为1维数组。

3.3错误类型码定义

错误类型的定义与实验书相同，实现了实验书上要求的全部17个错误类型：

|  |  |  |
| --- | --- | --- |
| Type1：变量未定义 | Type2：函数未定义 | Type3：变量重定义 |
| Type4：函数名重定义 | Type5：赋值操作类型不匹配 | Type6：等号坐边不是左值表达式 |
| Type7：操作数类型不匹配 | Type8：返回值类型错误 | Type9：函数实参形参不匹配 |
| Type10：对非数组进行数组访问 | Type11：调用非函数变量 | Type12：数组下标不为整数 |
| Type13：对非结构体使用”.” | Type14：访问未定义的域 | Type15：结构体域重定义 |
| Type16：结构体名重定义 | Type17：结构体未定义 |  |

3.4 语义分析实现技术

1.对于变量定义语句：

由于本次实验的语义分析是在生成语法树的过程中完成的，所以在生成一个变量节点时无法判断该变量是一个全局变量，或者是函数的局部变量，或者是一个结构体中的域。所以通过设置一个缓冲链表来解决这一问题：当生成一个新的变量节点时，将该节点插入缓冲链表中，如果规约过程中产生了一个函数节点或者结构体节点，这将该缓冲链表中的内容全部存入函数或者结构体的变量表中，然后将该缓冲链表清空。如果规约过程中产生了一个全局变量的定义节点，则将缓冲链表中的所有节点加入到全局变量表中，清空该链表。

2.对于函数定义语句：

函数定义语句如下：

Specifier FunDec CompSt

此时三个节点均已生成，从specifier节点中获得函数的返回值类型，从FunDec节点中获得函数名，如果名称重定义，则根据相应的错误类型号报错，否则从fundec中获取函数的形参，将其加入形参表，当归约到当前节点时，可知缓冲链表中的变量与数组为该函数的局部变量，将其加入函数的局部变量与数组表中，完成函数的定义。

3.对于结构体定义：

结构体定义语句如下：

STRUCT OptTag LC DefList RC

由OptTag节点中获得结构体名称，要求名称不能重定义，当规约到当前节点时说明缓冲链表中的变量与数组为结构体内的域，将其加入到结构体的内部变量数组链表中。

2.对于变量的使用：

首先判断该变量所在的函数的局部变量表中是否存在同名项，如果不存在，则在全局变量表中查找是否存在同名项，如果都不存在，则报错，如果存在，则将该id生

成的exp节点的类型赋为该局部变量在符号表中的类型，用于后续表达式求值过程中，类型是否匹配。

3.对于函数的调用：

函数调用语句如下：

Exp：ID LP RP|ID LP Args RP

首先判断函数表中，当前函数名是否存在，如果不存在，这判断其他表中是否存在同名项，如果存在则报错函数调用操作符的错用，否则报错为函数未定义。如果函数名在函数表中存在，则进一步判断实参与形参是否匹配，如果匹配，则将生成的Exp节点的类型赋为函数返回值的类型，以进行后续表达式求值的类型匹配。

4：对于结构体类型域的使用:

首先判断该变量是否为结构体变量，然后判断，该结构体中是否存在当前要使用的域。

5：对于表达式计算（Exp）过程中类型的匹配：

对于每一个Exp节点的生成，如果其运算符操作不匹配，或者存在子节点的类型为error则将该exp类型赋为error，否则，根据表达式求值的规则，为该Exp节点的类型信息赋值。

3.5语义分析结果展示

1.符号表生成：

使用的测试程序：

struct test{

int i;

int j[4];};

int x;

int z[10];

struct test y;

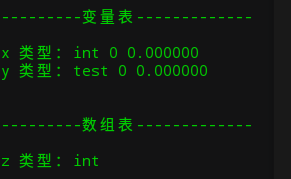
int main(){

int m;

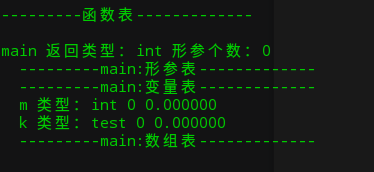
struct test k;

return 0;}

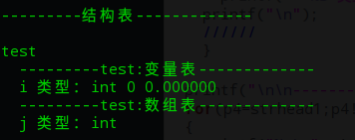
输出的符号表信息如下图所示：



全局变量表中保存了全局变量x，y以及数组z



函数main的返回类型为int，形参为空，局部变量由int类型的m，struct test类型的k。



结构体test符号表中的表项的信息如上。

1. 错误测试：

测试程序：

struct test

{

int i;

int j[4];

};

int main()

{

int m;

float m;

struct test k;

k(1,2);

return 0;

}

其中k不是函数，main函数中的局部变量m重定义，输出的错误信息如下：

深度截图20170113200725

错误类型分别为类型3和类型11。

# 4中间代码生成

## 4.1中间代码格式定义

1.中间代码的格式：

中间代码的格式使用实验指导书中提供的代码格式，并使用IR Simulator进行代码正确性的测试；

2.中间代码的存储形式：

程序所有的中间代码使用链表存储，链表的每一个节点代表一个中间代码语句，存储中间代码的结构体irnode定义如下：

struct irnode

{

int no;

int num;

char\*target;

char\*op;

char\*arg1;

char\*arg2;

struct irnode\*next;

}

其中no表示中间代码的序号，num表示中间代码的长度，包括操作符在内，每个操作数或者操作符的长度视为1，target中存储目的操作数的名字，op存储操作符的名字，arg1表示第一操作数，arg2表示第二操作数，next为下一句中间代码的节点的指针。

对于长度为4的中间代码，输出格式为target ：= arg1 op arg2

对于长度小于4的中间代码，输出格式为 target arg1 op arg2，arg1，arg2，target均可以为空字符串。

## 4.2中间代码生成规则定义

中间代码的生成规则采用实验指导书中提供的翻译模式实现，增加的翻译规则有：乘除表达式的生成规则（与加减类似），数组元素访问的生成规则（根据首元素的地址，计算要访问的数组元素的偏移量，从而进行访问）,以及对语句块CompSt进行处理的规则.

## 4.3 中间代码生成过程

一.对于exp语句：

调用void translate\_Exp(struct mytree\*Exp,char\*place)函数进行处理，其中的Exp为语法树中的Exp节点的指针，place为该Exp语句的求值结果应存的中间变量的名称，该中间变量的名称由调用translate\_Exp函数的函数使用new\_temp（）函数生成。

1.对于函数调用语句：

如果时对参数为空的函数进行调用，则首先判断是否时调用READ函数，如果是，则按照READ函数调用的方式生成中间代码，否则，生成中间代码：place：=CALL 函数名，实现函数的调用。

如果是对有参数的函数进行调用，首先判断是不是对write函数进行调用，如果是，则生成调用write函数的中间代码，如果不是，则调用translate\_Args（）函数，获取传入个各个参数，将其存储在链表arghead1中，对于得到的链表，遍历其每一个表项，生成中间代码：ARG args；然后生成调用函数的中间代码:CALL 函数名。

2.对于表达式求值语句：

对于不同的操作符，根据表达式求值的规则，将其以四元式的格式生成相应的中间代码，如果生成当前exp语句的子节点中仍有Exp节点，则递归调用translate\_Exp函数对该Exp进行处理，生成一个新的中间变量，作为函数调用时的place参数。

3.对于bool表达式语句：

默认使用bool表达式的语句与进行条件跳转的语句，调用translate\_Cond函数生成相应的跳转语句.

二.对于stmt语句:

1.如果时表达式语句,则调用translate\_exp进行处理

2.如果是语句块,则调用函数translate\_CompSt()进行处理;

3.如果是返回语句,则创建一个新的中间变量t1,作为plcae调用translate\_exp函数,产生中间代码RETURN t1

4.如果是一个if语句,则新建标签label\_true,label\_false,并在为真跳转的语句前,插如标签LABEL label\_true,在为假跳转的目标语句前创建标签LABEL label\_false,调用translate\_cond函数,生成条件判断的中间代码语句,并生成相应的goto语句,实现条件跳转.

5.如果是一个while语句,实现方式与ifelse类似.

三.对于函数:

1.在函数中间代码的起始位置,插入标号FUNCTION 函数名

2.如果函数为有参函数,则调用translate\_VarList获取调用函数传入的参数.

3.调用translate\_CompSt函数实现对函数体中代码的翻译.

1. 对于函数参数的处理(translate\_VarList函数):

对于函数参数表中的每一个参数,倒序生成中间代码:PARAM 形参名,将调用函数传入的传入取出,与形参匹配.

1. 对于条件表达式的处理(translate\_Cond函数):

如果操作符的左右操作数为exp则生成新的中间变量t1,t2分别作为place调用tanslate\_exp函数,生成对左右两个表达式进行求值的中间代码.新建新的中间代码t3,生成中间代码:t3:=t1 op t2;然后生成中间代码:IF t3 GOTO label\_true GOTO label\_false,其中的label\_true与label\_false由translate\_Stmt生成,并作为参数传入translate\_Cond函数.

最后调用irprint函数将生成的中间代码逐个输出.

## 4.4 中间代码生成结果展示

示例程序():

int fact(int n)

{

int j;

if (n == 1)

return n;

j=fact(n - 1);

return j \* n;

}

int main()

{

int m, result, i;

m = read();

if (m > 1)

result = fact(m);

else

result = 1;

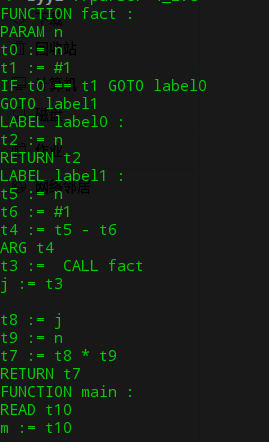
write(result);

return 0;

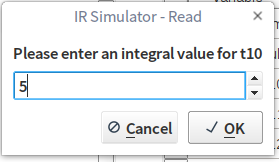
}

函数功能为输入一个数n,递归计算,1\*2\*3\*4\*...\*n的值并输出

生成的(部分)中间代码如图:



在IR simulator中运行的结果如图:



输入值5,结算结果:



1\*2\*3\*4\*5=120,中间代码能够正确实现原程序的功能.

# 5目标代码生成

## 5.1指令集选择

生成目标代码过程中使用的指令如下:

1. 算术运算指令:

加:add s1,s2,s3: s1=s2+s3;

Addi s1,s2,#i: s1=s2+i;

减:sub s1,s2,s3: s1=s2-s3;

乘:mul s1,s2,s3: s1=s2\*s3;

除:div s2,s3 mflo s1: s1=s2/s3;

1. 赋值指令:

move s1,s2: s1=s2;

3.内存操作指令:

读内存指令:lw s1,off(s2): s1=\*(s2+off);

写内存指令:sw s1,off(s2): \*(s2+off)=s1;

4.立即数指令:

li s1,#i: s1=i;

5.无条件跳转指令:

jr $ra: 跳转到$ra存的地址

jal name: 跳转到标号name处,并保存返回地址.

j label: 无条件跳转到标号label;

1. 条件跳转指令:

beq s1,s2,label: s1==s2时跳转到标签label

bne s1,s2,label: s1!=s2时跳转到label

bgt s1,s2,label: s1>s2时跳转到label

blt s1,s2,label: s1<s2时跳转到label

bge s1,s2,label: s1>=s2时跳转到label

ble s1,s2,label: s1<=s2时跳转到label

## 5.2寄存器分配算法

共有32个寄存器,编号分别为$0-$31,其中用$8-$25号寄存器作为通用寄存器,使用一个数组存储rag[i]是否被分配,根据寄存器中存的内容,给每个寄存器确定优先级,存储中间结果的寄存器优先级最低为1,存变量或者函数参数或者当前表达式的中间结果的寄存器优先级为2,当寄存器由空闲时,优先分配由空闲的寄存器,当没有空闲寄存器时,则优先置换优先级低的寄存器,即优先级为1的寄存器,当不存在优先级为1的寄存器时,再置换优先级为2的寄存器.

## 5.3 目标代码生成算法

遍历第三次实验生成的中间代码,对其中的每条语句进行逐条翻译为目标代码:

1. 对于FUNCTION name 语句:

1.生成标号:main:

2.在函数表中找到该函数,生成addi $sp,$sp,-num指令,为每一个参数以及局部变量在堆栈中分配空间,

1. 对于LABEL label:语句:生成标号label
2. 对于REUTRN t1语句：通过指令addi $sp,$sp,num释放为局部变量分配的堆栈空间,并输出指令move $v0,t1存的寄存器,将t1的值放入寄存器V０中做为返回值．输出指令jr %ra返回到调用处．
3. 对于数值计算语句：翻译为相应的加减乘除指令．
4. 对于PARAM　t1语句：通过指令＂move $a0,t1存储的寄存器＂将t1存入参数寄存器a0（只支持参数为一的函数的调用）.
5. 对于语句ARG t1,通过指令＂lw t1寄存器,%d($sp)＂将参数t1从其存的寄存器中取出，放入在堆栈中为参数t1分配的内存空间中．
6. 对于语句CALL 函数名，使用ｓｗ指令将所有存储局部变量以及函数参数的寄存器的值存入其在内存中的地址保护起来．通过＂addi $sp,$sp,-4”,”sw $ra,0($sp)”两条语句，实现对返回地址寄存器内容的保护，然后将寄存器清空，使用jal指令跳转到要到用的函数位置，返回后，使用＂lw $ra,0($sp)＂，＂addi $sp,$sp,4＂两条指令将保护的返回地址寄存器的值取回．在通过ｌｗ指令将保存在堆栈中的局部变量和函数参数的值重新分配到函数调用前占用的寄存器中．函数挑勇操作完成．
7. 对于ＩＦ语句：根据操作码op的不同，分别翻译未条件跳转指令中的一个从而实现条件跳转功能：beq,bne,bgt,blt,bge,ble
8. 对与read函数与write函数的调用：与普通函数调用类似，省略
9. 对于赋值语句r1:=r2:使用＂move r1使用的寄存器，r2使用的寄存器＂指令实现赋值操作．
10. 对于GOTO label语句：使用跳转指令”j label”实现无条件跳转．

## 5.4 目标代码生成结果展示

示例程序：

int main()

{

int a, b, i, n, c;

a = 0;

b = 1;

n = read();

i = 0;

while (i < n)

{

c = a + b;

write(b);

a = b;

b = c;

i = i + 1;

}

return 0;

}

程序功能：输入一个整数n，输出斐波那契数列的前ｎ项

生成的目标代码如下：

.data

\_prompt: .asciiz "Enter an integer: "

\_ret: .asciiz "\n"

.globl main

.text

//Read函数：

read:

li $v0, 4

la $a0, \_prompt

syscall

li $v0, 5

syscall

jr $ra

//write函数

write:

li $v0, 1

syscall

li $v0, 4

la $a0, \_ret

syscall

move $v0, $0

jr $ra

//main函数

main:

addi $sp, $sp, -100

move $t0,$zero

lw $t1,0($sp)

move $t1,$t0

li $t3,1

move $t2,$t3

lw $t4,4($sp)

move $t4,$t2

sw $t1,0($sp)

sw $t4,4($sp)

addi $sp,$sp,-4

sw $ra,0($sp)

jal read

lw $ra,0($sp)

addi $sp,$sp,4

lw $t4,4($sp)

lw $t1,0($sp)

move $t0,$v0

lw $t2,12($sp)

move $t2,$t0

move $t3,$zero

lw $t5,8($sp)

move $t5,$t3

label0:

move $t6,$t5

move $t7,$t2

blt $t6,$t7,label1

j label2

label1:

move $s0,$t1

move $s1,$t4

add $s2,$s0,$s1

lw $s3,16($sp)

move $s3,$s2

move $s4,$t4

sw $t1,0($sp)

sw $t4,4($sp)

sw $t5,8($sp)

sw $t2,12($sp)

sw $s3,16($sp)

move $a0,$s4

addi $sp,$sp,-4

sw $ra,0($sp)

jal write

lw $ra,0($sp)

addi $sp,$sp,4

lw $s3,16($sp)

lw $t2,12($sp)

lw $t5,8($sp)

lw $t4,4($sp)

lw $t1,0($sp)

move $t0,$t4

move $t1,$t0

move $t3,$s3

move $t4,$t3

move $t6,$t5

li $s0,1

move $t7,$s0

add $s1,$t6,$t7

move $t5,$s1

j label0

label2:

move $s2,$zero

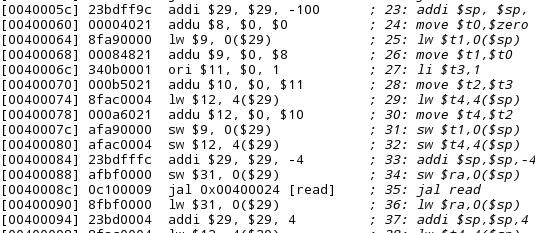
addi $sp, $sp, 100

move $v0,$s2

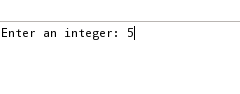
jr $ra

## 5.5目标代码运行结果展示

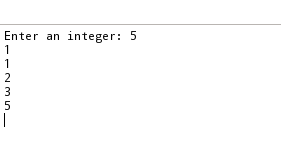
1.目标代码加载入QtSpim:

****

1. 执行生成的目标代码，输入数字５：



输入结果如下：



为斐波那契数列数列的前ｎ项，生成的目标代码成功实现了源程序的功能．

# 6结束语

6.1 实践课程小结

　　在本次实验共八次实验课的过程中，使用ｃ语言，借助flex,bison等工具，从词法分析，语法分析，中间代码生成，再到目标代码生成，一步一步实现了一个十分简陋的ｃ--程序的编译器，该编译器能够实现函数的定义与调用，结构体的定义与使用，能够支持int以及float类型的变量，能够实现while循环语句与if-else条件跳转语句，支持加减乘除运算操作，以及＆＆，｜｜的逻辑运算，只支持一维数组，由于生成目标代码过程中堆栈管理功能不完善，所以不能支持函数的递归调用，对于生成的中间代码与目标代码，没有进行优化，所以导致程序的输出结果很臃肿．本次实验虽然实验指导书上给了比较详细的指导，但是在一些具体的实现细节上，比如堆栈的管理，寄存器的分配上，仍然使用的与实验指导书不同的方法，导致程序实现过程中出现了各种问题，虽然最终这些问题得到了解决，但是仍然在程序中留下了许多后遗症：代码臃肿，部分功能的实现用了很麻烦的方法等．

6.2自己的亲身体会

　　编译原理实验是我认为这学期所有实验中，最难的一个，也花费了最多的时间，虽然最后的成果不是很让人满意，但是通过这一个一步一步走来的过程，仍让我对与课本上学习的编译的过程与方法有了更进一步的了解.在每一个实验的开始阶段，总会有点一头雾水，耽误一些时间，但是结合实验指导书，与课本上的知识，慢慢就会在脑海由形成一个大致的实现思路，有了思路之后，后面的实现过程要稍微容易一点．

**参考文献**

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008