華中科技大学

课 程 实 验 报 告

题目：基于精简 C 语言的 C-MIPS 编译器

课程名称：编译原理实验

专业班级：CS17

学　　号：U2017

姓　　名：

指导教师：徐丽萍

报告日期：2020 年 6 月 12 日

计算机科学与技术学院

目　录

1 概述

编译是由源程序到目标程序的一个转换过程，是当代计算机科学中不可或缺的重要技术。其可细分为词法分析，语法分析，语义分析，中间代码生成，代码优化，目标代码生成六个阶段，且可以通过其他数据结构的辅助加快编译进程或保证编译结果的绝对正确性。

本实验为实现一个小型的 C 语言到 MIPS 汇编语言的编译器，需要完成以上所有步骤并验证测试正确。实验需要选择一个合适的 C 语言子集，能够体现 C 语言基本语法，特有关键字等，同时要能在这个子集的基础上写出具有特色的测试程序。在通过本实验所做编译器编译完成后，通过 MARS 汇编器将汇编代码汇编为汇编语言后测试运行结果需和 C 程序功能预期结果相符。

实验分为四次，依次完成词法和语法分析，静态语义分析、中间代码生成和代码优化，目标代码生成。同时为了体现课程之间的关联性，测试程序最终生成的机器代码将在计算机组成原理课程设计所设计的 CPU 中执行。

词法和语法分析器分别利用现有工具，Flex，Bison 完成，根据输入的文法和语法树构建规则自动生成分析源程序的 C 语言程序，生成抽象语法树并打印之。

语义分析器需要一个 C 语言程序，通过遍历抽象语法树构造符号表并检验上下文是否有语义错误。

中间代码生成器在语义分析器的基础上，在同一遍遍历语法树的基础上，利用当下的符号表生成中间代码。优化器需要一个新的 C 程序，接收中间代码，利用 DAG 算法生成图，并在此基础上进行代码重构和优化，输出优化后的代码。

目标代码生成器需要一个单独的 C 程序，接收中间代码序列在中间代码的基础上，根据中间代码节点的类型，为临时变量，以及存储器变量的值和地址分配寄存器，并生成目标代码。

实验的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，提高学生系统软件研发技术。

2 系统描述

2.1 自定义语言描述

本实验选择的 C 语言子集由五类符号构成：标识符，运算符，定界符，关键字，常量。每类符号的组成见表 2.1。

表 2.1 C 语言子集符号集

符号类型	组成
标识符	符合 C 语言定义的标识符
运算符	+, -, *, /, ++, --, <<, >>, (), !, &&, ||, 关系运算符, =, [], .
关键字	int, float, char ,return, break, continue, struct, if, else, while, for, void
定界符	; / , / () / [] / {}
常量	整型，字符型，浮点型常量

总而言之，自定义语言支持的数据类型有整型，浮点型，字符型，支持数组定义和调用（支持多维数组定义与初始化），支持函数声明，函数定义，函数调用，支持规定的单目及运算符对应的表达式运算，支持循环，条件分支语句等。支持结构体的声明，结构变量的定义，支持结构体与成员变量的调用。不支持指针，取地址等运算符，不支持函数传址以及一些 C 语言隐式写法（如数组名指代的数组首元素地址）。

本实验使用的 C 语言子集语法与 C 语言完全相同，但语义规范与 C 语言有所不同（如函数不支持传入数组作为参数）。同时，词法构成与 C 语言完全一致。

2.2 单词文法与语言文法

2.2.1 单词文法

C 语言中单词分为五类，如表 2.1 所示，单词文法与 C 语言完全相同，对于五类单词，其中运算符，关键字，定界符的词法均为全词匹配，对应的单词是唯一的，因此词法不再赘述，只需要在词法工具中指定对应的单词即可。

2.2.1.1 标识符正规文法

对于标识符，C 语言中的标识符的正规文法表示如下：

$$标识符 \rightarrow LC$$

$$L \rightarrow A\text{-}Z \mid a\text{-}z \mid \_$$

$$C \rightarrow A\text{-}Z \mid a\text{-}z \mid \_ \mid 0\text{-}9 \mid \varepsilon$$

此文法可表示 C 语言中所有的合法标识符。

2.2.1.2 常量正规文法

对于常量，分别为整型，字符型，浮点型常量，为了简化实验内容，突出重点，因此，整型常量均为十进制且无后缀。整型常量正规文法表示如下：

$$整型常量 \rightarrow NZ$$

$$N \rightarrow 0\text{-}9$$

$$Z \rightarrow 0\text{-}9 \mid \varepsilon$$

浮点型常量有三类表示（无后缀），三类表示的示例表示如 3.14，.14，2E+15 正规文法表示如下：

$$浮点常量 \rightarrow M \mid P \mid A$$

$$M \rightarrow N.NZ$$

$$p \rightarrow .NZ$$

$$A \rightarrow Ze+N \mid ZE+N \mid Ze\text{-}N \mid ZE\text{-}N$$

$$N \rightarrow 0\text{-}9$$

$$Z \rightarrow 0\text{-}9 \mid \varepsilon$$

字符常量正规文法表示如下：

$$字符常量 \rightarrow 'C'$$

$$C \rightarrow 任意\ ASCII\ 字符$$

2.2.1.3 非法单词正规文法

由于 C 语言不支持某些非法单词，如 1ac，$ab 等，这些符号没有对应的文法，在源程序中出现错误会报错，因此规定了某些非法标识符，在识别到非法标识符之后会报错，一种非法标识符的正规文法如下：

$$非法标识符 \rightarrow AB$$

$$A \rightarrow CA \mid C$$

$$C \rightarrow 0\text{-}9 \mid @ \mid \# \mid \$$$

$$B \rightarrow DB \mid B$$

$$D \rightarrow A\text{-}Z \mid a\text{-}z \mid \_$$

单词文法相对而言表达比较简单，在实验中运用工具可以很好的表示，并将读取到的单词提供给语法分析程序使用。

2.2.2 语言文法

C 语言的语言文法在高级语言中相对而言是很简单的，精简 C 指令集更对其进行了精简，但是基本结构仍然是 C 语言的外部序列-语句序列-表达式。

2.2.2.1 外部定义序列正规文法

C 程序由外部定义序列构成，外部定义序列由外部变量定义，函数定义，函数声明，结构声明，结构变量定义构成，每一类外部定义有固定的语法格式。C 语言程序正规文法表示如下：

C 程序→外部定义|外部定义序列

外部定义序列→外部定义| ε

外部定义→函数定义|函数声明|外部变量定义|结构声明|结构变量定义

其中外部定义各自对应的语法格式如下，

函数定义正规文法：

函数定义→函数类型(参数列表)复合语句

参数列表→数据类型 标识符|数据类型 标识符 , 参数列表| ε

复合语句→{语句序列}

函数类型→ int | float | char |void

数据类型→ int | float | char

其中的标识符，数据类型等语素成分均来源于词法分析程序。

函数声明正规文法：

函数定义→函数类型(参数列表);

参数列表→数据类型 标识符|数据类型 标识符 , 参数列表| ε

复合语句→{语句序列}

函数类型→ int | float | char |void

数据类型→ int | float | char

外部变量定义：

外部变量定义→数据类型 外部变量序列;

外部变量序列→变量|数组|变量,外部变量序列|数组,外部变量序列

变量→标识符|标识符=表达式

数组→标识符 维度序列|标识符 维度序列={初始化序列}

维度序列→维度|维度 维度序列

维度→[表达式]

初始化序列→表达式序列|{初始化序列}|{初始化序列},初始化序列

表达式序列→表达式|表达式,表达式序列| ε

数据类型→ int | float | char

结构声明：

结构声明→struct 标识符 {成员变量序列};

成员变量序列→成员变量|成员变量 成员变量序列

成员变量→数据类型 变量序列;

变量序列→标识符|标识符 变量序列

数据类型→ int | float | char

结构变量定义：

结构变量→struct 标识符 结构变量序列;

结构变量序列→标识符|标识符 结构变量序列

外部定义序列组成了精简 C 语言的基本结构，而精简 C 语言的主体仍然是函数体中的语句序列。

2.2.2.2 语句序列正规文法

C 语言程序主体是一条一条的语句，包含表达式语句，返回语句，循环语句等，语句构成语句序列。语句序列正规文法如下：

语句序列→语句|局部变量|语句 语句序列|局部变量 语句序列 ε

语句→表达式语句|返回语句|while 语句|for 语句|if 语句|if-else 语句|break 语句|continue 语句|复合语句

对应语句的具体正规文法如下，

表达式语句：

表达式语句→表达式;

返回语句：

返回语句→return;|return 表达式;

while 语句：

while 语句→while(表达式) 语句

for 语句：

for 语句→for(F;表达式;表达式) 语句

F→局部变量|表达式

if 语句：

if 语句→if(表达式) 语句

if-else 语句：

if 语句→if(表达式) 语句 else 语句

break 语句：

break 语句→break;

continue 语句：

continue 语句→continue;

复合语句：

复合语句→{语句序列}

而局部变量序列正规文法如下：

局部变量定义→数据类型 局部变量序列;

局部变量序列→变量|数组|变量,局部变量序列|数组,局部变量序列

变量→标识符|标识符=表达式

数组→标识符 维度序列|标识符 维度序列={初始化序列}

维度序列→维度|维度 维度序列

维度→[表达式]

初始化序列→表达式序列|{初始化序列}|{初始化序列},初始化序列

表达式序列→表达式|表达式,表达式序列

数据类型→ int | float | char

在语句序列和外部定义序列中都出现有表达式，精简 C 语言中的表达式也尤其重要。

2.2.2.3 表达式正规文法

表达式是对值之间的运算，表达式的基本单元是值，值可以是变量，常量，数组调用，结构成员变量调用，函数调用等。正规文法如下：

表达式→表达式 双目运算符 表达式

表达式→++表达式

表达式→--表达式

表达式→!表达式

表达式→-表达式

表达式→表达式++

表达式→表达式--

表达式→(表达式)

表达式→标识符(表达式序列)

表达式→标识符 维度序列

表达式→标识符.标识符

表达式→标识符

表达式→整型常量

表达式→浮点型常量

表达式→字符型常量

双目运算符→ + | - | * | / | % | << | >> | = | += | -= | *= | /= | %= | <<= | >>= | && | || | == | != | > | >= | < | <=

表达式序列→表达式|表达式,表达式序列| ε

表达式的计算是 C 语言执行过程的基本组成内容，一切执行流程都是表达式的计算。

至此，精简 C 语言的文法设计已完成，结合语法，利用工具可以方便地实现词法分析和语法分析，最终生成抽象语法树。

2.3 符号表结构定义

在语义分析和中间代码生成过程中需要用到符号表，符号表的构成及其功能见表 2.2。

表 2.2 符号表结构定义

表项	备注	作用
符号名	符号名称	语义检查时符号比较的依据
别名	临时变量名称	生成中间代码用
符号类型	符号的类型，一个字符，其中函数定义：F，函数原型：D，变量：V，参数：P，数组：A，结构名：S，结构变量名：B，成员变量：N	明确符号的作用，避免同名符号的歧义
层号	指示变量所处的复合语句的层次，外部变量层次为 0	明确变量的位置，同名符号可能层号不同
类型	符号对应的数据类型	符号数据类型，用于类型判断
作用域	全局或局部	指示符号作用域
所属	指示函数形参符号对应的函数名，结构体成员变量对应的结构名	在检查同名符号时与类型结合明确是否属于同一函数或结构

符号表在语义分析过程中填充，根据语法树上下文和节点属性填充符号表。

2.4 错误类型码定义

语法错误是由 LALR(1)分析器自动报出，并指出在出错位置不该出现什么符号，希望出现什么符号，错误种类由文法决定，非常多，因此不单独编码。

在语义分析过程中，会出现和上下文有关的语义错误，如变量类型错误，函数返回值错误等，错误类型有限，因此予以编码以便于快速定位错误，可能出现的语义错误见表 2.3。

表 2.3 语义分析错误列表

编码	错误类型	编码	错误类型
1	函数类型暂时必须为 int	21	while 语句条件表达式语义错误
2	变量初始化类型错误或非常量	22	break 不在循环体内
3	变量重复定义	23	continue 不在循环体内
4	数组长度不为整型常量	24	for 语句内暂不支持数组定义
5	函数参数类型暂时必须为 int	25	for 语句第一条件语义错误
6	函数参数暂时不能为数组	26	for 语句第二条件语义错误
7	函数形参重复	27	for 语句第三条件语义错误

8	函数声明与函数定义类型不符	28	赋值运算左值不为变量
9	函数声明与定义形参数量不同	29	运算符右值语义错误
10	函数声明与定义形参类型不同	30	运算符左值语义错误
11	函数定义重复	31	运算符左右值类型不匹配
12	非 void 函数没有返回值	32	自增运算对象不为变量
13	结构成员变量暂时必须为 int	33	调用为定义或未声明的函数
14	结构成员变量重复	34	函数调用实参序列错误
15	结构重复声明	35	函数实参序列与形参类型不符
16	结构变量重复定义	36	函数实参序列与形参数量不符
17	没有名为该标识符的结构	37	未定义的变量，数组，结构
18	表达式语句中有语义错误	38	数组下标不为整数或调用超过一维
19	return 返回值与函数类型不符	39	数组调用维度与定义不符
20	if 语句条件表达式语义错误	40	结构内不存在该成员变量

在语义分析程序中，根据语法树节点之间的关系检查出的语义错误均会提示报错。

2.5 中间代码结构定义

中间代码采用四元式表示，四元式序列采用链表形式表示，节点类型及其含义见表 2.4。

表 2.4 中间代码节点类型

节点类型	op1	op2	result	释义
FUNC_IR	/	/	函数名	函数体开始标号
PARAM_IR	/	/	参数名	参数名
ASSIGN_IR	操作数	操作数	临时变量	temp=op1=op2，结果赋值临时变量
ARG_IR	/	/	参数名	函数传参，入栈
CALL_IR	/	/	函数名	调用函数
ARR_IR	数组名	下标	结果	result=op1[op2]
STRUCT_IR	结构名	成员	结果	result=op1.op2
RETURN_IR	/	/	临时变量	返回值为 result
IF_IR	标号	/	临时变量	if result==0 goto op1
LABEL_IR	/	/	标号	标号为 result
GOTO_IR	/	/	标号	goto result
EQ_IR	操作数	操作数	标号	if op1==op2 goto result
NE_IR	操作数	操作数	标号	if op1!=op2 goto result

GT_IR	操作数	操作数	标号	if op1>op2 goto result
GE_IR	操作数	操作数	标号	if op1>=op2 goto result
LT_IR	操作数	操作数	标号	if op1<op2 goto result
LE_IR	操作数	操作数	标号	if op1<=op2 goto result
ASSIGN	/	常量	临时变量	result=op2（常量赋值临时变量）
PLUS	操作数	操作数	临时变量	result=op1+op2
MINUS	操作数	操作数	临时变量	result=op1-op2
MUL	操作数	操作数	临时变量	result=op1*op2
DIV	操作数	操作数	临时变量	result=op1/op2
LSHFIT	操作数	操作数	临时变量	result=op1<<op2
RSHFIT	操作数	操作数	临时变量	result=op1>>op2
END_IR	/	/	/	链表结尾标志

中间代码节点采用双向链表，在语义分析和中间代码生成过程中根据语法树节点生成中间代码，语法树节点中如函数定义节点（其子节点为数据类型，参数列表，函数体），遍历到此时会生成一条类型为 FUNC_IR 的中间代码。

2.6 目标代码指令集选择

目标代码指令集是 MIPS 汇编指令集的子集，由于 MIPS 是精简指令集，因此指令条数较少，但是能够达到完备的功能，MIPS 指令集主要是对于寄存器的选择比较繁琐。指令集见表 2.5。

表 2.5 目标代码（MIPS）指令集

指令	功能	指令	功能
sw	存储器写入（字）	jr	寄存器跳转
lw	存储器读出（字）	j	无条件跳转
addi	立即数加	jal	过程调用
subi	立即数减	beq	相等跳转
add	加	bne	不等跳转
sub	减	bgt	大于跳转
sllv	变量左移	bge	大于等于跳转
srlv	变量右移	blt	小于跳转
and	与	ble	小于等于跳转
or	或	syscall	系统调用

中间代码生成目标代码过程中，利用上述指令集，配合寄存器分配算法，可以实现从中间代码到目标代码的完整转换。

3 系统设计与实现

3.1 词法分析器

3.1.1 词法分析器设计

词法分析器采用现有的词法分析软件 Flex，该工具的输入是单词文法，根据识别的结果为单词赋以相应的属性，并通过属性文法将单词识别的结果和属性递交给语法分析器。

需要编辑一个 lex.l 文件，写入设计中声明的精简 C 语言指令集中的五类单词的单词文法。其中标识符，常量的文法已在系统描述中描述，将其转化为符合 Flex 输入格式的正规式，标识符的正规式如图 3.1 所示，常量的正规式如图 3.2 所示，部分未定义标识符正规式如图 3.3 所示。

```
id    [A-Za-z_][A-Za-z0-9_]*
```

图 3.1 标识符正规式

```
int     [0-9]+
float   ([0-9]*\.[0-9]+)|([0-9]+\.)|([0-9]*[Ee][+-]?[0-9]+)
char   \'.\'
```

图 3.2 常量正规式

```
udefid [0-9@#$]+[A-Za-z_]+
```

图 3.3 部分未定义标识符正规式

运算符，定界符，关键字均采用全词匹配，在 lex.l 中直接使用引号包括即可。

对应的正规式后用“{}”包括识别到该单词需要执行的程序操作。

3.1.2 词法分析器实现

编写 lex.l 文件，在语法分析器实现后，在 parser.y 中定义语法的终结符的标识，终结符即为词法分析器中识别的单词，在属性文法的程序中返回对应单词（终结符）的标识，并向控制台输出识别的单词的内容和类型即可。

在 lex.l 中按照关键字，标识符，运算符，定界符，常量，非法符号的顺序编写单词文法，按照格式编写完成后，利用 Flex 程序生成 lex.yy.c 文件，即词法分析器，编译完成后即实现了词法分析程序。但词法分析程序需结合语法分析程序使用，因此不能单独执行功能。

3.2 语法分析器

3.2.1 语法分析器设计

语法分析器采用语法分析软件 Bison 实现，将在系统描述中规定的 C 语言子

集中的语言文法，转换为 Bison 支持的文法，并编写属性文法，在语法分析过程中生成抽象语法树节点，并将其连接为抽象语法树。

抽象语法树中每一个节点需要类型，所有非叶节点都是非终结符生成，所有叶节点都是终结符，终结符前文已叙，为词法分析程序所识别的所有单词，除此之外所有的内容均为单词的集合，即为非终结符。非终结符和终结符都要在输入 Bison 的文本中利用%type 和%token 指定，或者对于运算符（终结符）需要%left 或%right 指定结合性。

抽象语法树节点需要记录的内容在四次实验中内容依次增加，图 3.4 展示的是最终完成的完整编译器程序的抽象语法树的节点内容，为四次实验的集成。

```
struct node {

    enum node_kind kind;                //结点类型
    union {
          char type_id[33];          //由标识符生成的叶结点
          int type_int;              //由整常数生成的叶结点
          char type_char;            //由字符常数生成的叶节点
          float type_float;          //由浮点常数生成的叶结点
          };
    struct node *ptr[4];           //子节点
    struct node *parent;           //父节点
    int pos;                        //语法单位所在位置行号
    int type;                         //用以标识表达式结点的类型
    int num;                        //计数器，可以用来统计形参个数
    char Etrue[15],Efalse[15];        //对布尔表达式的翻译时，真假转移目标的标号
    char Snext[15];                 //结点对应语句S执行后的下一条语句位置标号
    struct codenode *code;            //该结点中间代码链表头指针
    int offset;                       //偏移量
    int place;                      //存放（临时）变量在符号表的位置序号
    int width;                      //占数据字节数
    };
```

图 3.4 抽象语法树节点结构体内容

抽象语法树的结构是根据文法而设计的，文法内容参见 2.2.2。

设计一个 mknode()函数，作为属性文法执行的程序，作用就是生成节点，并将父节点与子节点连接起来，由于 Bison 采用 LALR(1)分析法，语法树是自底向上生长的。

3.2.2 语法分析器实现

将系统描述中设计的语言文法转化为 Bison 可接受的文法书写方式，并且规定词法分析器中的单词种类码（终结符），以及语法分析过程中的非终结符，然后对于每一条语法规则，都写出其生成语法树节点的程序，如生成外部定义序列的属性文法表示如图 3.5 所示。

```
/*外部定义序列语法节点为一序列语法节点，其孩子为具体的五类定义*/
ExtDefList: {$$=NULL;}
          | Var ExtDefList {$$=mknode(EXTLIST,$1,$2,NULL,NULL,yylineno);}
          | FuncDec ExtDefList {$$=mknode(EXTLIST,$1,$2,NULL,NULL,yylineno);}
          | Func ExtDefList {$$=mknode(EXTLIST,$1,$2,NULL,NULL,yylineno);}
          | StructDec ExtDefList {$$=mknode(EXTLIST,$1,$2,NULL,NULL,yylineno);}
          | StructDef ExtDefList {$$=mknode(EXTLIST,$1,$2,NULL,NULL,yylineno);}
          ;
```

图 3.5 外部定义序列节点文法

左部为 External Define List（外部定义序列），右部是其组成，花括号内为创

建语法树节点的函数，传入的参数为节点类型，和右部非终结符对应节点的指针。

按照文法设计，通过属性文法，编写最终的输入文件 parser.y，作为 Bison 的输入，软件会生成一个.output 文件，该文件是 LALR(1)分析法的分析表。还会生成 parser.tab.c 和 parser.tab.h 文件，文件接收精简 C 语言源程序，生成抽象语法树并通过相应的辅助函数输出。

Bison 和 Flex 结合使用，组成完整的语法分析程序，其中 Flex 作为词法分析器起到的只是语法分析器辅助的作用，词法分析函数为 yylex()，语法分析函数为 yyparser()。两者之间的关系如图 3.6 所示。

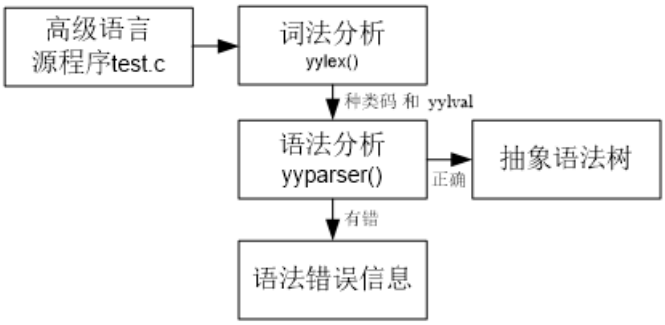

图 3.6 词法分析器和语法分析器关系

生成抽象语法树可以通过 display()函数遍历输出（此为实验一内容），由于自底向上分析方法的根节点最后输出，在根节点生成时的属性文法即可以调用后续函数，在最终的编译器中，语义分析函数便在此处调用。如图 所示。其中 ExtDefList($1)为语义分析程序的顶层函数，$1 为将外部定义序列节点（即语法树根节点）送入语义分析程序进行语法树遍历。

```
program: ExtDefList     { display($1,0);ExtDefList($1);}
       ;
```

图 3.7 程序入口节点，打印语法树和语义分析程序入口

3.3 符号表管理

3.3.1 符号表的实现

根据设计，符号表的数据结构如图 3.8 所示。每个符号对应一个节点，节点为一个结构体，根据的是符号表设计表项完成的数据项，节点类型为 opn，本次符号表采用由每个节点组成的顺序表。

```
struct symbol_table{
    struct symbol symbols[ MAXLENGTH ];
    int index;//index指向的是下一个写入符号表的位置
};

struct opn{
    int kind;                   //标识操作的类型
    int type;                   //标识操作数的类型
    int const_int;      //整常数值，立即数
    char id[33];        //变量或临时变量的别名或标号字符串
    int level;                  //变量的层号，0表示是全局变量，数据保存在静态数据区
    int offset;                  //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用
    //对于函数定义而言，这个相当于栈帧的大小，在result存放
};
```

图 3.8 符号表数据结构

符号表的填充设计了 fillSymbolTable ()函数，函数参数对应于符号表节点成员变量内容，符号表表头指针在每填充一个符号后会前移。

3.3.2 符号表的管理

符号表的填充是语义分析的根本内容，整个 C 程序的核心就是符号之间的关系和运算。

在遍历语法树的过程中，每遇到一个节点需要判断该节点是否需要填充进符号表，如遇到了函数定义节点，那么就要根据函数定义节点语法树的结构，找到函数类型，函数名填充进符号表，并找到函数参数，根据语法树结构，将其类型，名称，偏移量，以及其所属函数的函数名符号在符号表中的位置，依次填充进符号表。如果遇到了变量声明节点，需将其类型，变量名，所处的复合语句层号，偏移量填充入符号表。填充的方法根据不同的符号各有不同，方法比较繁杂，不再赘述。

C 语言的符号分为全局符号和局部符号，全局符号包含函数声明，函数定义，函数形参，结构声明，结构成员变量，全局变量。局部符号只有局部变量。在语义分析过程中，以复合语句为单位，每进入一个复合语句表示当前识别的符号为局部符号，遍历每退出一个复合语句表示当前复合语句内识别的所有符号在符号表中的内容需要被清除，本实验中的实验方法就是指定符号表的表头指针，退出复合语句时，将表头指针指向进入复合语句时的位置即可。

本实验符号表只在语义分析和中间变量生成过程中起重要作用。

3.4 语义检查

3.4.1 语义检查原理

语义检查，主要是检查上下文语义是否有冲突，如数据类型，控制流程是否有冲突，检查的依据基本是符号表内容，以及一些辅助的指示（如 break 不在循环体内）。

3.4.2 语义分析程序设计

语义检查本质上是遍历语法树，在遍历过程中，填充符号表，并检查语义。语义检查横向的流程是覆盖所有的外部定义序列，纵向的流程仍是以函数-语句-表达式三级。而外部定义序列，除了函数定义中的函数体外，其它的均为“声明”，即向符号表内填充内容（填充过程中需要判断重名），而不涉及到上下文语义之间的判断。

对于重名规则，按照 gcc-7.4 编译器标准判断。其中：外部变量，数组，结构变量只能与形参，成员变量重名；函数原型可以与函数定义，形参，成员变量

重；函数声明可以与函数原型，形参，成员变量重名；结构名可以与函数原型，成员变量，形参重名；成员变量可以与本结构成员变量外的一切符号重名；形参可以与本函数外的一切符号重名；局部变量可以与任何层次小于该变量的符号重名。

对于函数体的判断，不仅涉及到局部变量重名的判断，还涉及到符号之间关系的判断，按照函数-语句-表达式三级而言，函数定义时，函数头只针对符号表进行填充，而函数体，即复合语句，需要判断每一条语句的语义是否有误，每一条语句和上下文之间的关系是否有误。由于复合语句的存在，语句的语义检查实际上是一个递归的过程，如 if 语句中还存在真子句也需要调用语句检查的程序，因此需要在递归的过程中需要一些全局变量指示如是否处在循环体中。

对于表达式的语义错误，主要是类型错误的判断，本次实验采用的是强类型控制，即运算符两端类型需要严格匹配。由于运算结果的类型最终取决于操作数，操作数的来源是常量和变量或者调用，而变量，或者调用过程的类型都是来源于先前填充的符号表的（若符号表不存在该符号则为使用未定义符号）。表达式的语义判断也是通过递归实现的，因此在依次向下递归并回溯的过程中总会判断出一个表达式中符号左右的类型，至于其他的语义判断，如赋值运算左值必须为变量，则较为容易就能判断。

在完成针对所有外部定义序列的语义分析（同时也会生成中间代码），若无语义错误，则中间代码生成也完成，进入代码优化环节。

3.5 报错功能

报错功能分为词法，语法，语义报错三类，词法报错是通过词法分析器输入的分析依据（即词法正规式和相关的处理程序），当输入的单词不符合词法规则时，报错，指出某个位置识别到了错误的符号。

语法报错是根据 LALR(1)分析法，当出现分析栈栈顶无归约规则可以执行归约时就进行报错，语法错误种类非常多，报错是词法分析器自动生成的。

而语义报错的个数是有限的，需要自己设计完成，语义分析的报错就是根据表 2.3 中规定的错误类型，在语义分析中，通过语法树上下文和符号表，根据错误类型的定义检查该错误是否出现，若出现则输出之，若未出现则表明程序编写正确。

总之，编译原理中词法语法分析的报错功能保证了语法树的正确性，语义分析的报错功能保证了整个程序没有语义错误，也保证了整个程序没有任何问题。

3.6 中间代码生成

3.6.1 中间代码结构

中间代码结构是一个双向循环链表，节点由中间代码类型和三个操作数节点，还有指示是否为基本块入口的 flag 构成。而操作数节点的组成，需要记录该操作数的数据类型，偏移量，层号（指示全局局部变量），和名称。如图 3.9 所示。

```
struct opn{
    int kind;                   //标识操作的类型
    int type;                   //标识操作数的类型
    int const_int;      //整常数值，立即数
    char id[33];        //变量或临时变量的别名或标号字符串
    int level;                  //变量的层号，0表示是全局变量，数据保存在静态数据区
    int offset;                  //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用
    //对于函数定义而言，这个相当于栈帧的大小，在result存放
};
struct codenode {    //三地址TAC代码结点,采用单链表存放中间语言代码
        int  op;
        char flag;
        struct opn opn1;
        struct opn opn2;
        struct opn result;
        struct codenode  *next,*prior;
};
```

图 3.9 中间代码数据结构

3.6.2 中间代码生成过程

本实验中，中间代码生成和语义分析是同一趟遍历完成的，即对语法树某节点，先检查语义，语义正确后生成中间代码。

中间代码的生成最重要的环节就是临时变量的分配，临时变量很大程度上对应于寄存器，本实验中，对于中间代码的生成有以下几个原则：

1) 按照表 2.4 中的中间代码节点生成中间代码。

2) 一切值都必须通过临时变量进行运算（MIPS 指令集特征，不允许直接操作存储器）

3) 临时变量分为 temp 类临时变量和 addr 类临时变量，temp 类临时变量存放常量值，运算中间结果，addr 类临时变量存放存储器变量值（即变量，数组，结构体成员变量）。在表达式中，针对常量的运算将其赋值给临时变量后再进行运算，针对变量的运算将其赋值给 addr 类临时变量后再运算，赋值给 addr 临时变量的还有该变量对应的偏移地址（偏移地址存放于符号表中）。

4) 对于将常量值赋值给临时变量或将变量值取出的赋值过程使用的节点类型为 ASSIGN，而存储器写回操作用的赋值节点类型为 ASSIGN_IR。

5) 一切运算都有结果，该结果为一个值，对应于一个临时变量，在表达式分析中，右值就以这个临时变量作为操作数。

6) 将负号运算翻译为被减数为 0 的减法运算，将自增运算翻译为变量值+1 的运算，自减运算翻译为变量值-1 的运算，对于后缀自增自减，则将该变量原本的值返回给上层调用的值。

7) 对于关系运算，如 a<b 的值，由于指令集缺乏对应的直接运算指令，将其翻

译为如 if a<b goto L1 else goto L2 L1:result(临时变量)=1 goto END L2:result(临时变量)=0 END:……

8) not 运算的翻译同上，也翻译为等于或不等于 0 跳转为结果为 0 或 1。

9) 函数调用前需要有传参序列的中间代码。

10) break 和 continue 语句的翻译则是无条件跳转到该语句结束标号或条件判断标号。

11) 其它的中间代码翻译遵循四元式翻译的规则。

在生成中间代码过程中，混合运算表达式将被拆分成一条一条二元运算表达式，语法树从叶节点开始依次向上回溯，通过向符号表写入临时变量（运算结果）来向上依次传递运算结果。

3.7 代码优化

生成中间代码后，需要对其进行优化，优化的原则是针对基本快依据 DAG 算法进行优化，在生成中间代码的过程中，已经完成了基本块入口的标识（即中间代码节点中的 flag），在划分基本块时有了基本块入口标识划分非常容易，不多赘述。

在划分基本块后，针对每个基本块，利用 DAG 算法进行优化，DAG 算法优化的过程如下。

对基本块的每一四元式，依次执行：

1、

如果 NODE（B）无定义，则构造一标记为 B 的叶结点并定义 NODE（B）为这个结点；

如果当前四元式是 0 型，则记 NODE（B）的值为 n，转 4。

如果当前四元式是 1 型，则转 2.（1）。

如果当前四元式是 2 型，则：（Ⅰ）如果 NODE（C）无定义，则构造一标记为 C 的叶结点并定义 NODE（C）为这个结点，（Ⅱ）转 2.（2）。

2、

（1）如果 NODE（B）是标记为常数的叶结点，则转 2.（3），否则转 3.（1）。

（2）如果 NODE（B）和 NODE（C）都是标记为常数的叶结点，则转 2.（4），否则转 3.（2）。

（3） 执行 op B（即合并已知量），令得到的新常数为 P。如果 NODE（B）是处理当前四元式时 新构造出来的结点，则删除它。如果 NODE（P）无定义，则构造一用 P 做标记的叶结点 n。置 NODE（P）＝n，转 4.。

（4）执行 B op C(即合并已知量)，令得到的新常数为 P。如果 NODE（B）或 NODE（C）是处理当前四元式时新构造出来的结点，则删除它。如果 NODE

（P）无定义，则构造一用 P 做标记的叶结点 n。置 NODE（P）＝n，转 4。

3、

（1） 检查 DAG 中是否已有一结点，其唯一后继为 NODE（B），且标记为 op（即找公共子表达式）。如果没有，则构造该结点 n，否则就把已有的结点作为它的结点并设该结点为 n，转 4.。

（2）检查 DAG 中是否已有一结点，其左后继为 NODE（B），右后继为 NODE（C），且标记为 op(即找公共子表达式)。如果没有，则构造该结点 n，否则就把已有的结点作为它的结点并设该结点为 n。转 4.。

4、

如果 NODE（A）无定义，则把 A 附加在结点 n 上并令 NODE（A）＝n；否则先把 A 从 NODE（A）结点上的附加标识符集中删除（注意，如果 NODE（A）是叶结点，则其标记 A 不删除），把 A 附加到新结点 n 上并令 NODE（A）＝n。转处理下一四元式。

其中的 NODE 为 DAG 图中的节点，本实验中，DAG 图使用顺序表表示图，由于图中节点度最多为 2，因此，用顺序表表示非常简单。图节点的数据结构如图 3.10 所示。

```
struct DAGnode{

    int no;
    int ifactive;    //标记节点是否被删除
    int ifconst;     //标记节点常量或变量值,0:变量，1：常量，2：运算符
    int varnum;      //该节点依附变量数目，即节点附加值
    char addvar[20][15];  //依附于该节点的变量
    char strvalue[15];      //节点值
    int constvalue;      //节点常量值
    int lchild;
    int rchild;
};
```

图 3.10 图节点数据结构

DAG 优化后，将遍历整个 DAG 图，将所有依附的变量替换为当前节点变量，根据左右子树完成双目运算的中间代码翻译。

本实验中由于没有单目运算符，因此没有一元运算符的四元式，因此 DAG 优化相较于完整的算法更为简单。本实验 DAG 优化重视的是优化算法和思想。

3.8 汇编代码生成

在中间代码生成后，调用目标代码生成程序，将中间代码链表头指针传入，生成目标代码。

生成目标代码最主要的技术在于寄存器分配，本次实验寄存器分配原则见表 3.1。

表 3.1 寄存器分配原则

寄存器号	功能	备注
$0	恒零	/
$2	过程调用返回地址	/
$3,$4	syscall 指令传参	$3 为功能选择，$4 为传入数据
$8-$15	临时变量寄存器	通用寄存器
$16,$17	常量寄存器	涉及常量运算存放
$18	地址计算辅助寄存器	计算数组或结构体偏移地址中间值
$28	全局变量基址寄存器（开始位置 0x200）	全局数据区
$29	栈帧基址寄存器（开始位置 0x800）	栈区
$30	栈帧指针寄存器	栈顶
$31	返回地址（jal 指令）	

其中，通用寄存器的分配需要一个寄存器分配表，表项记录某临时变量与寄存器的对应关系，若临时变量为 addr 类型临时变量（存储器值），还需要取出其地址并存入寄存器中。寄存器分配表是一个顺序表，其节点类型如图 所示。

MIPS 指令中由于对存储器的读写必须通过寄存器，因此对于变量而言均需要将其读入寄存器，因此在寄存器分配的方法上，遵循如下原则：

1) 在任何变量调用之前（即 addr 类数据的 ASSIGN 类型中间代码翻译时），为其分配两个通用寄存器，一个存放值，一个存放地址，并将其寄存器号写入寄存器分配表。

2) 对于双目运算，如 temp=temp1+temp2，由于在取操作数时已经为 temp1 和 temp2 分配了寄存器（也可能有地址），则对于结果 temp，将 temp2 的寄存器转移给 temp，temp1 寄存器释放。

3) 对于关系运算符，在运算后将运算符左右的临时变量对应的寄存器全部释放。

4) 对于赋值运算符，在赋值后将右值按照其对应的地址写回，将左值的寄存器

分配给存储结果值的临时变量。

5) 对于调用的中间代码，将其翻译为 jal。

6) 对于 FUNC_IR 类型中间代码，将其翻译为标号，且执行栈帧操作，即返回地址进栈，栈帧基址进栈，栈顶作为新的栈帧基址，开辟栈帧，保护现场。

7) 对于返回的中间代码，将其返回值传入 2 号寄存器，在执行栈帧相关指令操作后将其翻译为 jr $31，栈帧相关操作为调用过程的逆过程。

8) 对于入栈出栈操作将其翻译为 lw&sw 与 addi 的组合。封装为 push()和 pop()。

9) 对于标号中间代码翻译为标号语句，无条件跳转语句翻译为 j 语句。

按照中间代码序列，一条一条翻译为目标代码，最终得到的目标代码，结束标志就是中间代码序列的结束标志。至此，整个编译过程结束。

4 系统测试与评价

4.1 测试用例

对于系统测试有正确测试用例和报错测试用例，正确测试用例是一个综合的程序，由于最终实现的 MIPS 汇编代码将在自己的 CPU 中运行，需要在测试目标代码中加入部分指示性说明（如全局数据区起始地址赋值，栈区基址赋值），以及 syscall 指令的功能正确性翻译。

正确性测试程序的功能为：依次向屏幕中输出数字 1 到 10，斐波那契数列前五项，以及 2 的 1 到 10 次幂。本测试程序全面测试了基本数据运算，各类循环语句，过程调用以及递归。测试程序参见附录 1。

报错测试用例是一个没有特定功能的程序，其中设置了多处语义错误，检查语义错误的报错情况。设置的错误为典型代表性的错误，并未覆盖全部语义错误。测试程序见附录 2。

4.2 正确性测试

在组成原理课程设计的 CPU 中运行最终生成的测试程序，会在显示控件中依次输出测试预期功能的结果。

执行程序，程序会读取源程序，生成目标程序 target.asm，部分目标代码，如图 4.1 所示。

```
system:
addi $30,$30,-4
sw $31,0($30)
addi $30,$30,-4
sw $29,0($30)
addi $29,$30,0
add $18,$0,$0
add $18,$18,$29
lw $4,8($18)
add $18,$0,$0
add $18,$18,$29
lw $3,12($18)
syscall
addi $30,$29,0
lw $29,0($30)
addi $30,$30,4
lw $31,0($30)
addi $30,$30,4
jr $31
test1:
addi $30,$30,-4
sw $31,0($30)
addi $30,$30,-4
sw $29,0($30)
addi $29,$30,0
addi $30,$30,0
addi $30,$30,-4
sw $8,0($30)
```

图 4.1 生成的部分目标代码

将目标代码导入 MARS 汇编器中，生成目标机器码，将生成的机器码导入 Logisim 中组成原理课程设计的 CPU 的指令存储器中，开启时钟，执行之。输出

如图 4.2，图 4.3，图 4.4 所示。

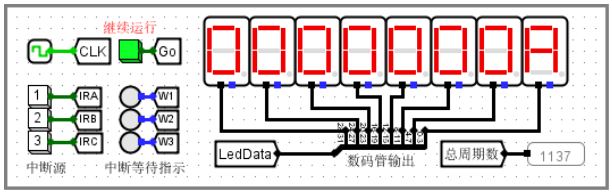

图 4.2 输出从 1 到 10 最终结果

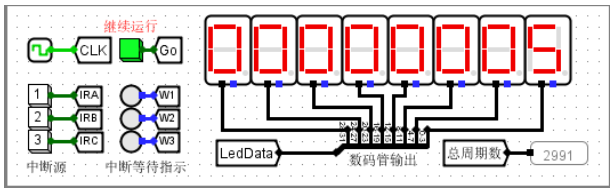

图 4.3 输出斐波那契数列前五项最终结果

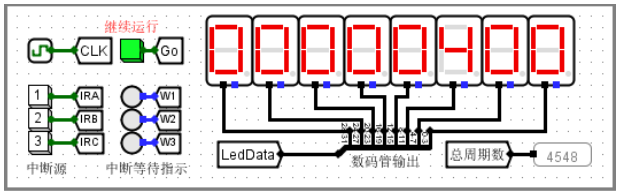

图 4.4 输出 2 的 1 到 10 次幂最终结果

最终执行完停机指令后会停机，至此，整个程序正确性测试成功。

4.3 报错功能测试

将错误程序输入编译器，CMD 中会提示错误，错误有错误类型，行号，以及简单描述，对比附录程序发现，程序中的错误和报错完全对应，但程序中的一个错误可能对应于多个报错。出现语义错误后，便不会进行后续的生成中间代码，已经生成的中间代码也将被舍弃。如图 4.5 所示。

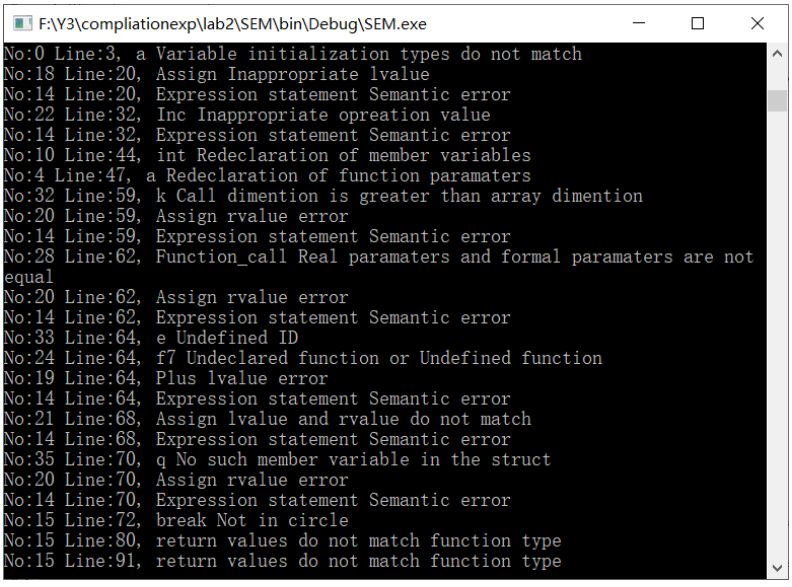

图 4.5 语义报错功能

4.4 系统的优点

系统的主要有点如下：

1) **功能丰富**：系统选取的 C 语言子集支持基本的 C 语言语法，如循环，数组，结构等，能够定义丰富的数据结构，可以通过结构体，多维数组定义多样数据结构，通过这些数据结构和基本的语句指令等编写绝大多数的程序，也可以编写复杂程序，如求递归计算斐波那契数列的程序。
2) **操作清晰**：系统只需要在命令行中通过执行参数输入待编译源程序以及目标程序的路径，即可完成编译，最终自动输出目标汇编程序。
3) **贴近实际**：编译器的词法报错，涵盖了 C 语言的基本规则，包括了 C 语言中未定义标识符；语法报错，严格遵循 C 语言语法规则；语义报错，规定的语义错误数量丰富，规定了 C 语言（不含指针）几乎所有的语义错误。
4) **目标指令精简**：采用了 MIPS 指令集，目标指令集也是 MIPS 指令集的一小部分子集，在编写绝大多数功能程序上绰绰有余，没有采用 MIPS 扩展指令（如 MOV，BGTZ 等，这些指令可以通过基本 MIPS 指令集等价替换得到），使得后续的汇编器工作效率提高
5) **测试方便**：本系统生成的目标指令指令集支持特殊的 syscall 指令的使用，支持在组成原理设计的 CPU 中运行，当然也支持在采用 MIPS 指令集的完整 CPU 中运行。

除此之外还有其它优点，如在中间代码生成过程中，规定了赋值语句和中间结果的临时变量的区别，以确保在各种运算中都能保证存储器值的正确性。在 if，while 语句等条件判断语句中判断条件不仅限于布尔表达式，而是一切表达式，中间代码简化了一些如+=，++，!等运算的表的方法（因为 MIPS 缺乏这样的运算机制，如 MIPS 无自增指令）。

4.5 系统的缺点

系统虽然有点明显，但缺点依然存在，系统缺点如下：

1) **不支持指针**：本次实验选取的精简 C 语言不支持指针，以及衍生的取地址，指针数组，指针函数等语法，缺乏指针的 C 语言是没有灵魂的，但由于指针的语法语义编程极其繁杂，且涉及到对硬件的直接读取，因此为了突出编译原理中各个环节的基本思想和方法，没有选择指针进入本实验的高级语言子集。
2) **不支持某些既定的语法**：由于 MIPS 指令集的限制，以及本实验测试用的组成原理设计的 CPU 得限制，某些语法不支持，如本实验在最终生成中间代码和目标代码时仅支持 int 类型数据，本实验的存储器操作只能操作字而不能操作字节。

3) **不支持弱类型定义**：由于 C 语言可以直接操作硬件，因此不同数据类型的相互赋值可以视为存储器操作，与数据类型无关，因此支持弱类型定义，但由于本实验支持的 MIPS 指令集的限制，以及实验复杂程度的限制，在选取 C 语言子集时规定语义即为只支持强类型定义，即运算符两端操作数必须为同一类型的数据或变量。

除此之外，还有一些其它的不足之处，但是瑕不掩瑜，编译器的主要功能及其正确性都得到了验证，因此，本次实验系统的缺点虽然存在，也有很大的改进空间，但总体是能接受的。

5 实验小结或体会

5.1 实验小结

在本次编译原理实验中，我主要做了以下工作：

1) 设计了精简 C 语言，基于 C 语言语法，功能，选取了其核心部分设计了 C 语言子集。

2) 设计了精简 C 语言的单词文法，并将其转化为了 Flex 软件可识别的正规式，

3) 以及相应属性文法。

4) 设计了精简 C 语言的语言文法，并将其转化为了 Bison 软件可识别的正规式，以及相应的属性文法。

5) 根据语言文法设计了抽象语法树的结构，以及相应构造语法树的函数。

6) 利用 Flex 和 Bison 实现了词法和语法分析，并生成了抽象语法树，并输出检验之正确。

7) 列举了 C 语言中常见的语义错误，并根据实验的实际定义了部分语义独有的错误，并编程遍历语法树检查语义错误实现之。

8) 设计了适合于精简 C 语言与组成原理课程设计的 CPU 适用的中间代码序列，并明确每条中间代码的功能。

9) 编程完成了遍历语法树方式的中间代码生成程序，设计了临时变量，标号，等符号生成程序，成功生成中间代码，验证得功能正确。

10) 编程完成了代码优化程序，依据 DAG 优化原理，编写了 DAG 优化各个步骤的程序，生成了有向无环图。并根据有向无环图生成了优化后的中间代码序列。验证得优化后的中间代码序列与优化前的中间代码序列功能相同且正确。

11) 设计了 MIPS 指令集寄存器分配算法，并根据寄存器分配算法设计了寄存器分配表。选取了目标代码的 MIPS 指令集序列。

12) 编程完成了目标代码生成程序，根据中间代码序列以及寄存器分配算法，实现了目标代码序列的生成。

13) 根据组成原理课程设计的 CPU 对目标代码进行特色改写，并生成了机器码。

14) 将机器码导入到组成原理课程设计的 CPU 中并执行，验证得功能正确，完成了编译器的功能设计，编程实现。

实验完成了上述工作并通过检查，正确地根据编译原理实现了一个简单编译器，完成了编译程序的构造。整个过程逐渐深入，逐渐加深了对课程内容的学习的印象，逐渐提高了自己的编程能力，提高了自己对于编译器执行过程的理解，提高了自己对编译器运行时出现的问题的认识能力，收获颇丰。

5.2 实验体会

本次编译原理实验，我以非常积极非常认真的态度，以非常快的效率，完美完成了所有实验，并在线上检查过程中认真听取了老师的意见，并与老师就有关实验内容的完成过程以及设计思路，调试思路，学习心得等均和老师进行了深入的交流。

我认为，编译原理实验是大学以来设计的最为成功的实验之一，它融合了以往的所学，如操作系统，组成原理，数据结构，并且十分考验计算机编程基础和基本算法的应用，堪比一次课程设计。

完成编译原理实验的过程对我而言是一次巨大的进步，从实验一连工具都搞不懂直到最后能够将源码翻译成目标代码在自己的 CPU 上，这是一个非常麻烦但收获很大的过程，就像举重运动员一样，一开始只能举一点重量，但是坚持每天突破一点，日积月累也会有很大的进步，编译原理实验我一共写了一个半月，每天都有所进步。

对于完成实验，我的经验主要有三点：

1) 要熟悉课程内容，即便不是完全掌握了课程内容，也要至少知道词法，语法，正规式，正规文法，自顶向下和自底向上分析法的思想以及其和语法树的联系，中间代码和目标代码的联系等。

2) 实验中要擅长读取源码，推陈出新，对于老师给的示例，要仔细体会其语言和文法之间的关系，这样就能很快地学会软件的使用方法。

3) 计算机组成原理，操作系统，数据结构，汇编语言程序设计等课程一定要十分熟悉，不然在完成编译原理实验的过程中一定会遇到这样那样的问题。

4) 要多和老师同学交流，对于老师建议要详细采纳，这样至少可以在这门课，这个实验的过程中学到有用的东西，也可以发现自己编程过程中可能存在的小问题。

在完成词法语法分析时，我参考了老师提供的基础代码，由于以往做 C 语言课程设计时，有过实现自顶向下语法分析器和词法分析自动机的经验，因此，学习 Flex 和 Bison 工具时，对于提供的示例，即可被工具识别的正规式及其属性文法，很快就可以上手，尤其是在语法分析程序 parser.y 的实现时，对于 mknode() 函数我很快就理解了其功能，对于语法树节点我设计了更加适合于我的编程风格的数据结构，因此词法和语法分析实验我完成的比较快，而且比较好。个人的体会就是，还是要掌握好课程内容，设计文法时如果掌握好了课程内容在调试过程中出的 bug 会更少，我在编写实验一内容时没有很好地掌握课程中 LALR(1)文法，侥幸有以往的编程经验，才得以顺利完成，在老师的提醒下，我巩固了该文法。

在完成语义分析程序时，由于有以往的课设编程经验，对于遍历语法树的过程，以及识别语义错误的过程我十分得心应手，最后听取了老师关于对错误进行

编号的建议，发现自己设计的程序远远超过了要求的识别错误的数量，自我感觉从这里开始，我对编译原理的理解提升了一个档次。也正是在遍历语法树识别语义错误的过程中，边写我便在考虑如何生成中间代码，以及有关临时变量的应用，总的来说，实验二是我感觉最简单的实验，但是编码较为复杂。

在完成中间代码生成程序时，我在学习了四元式的用法后，在设计程序时即在考虑如何利用中间代码生成目标代码，即在实验三时就在考虑实验四的内容了。在这个考虑的过程中，我发现仅有 temp 类临时变量并不能适配 temp 类所有的变量（或中间结果表示），于是加入了一类新的临时变量 addr，表示存储器变量，里面还存放了偏移地址等信息，在编码实现时，对表达式的递归过程中，我感觉也得心应手。很快就编写完成了程序。

在完成目标代码生成的过程中，由于实验三我在做的时候，已经考虑了实验三中间代码与实验四的关系，因此，实验四完成的非常顺利，非常快，仅仅用了两个课时就完成了，实验四主要是寄存器的分配与临时变量的关系，迅速就完成了。由于我操作系统知识和组成原理知识学得还不错，因此，对于调用过程和返回过程以及内存的分配十分熟悉，因此这些过程的汇编代码我写得比较顺利。而且，我还结合了我的组成原理课设 CPU，添加了系统调用 syscall 指令及其相关函数，参数代码，在检查时已向老师说明，不再赘述。最后实现了程序在自己的 CPU 上运行。在这个过程中，最大的收获便是，我利用编译原理实验的程序，发现了自己设计的 CPU 中的一个问题，即 lw/sw 指令的立即数没有进行符号扩展，只进行了 0 扩展，这个错误的发现与改正让我切实体会到了先后课程之间的联系，让我感受和印象非常深刻。

总的来看，编译原理实验我自我感觉完成的很不错，整个过程也很有收获很有见解，和老师的交流也很愉快。我对课程还是有一些小的建议，已经在检查时和老师交流过，如下：

1) 编译原理实验最好能与课程内容的联系更紧密，由于课程学习内容重点在于词法分析，语法分析，语义分析，属性文法的具体实现，而实验中这些细节被工具隐去，因此体会不深，在考试中也得以体现，由于对于词法语法分析具体程序的不熟悉，导致这些题目做起来还是有些难处。建议实验一改为在使用工具的基础上，再使用 C 语言完成一个简单的语法分析程序和简单的，以符号表示的词法分析程序，以加深对相应课程内容的印象。

2) 最好根据不同的语言子集选择提供相应的报错测试程序，在自己完成时，对于报错的测试需要自己编写程序，没有人擅长刻意书写错误的程序，因此总有一些错误难以注意到，因此对于报错功能的测试建议提供测试样例。

最后，非常感谢老师在实验过程中的指导！

参考文献

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016
[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

附　录

1.正确性测试程序

```
int test1(int n)
{
    return n;
}

int test2(int n)
{
    if(n==1||n==2)
        return 1;
    else
        return test2(n-1)+test2(n-2);
}


int test3(int n)
{
    int i=0;
    int j=1;
    int k=j;
    j<<=n;
    i++;
    return j;
}

void system(int k,int op);

int main()
{
    for(int i=1;i<=10;i++)
        system(test1(i),1);
    int j=1;
    while(j<=5)
```

```
    {
        system(test2(j),1);
        j++;
    }
    j=0;
    while(j<=10)
    {
        system(test3(j),1);
        j++;
    }
    system(0,0);
    return 0;
}
```

2.报错功能测试程序

```
int a='d',b=1;//初始化类型不匹配
char c='d';
float d=2.2;

int f1(int a,int b);



int f2(int c,int d)
{
    c=3;
    if(f1(a,b)==3)
        c=1;
    else
        d=3;

    int e;
    4=a+3;//右值错误
    while(-e--)
    {
```

```
        if(c==5)
            if(d<=3)
                return a=b;

        char f;

        for(int g=1;g>=0;g++)
        {
            g++;
            3++;//右值错误
        }
            break;

    }
    return 1;
}

struct s1{
    int a,b;
    char c,d;
    float e,f;
    int a;//重复声明
};

int f3(int a,float b,char c,int a);//重复声明


struct s1 h;

int j[10];

char k[11][12];

int main()
{
```

```
    h.a=j[1];
    h.c=k[1][2][3];//调用维度过多
    a=b=j[2];

    a=f3(h.a,3.2,'4',5);//函数参数类型不匹配

    e+f7();

    k[h.b][1]='c';

    a=h.c;

    b=h.q;//未定义的成员变量

    break;//不在循环内的 break

    return h.b;
}

int f4()
{
    float a;
    return a;//返回值与类型不符
}

char f5()
{
    int k;
    return 'a';
}

void f6()
{
    return 1;//void 函数有返回值
}
```