

目录

1 选题背景.....	1
1.1 任务.....	1
1.2 目标.....	1
1.3 源语言定义.....	1
2 实验一 词法分析和语法分析.....	2
2.1 单词文法描述.....	2
2.2 语言文法描述.....	3
2.3 词法分析器的设计.....	4
2.4 语法分析器设计.....	4
2.5 语法分析器实现结果展示.....	5
3 语义分析.....	8
3.1 语义表示方法描述.....	8
3.2 符号表结构定义.....	8
3.3 错误类型码定义.....	10
3.4 语义分析实现技术.....	11
3.5 语义分析结果展示.....	12
4 中间代码生成.....	14
4.1 中间代码格式定义.....	14
4.2 中间代码生成规则定义.....	14
4.3 中间代码生成过程.....	17
4.4 代码优化.....	17
4.5 中间代码生成结果展示.....	18
5 目标代码生成.....	19
5.1 指令集选择.....	19
5.2 寄存器分配算法.....	19
5.3 目标代码生成算法.....	20
5.4 目标代码生成结果展示.....	20
5.5 目标代码运行结果展示.....	23
6 结束语.....	24
6.1 实践课程小结.....	24
6.2 自己的亲身体会.....	24
参考文献	25
附件：源代码（见附件文件夹，里面包含了一些测试用例）	26

1 选题背景

1.1 任务

完整实现一个简单编译器，加深课程中关键算法的理解，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。

1.3 源语言定义

选用 C 语言的简单集合 C--语言。

2 实验一 词法分析和语法分析

2.1 单词文法描述

以下描述采用的是 flex 的正则表达式形式:

预先定义一下词法单元:

```
digit [0-9]
hexdigit [0-9a-fA-F]
octdigit [0-7]
hexnum "0x" {hexdigit}+
octnum 0 {octdigit}+
decnum 0 | ([1-9] {digit}*)
FLOAT [0-9]+\.[0-9]*
compareop ["=" "<" ">" "<=" ">=" "<>"]
letter [A-Za-z]
id "_"? {letter} ({digit} | {letter} | _)*
delim [" "]
whitespace {delim}+
```

词法描述如下所示, 其中 return 后面的表示对该词法的定义, 注释后面是具体解释:

```
"/".*\n {printf("COMMENT %s at line %d\n", yytext, yylineno);}
"/*".*"/"\n {printf("COMMENT %s at line %d\n", yytext, yylineno);}
"/*" {BEGIN COMMENT; printf("begin COMMENT at line %d: ", yylineno);}
<COMMENT>\n {printf("\n"); yylineno;}
<COMMENT>"*/" {BEGIN 0; printf("COMMENT end at line %d\n", yylineno);}
<COMMENT>"*/" [ \t]*\n {BEGIN 0; printf("COMMENT end at line %d\n",
yylineno);}
<COMMENT>. {printf("%s", yytext);} //上述是对注释的定义。
\n {} //回车符, 忽略
\t {} //制表符, 忽略
; {return SEMI;} //分号
, {return COMMA;} //逗号
= {return ASSIGNOP;} //赋值号
{compareop} {return RELOP;} //比较符号
"+" {return ADD;} //加号
"-" {return SUB;} //减号
"*" {return MUL;} //乘号
"/" {return DIV;} //除号
"&&" {return AND;} //与
"||" {return OR;} //或
"." {return DOT;} //点
"!" {return NOT;} //非
"(" {return LP;} //左圆括号
```

```

")" {return RP;}//右圆括号
"[" {return LB;}//左方括号
"]" {return RB;}//右方括号
"{" {return LC;}//左花括号
"}" {return RC;}//右花括号
"int" {return TYPE_INT;}//整型定义关键字
"float" {return TYPE_FLOAT;}//浮点数定义关键字
"struct" {return STRUCT;}//结构体定义关键字
"return" {return RETURN;}//返回关键字
"if" {return IF;}//关键字 if
"then" { return THEN;} //关键字 then
"else" {return ELSE;}//关键字 else
"while" {return WHILE;}//关键字 while
{hexnum} {return HEX;}//十六进制数
{octnum} {return OCT;}//八进制数
{decnum} {return DEC;}//十进制数
{FLOAT} {return FLOAT;}//浮点
{id} {return ID;}//标识符
{whitespace} {}//空格
. {}//匹配其余字符

```

2.2 语言文法描述

语言语法规则采用实验书附录部分 C--语言文法。

从 C—语言文法的正则表达式中可以看出，C—语言文法主要由 High-level Definitions, Specifiers, Declarators, Statements, Local Definitions, Expressions 共 6 个部分组成，他们的描述如下：

High-level。这一部分的产生式与 C—语言中所有的高层定义语法有关，即全局变量和函数定义。

Specifiers。这一部分的产生式与类型定义有关 **Specifier** 是类型描述符，他可以是 **int** 或者 **float**，还可以是定义的结构体，也可以在定义结构体的时候初始化结构体变量。

Declarators。这一部分与变量和函数的定义有关。可以是一个或者多个变量，也可以是函数的定义或者声明。

Statements。这一部分的产生式主要与语法有关。包括函数两个花括号之间的语法单元的产生，各种局部变量的声明定义，表达式的产生，**IF** 语句，**IF-ELSE** 语句，**WHILE** 语句的产生。

Local Definition。这一部分的产生式主要与局部变量的定义有关，包括对于局部变量的赋值操作。

Expressions。这一部分的产生式主要与表达式的产生有关，包括普通的基本表达式和实参列表等。

对于表达式，各种运算符之间的优先级与结合性的关系如表 2-1 所示，且表中的优先级为递增。

表 2-1: C—语言文法的优先性与结合性关系表

Right	ASSIGN
Left	OR
Left	AND
Left	RELOP
Left	ADD、SUB
Left	MUL、DIV
Right	NOT
Left	LB RB LC RC LP RP DOT
Nonassoc	ELSE

具体文法与教程后面的描述基本一致，不同的地方是在 if-else 语句，我的文法为

Stmt :

IF LP Exp RP THEN Stmt

| IF LP Exp RP THEN Stmt ELSE Stmt,

在中间加入了一个 THEN 关键字。

2.3 词法分析器的设计

词法分析器的主要任务是将输入文件中的字符流组织成词法单元流，在某些字符不符合程序设计语言词法规范时要有能力报告相应的错误。本实验中采用的方法是利用工具 **GNU Flex** 生成，根据工具要求的输入所编写的词法规范正则式，其理论基础是正则表达式和有限状态自动机。

具体设计分析器时，主要是根据 **flex** 的语法，结合课上学习的正则表达式以及实验指导书附录 A 的 **tokens** 的提示进行设计。设计的比较成功的是实现了注释与多行注释的实现，这里主要说一下多行注释的实现：

当遇到“/*”的时候，进入 **COMMENT** 状态，在这个状态下，如果遇到“*/”退出 **COMMENT** 状态，否则，则认为匹配到的都是注释。这个多行注释利用了 **flex** 自带的一个状态来实现，这部分是通过《**lex** 与 **yacc**》这本书来学习实现的。

2.4 语法分析器设计

语法分析程序的主要任务是读入词法单元流，判断输入程序是否匹配程序设计语言的语法规则，并在规范匹配的情况下构建起输入程序的静态结构。本实验中采用的方法是利用工具 **Bison**，其前身是 **yacc**，其生成的语法分析程序采用了自底向上的 **LALR (1)** 分析技术，编写完语法分析程序之后编译它，会生成一个 C 语言文件，其中 **yyparse ()** 和词法分析程序产生的 **yylex ()** 需要联合使用，在 **main** 函数中编写显示调用，即可实现对文件的词法语法分析。

完成语法分析的同时还要建立语法树，为下一个实验，语义分析打好基础，语法树的节点定义如下图 2-1 所示：

```

typedef struct node{
    int type; //分成终结符和非终结符
    char *name; //非终结符和终结符的名字
    struct node* child; //保存儿子节点
    struct node* bros; //保存兄弟节点
    int line; //保存行号
    union{
        char* idname; //若是id, 则保存id的名字
        int intval; //若是int, 则保存int的值
        float fltval; //保存float的值
    };
    int expType;
    int pExpType; //if it's struct or func ,pType will point to the addr of the type
    int dimen; //表示维度, 为1为1维数组, 其余同理;
    int isleftVal; //is left value
    int place; //where to store the exp or the num val
    int codeLine; //code line
    char code[100];
    int oprandType; //id => 1, num => 0
    Oprand oprand;
    union{
        BoolList * boolList;
        List * nextList;
    };
}Node;

```

图 2-1

上述数据结构可以保存生成语法树时应有的信息，比如是否是非终结符，节点的名字等等。

通过 bison 自下而上的规约过程，通过综合文法，逐步规约生成语法树。同时根据节点里面保存的 level 来确定缩进。如果没有语法或者词法错误，则根据根节点，遍历输出语法树。

2.5 语法分析器实现结果展示

实验书上的例子全部都通过了测试，这里举两个例子作为语法分析器实现的结果。

结果展示：

1) 程序有误时输出错误信息：

输入的代码如图 2-2，报错如图 2-3 和图 2-4：

```

int main()
{
    int a,b;
    a = ;
    b + ();
    return 0;
}

```

图 2-2

```

syntax error Error type B at ; LINE 4.9-4.10: ID : b

```

图 2-3

```
syntax error Error type B at ) LINE 5.10-5.11: SEMI : ;
```

图 2-4

2) 当输入正确的符合语法和文法的代码时，输出语法树，截图如图 2-6，图 2-7（语法树较长，这里给出部分截图，-1 和括弧里面的数是后面测试的时候用到的）。

输入的代码如图 2-5:

```
int main(int dd)
{
    int a = 0, b = 1, i = 0, n;
    n = read();
    while(i < n)
    {
        int c = a + b;
        write(b);
        a = b;
        b = c;
        i = i + 1;
    }
    write(a);
    return 0;
}
```

图 2-5

```
BEGIN TO PRINT TREE:
Program -1 (1)
  ExtDefList -1 (1)
    ExtDef -1 (1)
      Specifier 0 (1)
        TYPE_INT 0
      FuncDec 0 (1)
        ID 3: main
        LP -1
        VarList -1 (1)
          ParamDec 0 (1)
            Specifier 0 (1)
              TYPE_INT 0
            VarDec 0 (1)
              ID 0: dd
          RP -1
        CompSt -1 (2)
          LC -1
          DefList -1 (3)
            Def -1 (3)
              Specifier 0 (3)
                TYPE_INT 0
              Declist 0 (3)
                Declist 0 (3)
                  VarDec 0 (3)
                    ID 0: a
```

图 2-6

```

    ASSIGNOP -1
    Exp 0 (3)
    INT 0 (3)
    DEC 0: 0
COMMA -1
DecList 0 (3)
  DecList 0 (3)
    VarDec 0 (3)
      ID 0: b
      ASSIGNOP -1
      Exp 0 (3)
      INT 0 (3)
      DEC 0: 1
    COMMA -1
    DecList 0 (3)
      DecList 0 (3)
        VarDec 0 (3)
          ID 0: i
          ASSIGNOP -1
          Exp 0 (3)
          INT 0 (3)
          DEC 0: 0
        COMMA -1
        DecList 0 (3)
          Dec 0 (3)
            VarDec 0 (3)
              ID 0: n

```

图 2-7

输出结果与预期一致。

3 语义分析

3.1 语义表示方法描述

语义分析实验目标是根据语法分析和词法分析的结果对输入的代码段进行除了语法还有逻辑上的检查，同时建立符号表，为生成代码做基础。

语义分析有两种方法：

- 1) 根据实验 1 生成的语法树，遍历语法树，生成符号表并检查错误。
 - 2) 利用语义制导的方法，采用 L-翻译模式进行边语法分析，同时语义分析。
- 我这里采用的是第二种方法。这种方法简单有效，同时跟能书上知识有重合。

3.2 符号表结构定义

在这次实验中，我的符号表采用的是链表栈的结构。栈中的每个元素都是一个符号表，栈顶为当前符号表，因此可以实现局部变量。符号表为单向链表的结构，里面的每个符号都保存了符号的定义、名字、类型、维度等。如果该符号是一个复杂变量，比如结构体，函数，则还有一个指针来保存这个符号对应的更加具体的信息。

符号节点的定义如图 3-1：

```
typedef struct _idnode
{
    char * idname;
    int idtype;
    int pType; //if it's struct or func ,pType will point to the addr of the type
    int dimen; //维度，一维数组该数值为1
    int place; //where to store the id
    int size; //size
}IdNode;
```

图 3-1

idname 保存符号名，idtype 保存类型，pType 保存复杂类型的信息的地址(使用时要强制类型转换成对应的指针)，dimen 保存维度，非数组变量为 0 等等。

同时，为了允许结构体的定义，还须定义结构体表，里面保存每一个结构体声明的域。结构体信息节点定义如图 3-2：

```
typedef struct _structDefList
{
    char *name; //save the def name
    int thisDefType;
    int pthisDefType; //if it's struct or func ,pType will point to the addr of the type
    int dimen;
    int offset;
    struct _structDefList * next;
}StructDefList;
```

图 3-2

其中，name 保存了成员的名字，thisDefType 保存的是该成员的类型，pthisDefType 保存了成员的复杂类型信息，dimen 保存成员的维度，offset 保存的是成员的偏移量，next

指向下一个成员。

结构体表定义如图 3-3:

```
typedef struct _structType
{
    char *name; //save the structtypeTag name
    int MemNums; //save the structtype member NO.
    int size; //the struct size
    StructDefList *structDefListHead; //save the structtype membertype
    struct _structType * next;
}StructType;

struct _structTab
{
    StructType * structTagTabHead;
    StructType * structTagTabTail;
}StructTab;
```

图 3-3

其中 name 保存了结构体的名字, MemNums 保存了结构体拥有的成员数, size 保存了结构体所占的内存空间, 单位为字节。structDefListHead 指向第一个成员。next 指向下一个已经声明了的结构体

函数也是符号, 因此也要建立函数的信息结构。参数表定义如图 3-4:

```
typedef struct _argList
{
    int thisArgType;
    int pthisArgType; //if it's struct or func ,pType will point to the addr of the type
    struct _argList * next;
}ArgList;
```

图 3-4

函数的定义如图 3-5:

```
typedef struct _funtype
{
    int DefReturnType;
    int DefPReturnType;
    int ReturnType;
    int codeline; //store code line
    int ArgNums;
    ArgList *argListHead;
}FunType;
```

图 3-5

DefReturnType 保存的是定义的时候函数的返回类型, DefPReturnType 是考虑到了可能返回类型是复杂类型, ReturnType 保存的是实际返回的类型, ArgNums 保存的是参数的个数, argListHead 指向参数表的头。

解决了复杂类型后, 就可以构建符号表了。符号表的定义如图 3-6, 先定义了一个链表, 然后根据链表定义了一个栈, 并声明了一个全局变量来作为进行语义分析的符号表。

```

typedef struct _symnode
{
    IdNode * node;
    struct _symnode * next;
}SymNode;

typedef struct symtab
{
    SymNode* Head;
    SymNode* Tail;
}_SymTab;

typedef struct _symtabstacknode
{
    int level;
    _SymTab * SymTab;
    int num;
    struct _symtabstacknode * next;
}SymTabStackNode;

struct _symstack
{
    SymTabStackNode * Top;
}SymStack;

```

图 3-6

3.3 错误类型码定义

实现了实验指导书上 17 个基础错误类型的检查和报错功能，具体的错误类型的提示如下：

1. 错误类型 1：变量在使用时未经定义。
错误信息提示：“error type 1 at line %d : %s is not defined”。
2. 错误类型 2：函数在调用时未经定义。
错误信息提示：“error type 2 at line %d : function %s is not defined”。
3. 错误类型 3：变量出现重复定义或变量与前面定义过的结构体名字重复。
错误信息提示：“error type 3 at line %d: repeat declaration var %s”。
4. 错误类型 4：函数出现重复定义（即同样的函数名被多次定义）。
错误信息提示：“error type 4 at line %d: repeat declaration function %s”。
5. 错误类型 5：赋值号两边的表达式类型不匹配。
错误信息提示：“error type 5 at line %d: expression type conflict when ASSIGNOP”。
6. 错误类型 6：赋值号左边出现一个只有右值表达式。
错误信息提示：“error type 6 at line %d : can't ASSIGN a left value”。

7. 错误类型 7: 操作数类型不匹配或操作数类型与操作符不匹配 (例如整型变量与数组变量相加减, 或数组 (或结构体) 变量与数组 (或结构体) 结构体变量相加减)。错误信息提示: “error type 7 at line %d: this operation only allow INT or FLOAT expression” 或者 “error type 7 at line %d: expression type conflict”。
8. 错误类型 8: return 语句的返回类型与函数定义的返回类型不匹配。错误信息提示: “error type 8 at line %d : return type conflict(no return type)”。
9. 错误类型 9: 函数调用时实参与形参的数目或类型不匹配。错误信息提示: “error type 9 at line %d : function args type not match.”。
10. 错误类型 10: 对非数组型变量使用 “[...]” (数组访问) 操作符。错误信息提示: “error type 10 at line %d : Exp is not an array”。
11. 错误类型 11: 对普通变量使用 “(…)” 或 “()” (函数调用) 操作符。错误信息提示: “error type 11 at line %d : var %s is not a function”。
12. 错误类型 12: 数组访问操作符 “[...]” 中出现非整数 (例如 a[1.5])。错误信息提示: “error type 12 at line %d : index is not an integer”。
13. 错误类型 13: 对非结构体型变量使用 “.” 操作符。错误信息提示: “error type 13 at line %d : Exp is not an struct”。
14. 错误类型 14: 访问结构体中未定义过的域。错误信息提示: “error type 14 at line %d : struct doesn't have this field”。
15. 错误类型 15: 结构体中域名重复定义 (指同一结构体中), 或在定义时对域进行初始化 (例如 struct A{int a = 0;})。归类到错误类型 3。
16. 错误类型 16: 结构体的名字与前面定义过的结构体或变量的名字重复。错误信息提示: “error type 16 at line %d: repeating declaration of struct %s”。
17. 错误类型 17: 直接使用未定义过的结构体来定义变量。错误信息提示: “error type 17 at line %d: undefined struct %s\n”。

3.4 语义分析实现技术

语义分析采用 L 翻译模式, 其核心思想给每一个终结符或非终结符赋予一个或多个属性值, 利用这些属性值在规约时进行传播, 同时进行类型的检查, 建立符号表。

在进行语义分析和类型检查的过程中要不断对符号表进行查询和修改操作。如果碰到错误, 统一记录下来, 最后输出, 直到完成整棵语法树的检索。

3.5 语义分析结果展示

这次实验的亮点是，实现了多级符号表，支持局部变量，逻辑与现在主流语言一样，局部变量覆盖，从栈顶往下查询符号，查询到最近的符号为止，并且，可以支持结构嵌套等多种复杂的数据类型结构。但比较可惜的是，因为时间不够，采用的是单向链表而不是哈希表作为符号表的结构。

由于课本上的例子比较单一，这里给出的测试样例是自己设计的，里面包含了局部变量，结构嵌套等比较复杂的情况。

输入的代码如图 3-7，图 3-8：

```
int a[10];
struct g
{
    int b;
    int c;
    float f;
}d;
struct gg
{
    int b;
    int c;
    float f;
    struct g a;
}ss;

int a;
struct g q(int asd, float basd)
{
    int a;
    int b;
    float c;
    s();
    ss.a.f;
    ss.b.a;
    a+b;
    a*b;
    a*c;
    d + f;
    asd + sg;
    q(a,c);
    return a;
}
```

图 3-7

```
int p()
{
    struct g xixi;
    //int b[2][2];
    a[10];
    b[10][10];
    b[1][1][1];
    xixi.f;
    //a[1.0];
    q(1, 2);
    //a[1] = q(1, 2.5);
    q(1, d.f).f;
    xixi = q(1, d.f);
    a.s;
    d.f;
    a[2] = 2.5;
    a[2] = 2;
}

int p(int qwe, float qwed)
{
    qwe;
    qwed;
    {
        int qwe;
        int we;
    }
    qwe;
    we;
    2 = qwe;
}
```

图 3-8

编译器的报错如图 3-9，可以发现该报错的地方实现了报错，同时也支持结构嵌套的访问：

```
there is 25 error:

error type 3 at line 17: repeat declaration var a
error type 2 at line 25 : function s is not defined
error type 13 at line 27 : Exp is not an struct
error type 7 at line 30: expression type conflict
error type 1 at line 31 : f is not defined
error at line 31: previous expression type conflict
error type 1 at line 32 : sg is not defined
error at line 32: previous expression type conflict
error type 8 at line 34 : return type conflict
error type 1 at line 42 : b is not defined
error type 10 at line 42 : Exp is not an array
error type 10 at line 42 : Exp is not an array
error type 1 at line 43 : b is not defined
error type 10 at line 43 : Exp is not an array
error type 10 at line 43 : Exp is not an array
error type 10 at line 43 : Exp is not an array
error type 9 at line 46 : function type not match
error type 9 at line 46 : codeLine is 0.
error type 13 at line 50 : Exp is not an struct
error type 5 at line 52: expression type conflict when ASSIGNOP
error type 8 at line 54 : return type conflict(no return type)
error type 4 at line 56: repeat declaration function p
error type 1 at line 65 : we is not defined
error type 6 at line 66 : can't ASSIGN a left value
error type 8 at line 67 : return type conflict(no return type)
```

图 3-9

4 中间代码生成

4.1 中间代码格式定义

中间代码生成，就是在语法树和符号表的帮助下，将输入的高级语言源代码，翻译成接近目标代码的中间代码。

因为 LALR(1)文法是从左到右规约的文法，与代码生成的顺序一致，因此可以直接用语义制导的方法来逐步生成中间代码，所以这次实验，采取的生成方法还是 L-翻译模式，生成中间代码。。实验中，用到了很多课本上学到的东西，比如拉链回填，综合属性，继承属性等等。

中间代码格式为四元式。具体格式如表 4-1

表 4-1 中间代码语句与功能对照表

中间代码语句	描述
TABLE x :	定义标号 x
func f :	定义函数 f
x := y	赋值操作
x := y + z	加法操作
x := y - z	减法操作
x := y * z	乘法操作
x := y / z	除法操作
x := &y	取 y 的地址赋给 x
x := *y	取以 y 值为内存单元的内容赋给 x
*x := y	取 y 值赋给以 x 值为地址的内存单元
goto x	无条件跳转至标号 x
IF x [op] y goto LABEL z	如果 x 与 y 满足[op]关系则跳转至标号 z
RETURN x	退出当前函数并返回 x 值
PUSH x	传实参 x
return to x	调用函数，并将其返回值赋给 x
Param x	函数参数声明

4.2 中间代码生成规则定义

主要是采用一一对应的方式进行中间代码的生成

表 4-2 基本表达式的翻译模式

makeTAC (opname, dst_operand, src1_operand, src2_operand) = case Exp of	
INT	\$\$->operand.operandType = INT; \$\$->operand.operandVal = \$1->child->intval;
ID	\$\$->operand.operandType = 1; \$\$->operand.operandVal = temp->place;
Exp1 ASSIGNOP Exp2 (Exp1 -> ID)	root->place = left->place; root->operand.operandType =

	left->operand.operandType; root->operand.operandVal=left->operand.operandVal; makeTAC("ASSIGNOP",&left->operand,&right->operand, NULL);
Exp1 ADD SUB MUL DIV Exp2	root->operand.operandVal = newtemp(); root->operand.operandType = 1; makeTAC(op->name,&root->operand,&left->operand, &right->operand);
NOT SUB Exp1	root->operand.operandVal = newtemp(); root->operand.operandType = 1; makeTAC(op->name,&root->operand,&left->operand, NULL);
Exp1 RELOP Exp2	root->operand.operandVal = 0; root->operand.operandType = 0; root->codeLine = codecnt + 1; makeTACList(root->boolList->trueList,TACcnt); makeTACList(root->boolList->>falseList, TACcnt + 1); makeTAC(op->name, NULL, &left->operand, &right->operand);//cmp jmp makeTAC("jmp", NULL, NULL, NULL);//directed jmp makeList(root->boolList->trueList, codecnt); makeList(root->boolList->>falseList, codecnt + 1);
Exp1 AND Exp2	backPatch(\$1->boolList->trueList, \$3->place); merge(\$\$->boolList->>falseList,\$1->boolList->>falseList, \$4->boolList->>falseList); \$\$->boolList->trueList=\$4->boolList->trueList
Exp1 OR Exp2	backPatch(\$1->boolList->>falseList, \$3->place); merge(\$\$->boolList->trueList,\$1->boolList->trueList,\$4->boolList->trueList); \$\$->boolList->>falseList=\$4->boolList->>falseList;
ID LP Args RP ID LP RP	root->operand.operandVal = newtemp(); root->operand.operandType = 1; sprintf(temp, "CALL %s", ID->idname); makeTAC(temp,&root->operand,NULL,NULL);
Exp DOT ID	root->operand.operandVal = newtemp(); root->operand.operandType = 1; root->codeLine = codecnt + 1; Operand tempOperand;

	tempOprand.oprandType = 0; tempOprand.oprandVal = temp->offset; makeTAC("LEA",&root->oprand,&Struct->oprand, &tempOprand);
Exp LB Exp RB	root->oprand.oprandVal = newtemp(); root->oprand.oprandType = 1; root->codeLine = codecnt + 1; makeTAC("LEA",&root->oprand,&Exp->oprand, &index->oprand);

表 4-3 语句的翻译模式

makeTAC (opname, dst_oprand, src1_oprand, src2_oprand) = case Stmt of	
Exp SEMI	makeTAC("Stmt", NULL, NULL, NULL);
CompSt	makeTAC("LC", NULL, NULL, NULL); makeTAC("RC", NULL, NULL, NULL);
RETURN Exp SEMI	makeTAC("RETURN",&(\$2->oprand), NULL, NULL);
IF LP Exp RP THEN M Stmt	backPatch(\$3->boolList->>trueList, \$6->place); backPatch(\$3->boolList->>falseList,LABLEcnt) Oprand temp; temp.oprandType = 0; temp.oprandVal = LABLEcnt; makeTAC("LABEL", &temp, NULL, NULL);
IF LP Exp RP THEN M Stmt N ELSE M Stmt	ackPatch(\$8->nextList, LABLEcnt); Oprand temp; temp.oprandType = 0; temp.oprandVal = LABLEcnt; makeTAC("LABEL", &temp, NULL, NULL); backPatch(\$3->boolList->>trueList, \$6->place); backPatch(\$3->boolList->>falseList,\$10->place)
WHILE LP M Exp RP M Stmt N	backPatch(\$8->nextList, \$3->place); backPatch(\$4->boolList->>trueList, \$6->place); backPatch(\$4->boolList->>falseList,LABLEcnt) Oprand temp; temp.oprandType = 0; temp.oprandVal = LABLEcnt; makeTAC("LABEL", &temp, NULL, NULL);

表 4-4 Dec 翻译模式

makeTAC (opname, dst_oprand, src1_oprand, src2_oprand) = case Dec of	
VarDec	makeTAC("New Var", &temp1, NULL,NULL);
VarDec ASSIGNOP Exp	temp1.oprandType = 1; temp1.oprandVal = \$1->place; makeTAC("New Var", &temp1, NULL,NULL); Oprand temp2;

	temp2.oprandType = \$3->oprand.oprandType; temp2.oprandVal = \$3->oprand.oprandVal; makeTAC("MOVE", &temp1, &temp2, NULL);
--	--

表 4-5 函数参数的翻译模式

makeTAC (opname, dst_oprand, src1_oprand, src2_oprand) = case Args of	
Exp	makeTAC("PUSH", &(\$1->oprand), NULL, NULL);
Exp COMMA Args1	makeTAC("PUSH", &(\$1->oprand), NULL, NULL);

4.3 中间代码生成过程

最初实验的时候，没有考虑到目标代码的生成，因此中间代码保存的方式比较简单，用的是字符串保存。在做目标代码的时候，发现这样子生成目标代码很繁琐，因此定义了一个四元式来保存中间代码，同时扩展了语法树的域方便中间代码的生成。具体定义如图 4-1，图 4-2：

```
typedef struct _oprand
{
    int oprandType;//1=>var , 0 => const
    int oprandIsAddr;//1 => is addr 0 => not addr
    int oprandVal;
}Oprand;
```

图 4-1

```
struct
{
    char * opName;
    Oprand dst;
    Oprand src1;
    Oprand src2;
}TAC[100];
```

图 4-2

根据上述中间代码的数据结构，结合 4.2 节所述的中间代码生成规则，在生成语法树，进行语义分析的同时，生成中间代码，保存起来，最后将中间代码遍历输出。

4.4 代码优化

- 代码优化主要做的有：
- 1、立即数操作不新建临时 t 来保存，而是直接向上传递。
 - 2、拉链回填，减少了 jmp 和 label 的次数。

4.5 中间代码生成结果展示

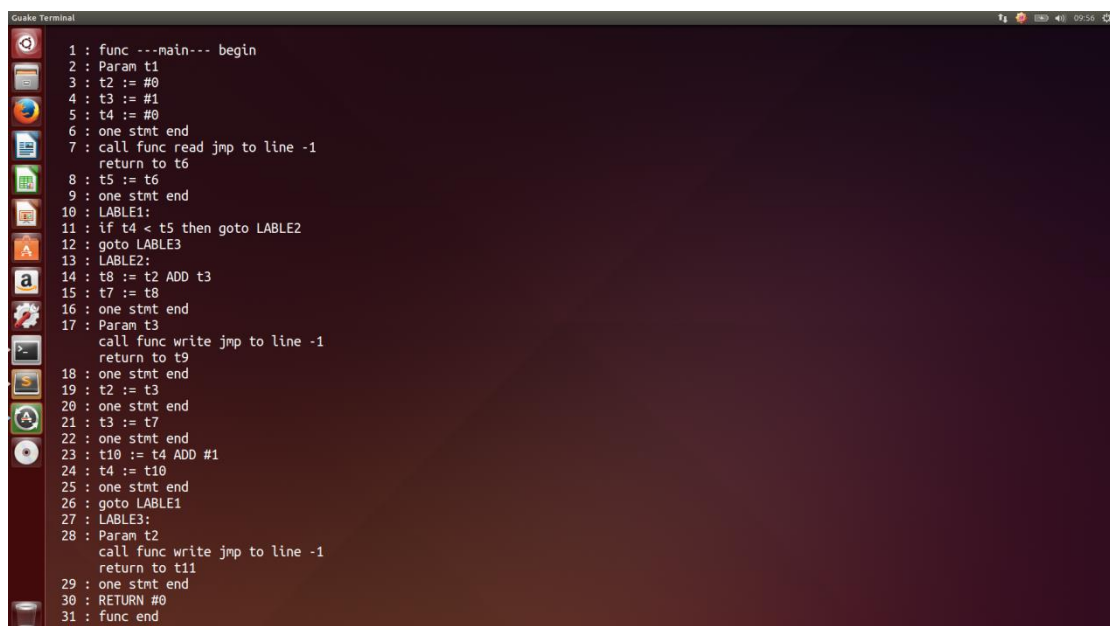
完成了实验三要求的全部必做部分，以下为部分测试效果截图。

样例输入斐波拉契数列如图 4-3:

```
int main(int dd)
{
    int a = 0, b = 1, i = 0, n;
    n = read();
    while(i < n)
    {
        int c = a + b;
        write(b);
        a = b;
        b = c;
        i = i + 1;
    }
    write(a);
    return 0;
}
```

图 4-3

输出为图 4-4:



```
1 : func ---main--- begin
2 : Param t1
3 : t2 := #0
4 : t3 := #1
5 : t4 := #0
6 : one stnt end
7 : call func read jmp to line -1
   return to t6
8 : t5 := t6
9 : one stnt end
10 : LABEL1:
11 : if t4 < t5 then goto LABEL2
12 : goto LABEL3
13 : LABEL2:
14 : t8 := t2 ADD t3
15 : t7 := t8
16 : one stnt end
17 : Param t3
   call func write jmp to line -1
   return to t9
18 : one stnt end
19 : t2 := t3
20 : one stnt end
21 : t3 := t7
22 : one stnt end
23 : t10 := t4 ADD #1
24 : t4 := t10
25 : one stnt end
26 : goto LABEL1
27 : LABEL3:
28 : Param t2
   call func write jmp to line -1
   return to t11
29 : one stnt end
30 : RETURN #0
31 : func end
```

图 4-4

结果与预想的一致。

5 目标代码生成

5.1 指令集选择

本次实验指令集选择了 MIPS32，为 RISC 指令集。本次实验中指令选择方式是逐条将中间代码对应到目标代码上，与实验指导书上的基本一致。如表 5.1 所示。

表 5.1 中间代码与 MIPS32 指令对应

中间代码	MIPS32 指令
LABEL x :	x:
x := #k:	li reg(x), k
x := y	move reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	subu reg(x), reg(y), k
x := y - z	subu reg(x), reg(y), reg(z)
x := y * z	mul reg(x), reg(y), reg(z)
x := y / z	div reg(x), reg(y), reg(z)
x := *y	lw reg(x), 0(reg(y))
*x = y	sw reg(y), 0(reg(x))
GOTO x	无条件跳转至标号 x
x := CALL f	jal f move reg(x), \$v0
RETURN x	move \$v0, reg(x) jr \$ra
IF x == y GOTO z	beq reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z
IF x > y GOTO z	bgt reg(x), reg(y), z
IF x < y GOTO z	blt reg(x), reg(y), z
IF x >= y GOTO z	bge reg(x), reg(y), z
IF x <= y GOTO z	ble reg(x), reg(y), z

reg(x)表示变量 x 所分配的寄存器。

5.2 寄存器分配算法

寄存器分配的算法类似于 gcc 对 x86 指令集的分配方法，中间变量放入 \$t0~\$t7 号寄存器中，局部变量放在栈中，函数的形参存储在 \$a0~\$a3 号寄存器中。这样的好处是，局部变量不用担心没有分配空间，使用了 t 寄存器也能避免使用的寄存器过少，但是还是存在要频繁往内存读写。

因此，要创建函数的活动记录，才能正确的使用栈，并对栈恢复。

AR 的定义如图 5-1:

```
struct
{
    struct
    {
        int type;//temp, or var or param
        int addr;//if var or param, show the dstCode reg num or addr
    }typeTab[100];
    int varNum;
    int paramNum;
    int tempNum;
    int argNum;
    int isPush;
    int compstVarNumStack[100];
    int stackTop;
}funcAc;
```

图 5-1

AR 中保存了变量个数，参数个数，使用的 temp 寄存器的个数，由于是支持局部变量覆盖，所以还有各个代码块的变量个数栈，来维护进入或者退出代码块时栈的结构。

5.3 目标代码生成算法

主要方法是逐句翻译，需要注意的是，在函数调用的时候，要保护现场，将已经使用的寄存器压栈，函数调用完成之后，根据函数的活动记录，去恢复栈帧。

5.4 目标代码生成结果展示

输入的样例程序为:

```
int fact(int n)
{
    if(n == 1) then
        return n;
    else
        return (n * fact(n - 1));
}

int main()
{
    int m, result;
    m = read();
    if(m >= 1) then
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}
```

图 5-2

目标代码生成的结果为图 5-3:

```

.data
_prompt: .asciiz "Enter an integer: "
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0, 4
    la $a0, _prompt
    syscall
    li $v0, 5
    syscall
    jr $ra

write:
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, _ret
    syscall
    move $v0, $0
    jr $ra

```

图 5-3

```

fact:
    addi $sp, $sp, -4
    sw $a0, 0($sp)
    lw $t0, 0($sp)
    li $t1, 1
    beq $t0, $t1, label1
    j label2
label1:
    lw $t0, 0($sp)
    addi $sp, $sp, 4
    move $v0, $t0
    jr $ra
    j label3
label2:
    lw $t2, 0($sp)
    sub $t1, $t2, 1
    move $a0, $t1
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal fact
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    addi $sp, $sp, 4
    lw $t4, 0($sp)
    mul $t3, $t4, $v0
    addi $sp, $sp, 4
    move $v0, $t3
    jr $ra

```

图 5-3 (续)

```

main:
    addi $sp, $sp, -4
    addi $sp, $sp, -4
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal read
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    sw $v0, 4($sp)
    lw $t0, 4($sp)
    li $t1, 1
    bge $t0, $t1, label4
    j label5
label4:
    lw $t0, 4($sp)
    move $a0, $t0
    addi $sp, $sp, -4
    sw $t0, 0($sp)
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal fact
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    lw $t0, 0($sp)
    addi $sp, $sp, 4
    sw $v0, 0($sp)
    j label6
label5:
    li $t0, 1
    sw $t0, 0($sp)

```

图 5-3（续）

```

label6:
    lw $t0, 0($sp)
    move $a0, $t0
    addi $sp, $sp, -4
    sw $t0, 0($sp)
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal write
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    lw $t0, 0($sp)
    addi $sp, $sp, 4
    li $t0, 0
    addi $sp, $sp, 8
    move $v0, $t0
    jr $ra

```

图 5-3（续）

5.5 目标代码运行结果展示

对 5.4 节中样例程序生成的目标代码，使用 SPIM Simulator 进行运行。代码目标是通过递归实现对阶乘求值，运行结果如下图所示。

```
piggy@piggy-VirtualBox:~/compiler$ ./parser test3 2.s  
  
piggy@piggy-VirtualBox:~/compiler$ spim -file 2.s  
SPIM Version 8.0 of January 8, 2010  
Copyright 1990-2010, James R. Larus.  
All Rights Reserved.  
See the file README for a full copyright notice.  
Loaded: /usr/lib/spim/exceptions.s  
Enter an integer: 7  
5040
```

运行结果与预期完全一致

6 结束语

6.1 实践课程小结

本次编译实验共分四个部分，首先是词法分析和语法分析，接着是语义分析，然后进行中间代码生成，最后实现目标代码生成。四次实验下来，实现了一个基于 `c--` 语言的简单简陋的编译器。

6.2 自己的亲身体会

总的来说，这次实验难度较大，感触也很多，收获也很多。

编译原理实验可以说是目前花费时间最多，写的最困难的一次实验。

首先遇到的困难就是，指导书内容的不完善，课上对 `flex` 和 `bison` 的使用方法讲述的不多，在自己查阅了大量资料，借阅相关书籍，才克服了这一个困难。

然后是语义分析，到了语义分析阶段，大家的写法千奇百怪，有的是遍历多次语法树，而我使用的是语义制导的方法，但是语义制导的方法会导致代码逻辑混乱，格式不优雅，前后端没有分开等多种开发的困难，同时，还涉及到要对大量的数据结构进行维护，还好相关知识在课上掌握的比较熟练。

在生成中间代码的时候，起初没有考虑目标代码，导致在做第四次实验的时候，还要对代码有大部分的修改浪费了很多时间。

最后生成目标代码的时候，只能说感谢计算机系统基础的实验，让我对栈和 `gcc` 对栈的使用有比较深刻的理解，这部分仿照 `x86` 的结构，最后正确实现了。

虽然遇到了很多的困难，但收获也是很大的。先不谈最后会写编译器有什么用，这次实验是自己第一次独立的开发了将近 10 个 `.c` 文件，接近 3000 多行代码的开发，不仅巩固了 `c` 语言的知识，对数据结构的掌握也进一步加深了。而当最后代码能运行，甚至在自己组成原理实验上，实现的 `cpu` 运行的时候，还是非常开心的。

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

附件：源代码（见附件文件夹，里面包含了一些测试用例）