

编译系统设计

C-编译器设计报告

成员

纪友升 3090100703

赵冰骞 3090103420

时间：2012 年 6 月 18 日

目录

1. 实验概述	3
2. 系统需求	3
3. 总体设计	4
3.1 系统设计以及模块划分	4
3.2 任务分工	5
4. 模块设计	5
4.1 词法分析	5
4.2 语法分析以及语法树建立	5
4.3 符号表建立	9
4.4 语义分析及类型检查	11
4.5 代码生成	11
5. 测试及结果	12
5.1 编译阶段测试	12
5.2 生成代码以及最终测试	16
6. 说明以及体会	26

1. 实验概述

在《编译原理》以及《编译系统设计》两门课的学习中，我们掌握了计算机程序编译的基本原理以及技术。一个程序的编译一般来说，要经过词法分析、语法分析，语义分析、源代码优化、目标代码生成，目标代码优化等过程。而在这次实验中，我们将应用这些知识，实现一个简单的 C-编译器。

2. 系统需求

实验要求实现一个 C-编译器。C-是 C 语言的一个子集，但是包含了 C 语言的基本特性。在 C-中，变量类型只有 int 型，但是使用数组，而且函数参数也可以是数组类型。这实际上就是指针类型。C-实现了函数，这也确定了运行时环境是基于栈的。在函数以及函数中的表达式块中，可以定义变量，但是而且这些变量属于不同的作用域，因此在编译器以及运行时环境的实现中也要考虑这些情况。全面了解一个语言的特性，有助于我们的编译器的实现。

在这次的 C-编译器设计中，我们将目标语言选定为 NASM 汇编语言，之后再利用 NASM 汇编器生成可执行文件运行、测试。

C-的基本语法如下：

1. $program \rightarrow declaration-list$
2. $declaration-list \rightarrow declaration-list\ declaration \mid declaration$
3. $declaration \rightarrow var-declaration \mid fun-declaration$
4. $var-declaration \rightarrow type-specifier\ ID\ ; \mid type-specifier\ ID\ [NUM]\ ;$
5. $type-specifier \rightarrow int \mid void$
6. $fun-declaration \rightarrow type-specifier\ ID\ (params)\ compound-stmt$
7. $params \rightarrow param-list \mid void$
8. $param-list \rightarrow param-list\ ,\ param \mid parm$
9. $param \rightarrow type-specifier\ ID \mid type-specifier\ ID\ [\]$
10. $compound-stmt \rightarrow \{ local-declarations\ statement-list \}$
11. $local-declarations \rightarrow local-declarations\ var-declaration \mid empty$
12. $statement-list \rightarrow statement-list\ statement \mid empty$
13. $statement \rightarrow expression-stmt \mid compound-stmt \mid selection-stmt \mid iteration-stmt \mid return-stmt$
14. $expression-stmt \rightarrow expression\ ;\ ;$
15. $selection-stmt \rightarrow if(expression)\ statement \mid if(expression)\ statement\ else\ statement$
16. $iteration-stmt \rightarrow while(expression)\ statement$
17. $return-stmt \rightarrow return\ ; \mid return\ expression\ ;$
18. $expression \rightarrow var = expression \mid simple-expression$
19. $var \rightarrow ID \mid ID\ [expression]$
20. $simple-expression \rightarrow additive-expression\ relop\ additive-expression \mid additive-expression$
21. $relop \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=$
22. $additive-expression \rightarrow additive-expression\ addop\ term \mid term$
23. $addop \rightarrow + \mid -$
24. $term \rightarrow term\ mulop\ factor \mid factor$

```
25. mulop  $\rightarrow$  * | /  
26. factor  $\rightarrow$  ( expression ) / var / call / NUM  
27. call  $\rightarrow$  ID ( args )  
28. args  $\rightarrow$  arg-list / empty  
29. arg-list  $\rightarrow$  arg-list , expresion / expression
```

3. 总体设计

3.1 系统设计及模块划分

按照一般程序编译的过程，我们将这次实验分为以下几个阶段：词法分析、语法分析及语法树的建立、符号表的建立、类型等语义的检查、代码生成。其中，每个阶段的任务为：

词法分析：读入源程序文件中的字符序列，将这些字符划分为单词（Token）序列。这个过程可以看成是一个模式识别的过程，可以利用自动状态机来实现，也可以利用现有的工具（Lex 等）自动生成。

语法分析以及语法树的建立：这一阶段读入由词法分析器产生的 Token，根据 C-的语法规则分析源程序的语法结构。在分析源程序语法结构的同时，利用树等数据结构，将源程序表示成方便后续处理步骤的结构。这里主要使用了语法树这种表示方法。

符号表的建立：在语义分析以及代码生成等阶段，我们经常会需要查询一些信息，例如变量或者函数是否已经定义，变量或者函数的类型、函数的参数个数以及类型、变量在堆栈中的地址等信息。由于符号表会经常被访问（插入，查询等），因此需要设计一种比较高效的数据结构方便使用。另外，我们将符号表的建立作为一个相对独立的阶段，以方便信息的统一。这一阶段要为各个作用域建立符号表，将变量、函数等标识符插入符号表，同时更新一部分类型等信息。而变量在堆栈中的地址这些运行时相关的信息需要在代码生成阶段再更新。

类型等语义的检查：这一阶段对程序进行语义检查，包括检查使用的变量、函数是否已经定义，函数、变量类型是否匹配，调用函数的参数个数、类型是否匹配等。这是编译检查的最后一个阶段，这一阶段要将程序中在编译阶段能够检查出的错误都检查出来。因为后面的代码生成阶段将对没有编译错误的源程序生成目标代码。

代码生成：经过前面的一系列处理，现在已经得到无错的程序语法树及其符号表。这一阶段的任务就是根据这些已有信息生成可以在目标机器上运行的目标代码。我们选择的目标环境是 x86 的 linux，在其上可以运行的汇编有 nasm 汇编等。具体任务是：根据语法树生成 nasm 汇编，使用 nasm 汇编器生成 elf 可执行连接格式，最后使用 GNU 的 ld 连接器整合目标文件，生成可执行文件。

最后，要说明的是，编译检查的各个阶段，要实现错误恢复功能。这也是一个编译器的重要特性。编译器程序中有一个全局变量 Error 指示当前是否已经出现错误。如果出现错误，则输出错误信息，将 Error 变量置为 1，然后继续该阶段的执行。阶段执行完成之后将根据 Error

的值确定是否进入下一阶段还是终止运行。

在本实验中，将区分 bool 类型和 int 类型变量。

3.2 任务分工

姓名	任务
纪友升	词法语法分析、符号表建立以及语义分析
赵冰蹇	代码生成

4. 模块设计

4.1 词法分析

这一阶段利用了词法分析器生成工具 Lex 自动生成,要识别的 Token 有以下这些:

C-语言保留字(关键字),包括 int, void, if, else, return, while.

运算符,包括: <=, <, >=, >, ==, !=, +, -, *, /.

其他标点符号: {, }, [,], (,), ;, ,

另外还包括数字、标识符等。

对于程序中的空格,一律忽略。而对于每一个换行符,都对全局变量“lineno”进行自增。这一变量是用来记录每一个 Token 在被识别的时候对应的行号,当编译器检测到程序错误的时候可以输出该行号方便检查。这一部分的代码可以参见文件“scanner.l”。使用 lex 运行后可以生成词法分析器代码文件 lex.yy.c。当然对于词法分析器中用到的宏定义,为了与 yacc 生成的语法分析器保持一致,需要加入 yacc 生成的头文件 y.tab.h。

4.2 语法分析以及语法树建立

这一部分由 yacc 生成。Yacc 中需要用到由词法分析器产生的 token, 包括以下:

```
%token INT VOID IF ELSE RETURN WHILE
%token LBRACKET RBRACKET LBRACE RBRACE LPARENT RPARENT
%token LE LT GE GT EQ NE ASSIGN PLUS MINUS TIMES DIVID
%token SEMICOLON COMMA
%token NUM
%token ID
%token ERROR
```

这些 Token 除了 ID 是字符串类型,其他都是整型。另外,在语法分析的过程中需要用到其他一些变量,这些变量是在语法规则中定义的一些标识符。这些中间节点的类型均为自定义的语法树节点类型的指针。声明如下:

```
%union{
    TreeNode* node;
```

```

    char* str;
    int val;
}

%type <val> INT VOID IF ELSE RETURN WHILE LBRACKET RBRACKET
LBRACE RBRACE LPARENT RPARENT
%type <val> LE LT GE GT EQ NE ASSIGN PLUS MINUS TIMES DIVID
SEMICOLON COMMA NUM ERROR
%type <str> ID
%type <val> relop mulop addop
%type <node> program declaration_list declaration
var_declaration type_specifier
%type <node> fun_declaration params compound_stmt param_list
param local_declarations statement_list
%type <node> statement expression_stmt selection_stmt
iteration_stmt return_stmt
%type <node> expression var simple_expression
additive_expression term factor call args arg_list

```

TreeNode 是自定义的语法树的节点类型, 其定义参照文件 global.h, 具体定义如下:

```

typedef struct treeNode{
    struct treeNode* child[MAXCHILDREN];
    struct treeNode* sibling;
    int lineno;
    NodeKind kind; /*main type*/
    union{
        DeclType declType;
        VarType varType;
        StmtType stmtType;
        ExprType exprType;
        TermType termType;
        FactorType factorType;
        ParaType paraType;
    } subkind;

    union{
        int val;
        char* name;
    } attr; /***attribute**/
    int indexOrOperator; /***the index of the array or operator
of a expression***/
    ValueType type; /***the type of this node, for
expression***/

```

```

    StTable* handle;/**the scope of the node in the symbol
table***/
}TreeNode;

```

在语法分析器进行自底向上的语法分析的同时，我们也根据语法规则将识别到的中间节点加入到语法树中。我们将语法树主要分为以下类型：

```

typedef enum {DeclK, ParaK, StmtK, ExprK, VarK,
AddexprK, TermK, FactK, CallK, ArgsK } NodeKind;

```

其中，DeclK 表示该节点为声明类型，ParaK 表示函数参数类型，StmtK 表示语句类型，ExprK 表示表达式类型，VarK 表示变量类型，另外，还有 AddexprK, TermK, FactK, CallK, ArgsK 等类型，这些都可以在语法规则中找到相对应的规则。

另外，每一个基本类型的节点可能对应不同的子类型。例如，一个子句类型的节点可能是表达式子句，也可能是 if 子句，还可能是 while 子句。根据这些情况，我们又有以下的子类型分类：

```

typedef enum {SimVarDecl, ArrVarDecl, FunDecl} DeclType;
typedef enum {ArrayVar, SimpVar} VarType;
typedef enum {ComStmt, ExprStmt, SeleStmt, IterStmt, ReturnStmt}
StmtType;
typedef enum {AssignExp, SimpExp} ExprType;
typedef enum {Multiple, Single} TermType;
typedef enum {Expr, Var, Call, Num} FactorType;
typedef enum {Simple, Array} ParaType;

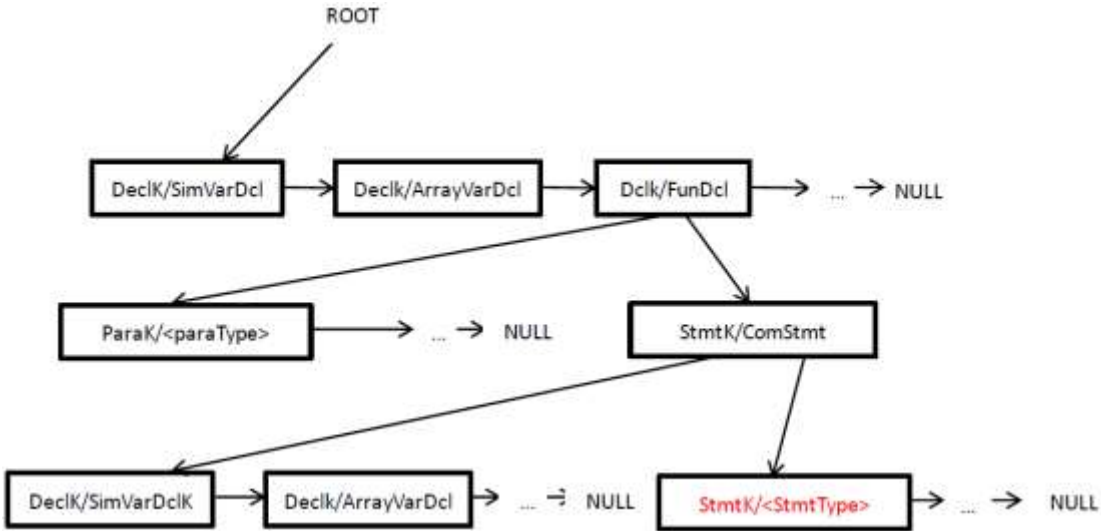
```

为了进一步说明语法树的结构，我们将配合图表进行说明。编译阶段将有一个 root 指针作为整个程序的根指针。

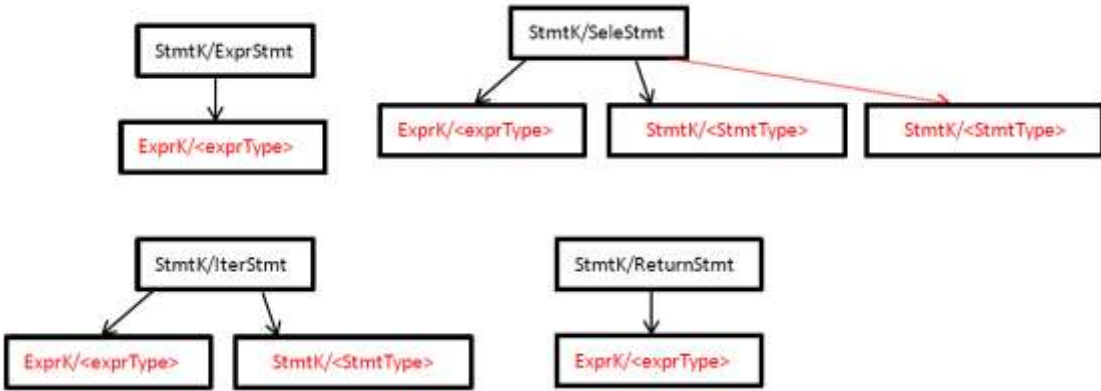
一个程序可以抽象成一些系列的生命，包括简单类型变量的声明，数组变量的声明，函数的声明。Root 指针指向第一个声明，而这些声明是通过他们的兄弟指针（sibling）连接起来。

在这些声明中，只有函数类型的声明节点会有孩子表示这个函数中的一些定义。其第一个孩子指针指向函数参数，这些参数通过兄弟指针连接。第二个孩子指针指向一个 compound statement 类型的节点。

Compound 类型的节点第一个孩子是变量声明，通过兄弟指针连接。第二个孩子是一系列的子句，也通过兄弟指针连接。这些子句可能是不同的类型，因此也有不同的孩子。



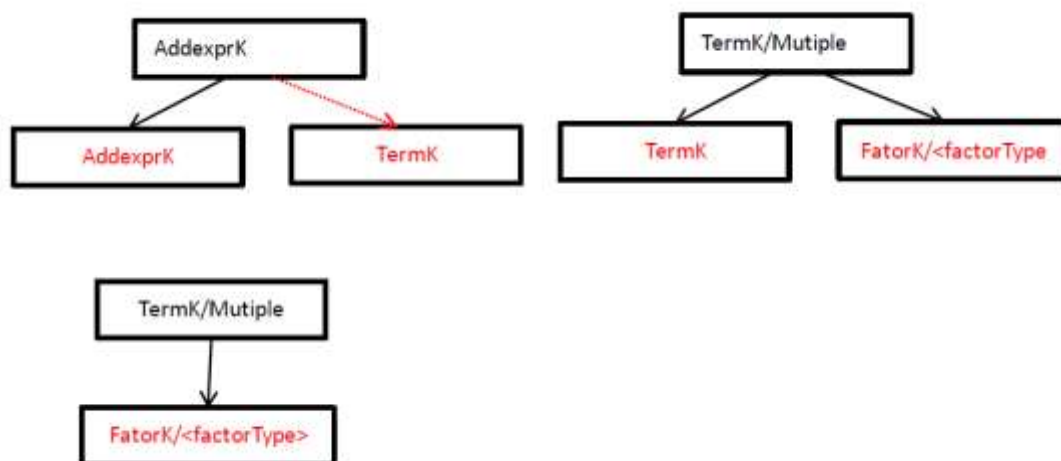
在子句类型中，除了上面已经提到过的 compound 类型外，还有可能是 expression 类型、selection 类型、iterator 类型、return 类型。这些类型的结构如下：



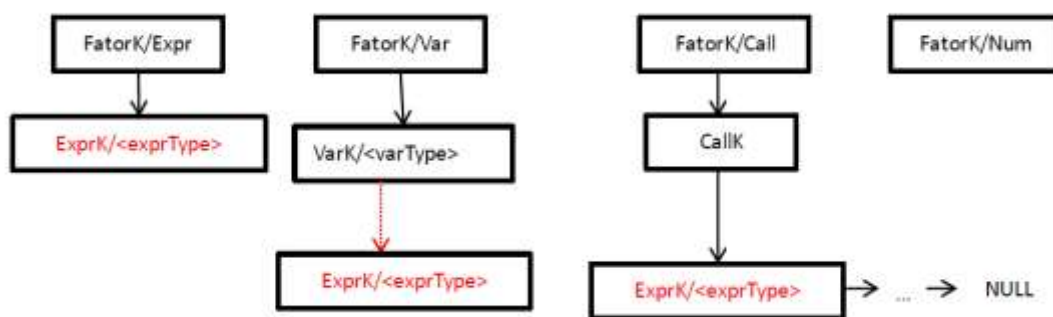
在上面这些节点中，可能会包含表达式节点，表达式节点可能有以下类型：



其中 $\text{addexprK} \mid \text{TermK}$ 表示该节点可能会是这两种类型之一，这其中 TermK 又有两种子类型：



最后就是 Fator 这种节点类型了，这种节点类型可能是以下这几种情况，包括表达式类型，变量类型，函数调用类型，简单的整数类型：



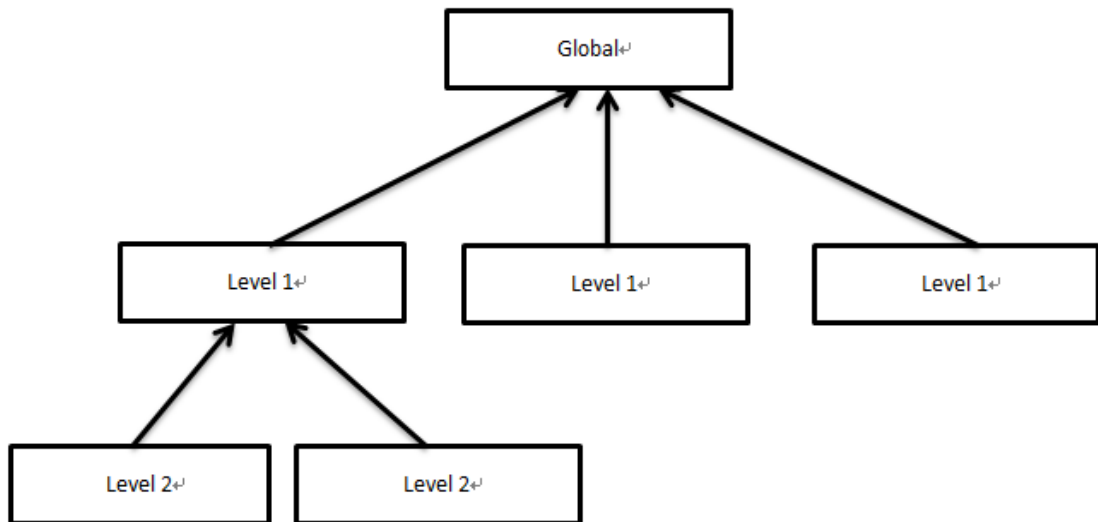
通过在 yacc 中对应的语法规则中添加相应的操作，我们可以再语法分析完成之后构建一棵完整的语法分析树。

为了完成错误恢复功能，当遇到语法错误的时候，可以忽略错误，为该错误创建一个空节点，然后继续分析。

这部分代码参见文件 scanner.l，另外树的节点的定义等一些全局数据结构定义在 global.h 中。需要说明的是我们要在使用 yacc 生成词法分析器的时候加上参数 -d 以便生成头文件 y.tab.h，从而让 lex 从该头文件中得到对应 Token 的宏定义，使系统保持一致。

4.3 符号表建立

符号表是利用哈希表实现的。但是因为不同的标识符可能在不同的作用域，一次我们需要多个哈希表。整个符号表的结构可以表示如下：



整个符号表是一棵反向连接的树，其中每一个节点为一个哈希表，代表一个作用域：Global 是全局作用域，包括全局定义的变量以及一些函数，level 1 是函数定义的变量以及参数，但是一个函数中可能会有子块，每一个子块是一个子作用域，因此需要新建一个符号表并将其指针指向上一级的符号表。这样在标识符插入的时候插入到当前作用域，而查找的时候从当前作用域开始，一级级往上查找，一旦找到对应的记录立刻返回，该记录便是该变量对应的真正记录。另外，为了对所有的符号表进行管理，符号表中将这些节点用链表串起来，这里没有表示出来。

符号表定义在 symbol.h 以及 symbol.c 两个文件中。主要数据结构如下：

```

/*****
 * the structure for store informations
 * *****/
typedef struct stNode{
    char* name; /*identifier name*/
    int addr; /*address during running time*/
    ValueType type; /*type*/
    int count; /*if it is a function, number of parameter*/
    StPara* para; /*parameters*/
    int isFun; /*it is a function or not*/
    struct stNode* next;
} StNode;
/*****
 * symbol table, a table is also a scope
 * *****/
typedef struct stTable{
    StNode* table[SIZE];
    ValueType funType; /*type of the current function, for
checking return*/
    struct stTable* parent;

```

```
} StTable;
```

符号表的建立在 `analysis.c` 中定义。主要是通过函数 `buildSymtab()` 对语法树进行先序遍历。当遇到声明节点时，将标识符以及是否为函数等信息加入到符号表中。进入函数或者进入新的作用域的时候，需要调用函数 `StTable* stNewScope(StTable* table)`；创建新的作用域，并将新的作用域指向上一级作用域的表 `table`。（每一个树节点都有一个 `StTable` 类型的指针指向对应的作用域符号表）。如果出现重复定义的时候，将 `Error` 置为 1，继续执行。遍历完成之后检查 `Error` 以确定时候进入类型检查阶段。

4.4 语义分析以及类型检查

这一阶段对程序进行语义检查，包括检查使用的变量、函数是否已经定义，函数、变量类型是否匹配，调用函数的参数个数、类型是否匹配等。主要是对语法树进行后序遍历，将类型等综合属性往上传递，同时配合前面已经建立的符号表的信息进行检查。在往上传递的过程中，如果遇到操作符量表的操作数类型不匹配，复制语句中将不同类型的数值复制给某一变量，函数调用的时候参数数量不匹配或者对应位置的参数类型不匹配等情况，则将 `Error` 置为 1，同时将冲突情况采用一些默认值进行处理，然后继续执行。

这一部分代码参见文件 `analysis.h` 以及 `analysis.c`。

4.5 代码生成

这一阶段的主要目标是生成 `nasm` 汇编代码，其后的汇编、连接都由工具完成，无需过多关注。

首先，将 `nasm` 汇编的语句模型定义好，其后代码生成的时候只需调用这些语句模型，传入相应的参数，便能生成所需的 `nasm` 汇编代码。同时，在 `nasm` 中各变量名、函数名不方便使用原代码中的名字。在此，我们选用了经典的 ELF hash 函数，将各变量名、函数名转换为哈希值。在本系统中，我们不调用标准的 C 库，对于输入输出，我们通过编写 `nasm` 汇编实现。这部分代码可以参见 `encode.c/h` 文件。

接下来，在遍历语法树的过程中，调用相应的 `nasm` 汇编语句模型，生成相应的汇编代码。对于全局变量，将其定义在 `bss` 段中；其后是常量的 `data` 段。在函数定义前，将系统自身的输入、输出函数代码加入。对于 `main` 函数，将其定义为 `global _start` 函数代码段，作为代码的函数入口。调用函数前，需要将该函数所需的变量值压栈，然后调用 `call`，跳转到目标函数代码段。在目标函数代码段中，先将参数值按压栈的反顺序取出，进行函数的运算，最后如果有返回值，则将其压栈，在函数调用处再取出。这一系列的处理过程，都是通过语法树的变量实现。在语法树中，每个节点都有定义有其节点类型，然后再根据其子类型来进行具体的操作。当遇到变量、函数时，在其定义时需要将其的栈偏移存入符号表中，以便以后调用时查找；在其调用处，查询符号表中的栈偏移及其是否为数组，在将其作为参数生成具体的 `nasm` 代码。

5. 测试及结果

5.1 编译阶段测试

在阶段测试中，将输出语法树以及符号表，方便测试。例如以下程序，使用到了大部分 C-语言的元素。

```

1 int a;
2 int b;
3 int main(void)
4 {
5     int a;
6     if(a==1){a=1;}
7     if(a==2){a=2;}
8     else{a=3;}
9     while(a==2){a=1;}
10    return 0;
11 }

```

对应的语法树如下（先序遍历输出）：

```

jys@jys-Aspire-4736:~/workspace/compiler$ ./a.out pro
1 DeclK
2 DeclK
3 DeclK
|11 StmtK
||5 DeclK
||7 StmtK
|||6 ExprK
||||6 TermK
|||||6 FactK
||||||6 VarK
|||||6 TermK
|||||6 FactK
|||6 StmtK
|||6 StmtK
||||6 ExprK
|||||6 VarK
|||||6 ExprK
||||||6 TermK
|||||||6 FactK

||8 StmtK
|||7 ExprK
|||7 TermK
|||7 FactK
|||7 VarK
|||7 TermK
|||7 FactK
||7 StmtK
||7 StmtK
|||7 ExprK

```

```

|||||7 VarK
|||||7 ExprK
|||||7 TermK
|||||7 FactK
|||8 StmtK
|||8 StmtK
|||8 ExprK
|||8 VarK
|||8 ExprK

|||||8 TermK
|||||8 FactK
||9 StmtK
||9 ExprK
|||9 TermK
|||9 FactK
||||9 VarK
|||9 TermK
||||9 FactK

|||9 StmtK
|||9 StmtK
|||9 ExprK
||||9 VarK
||||9 ExprK
||||9 TermK
||||9 FactK
||10 StmtK
||10 ExprK
|||10 TermK
||||10 FactK

```

其中前面的“|”符号表示层数，没有前置“|”符号即表示为全局节点。可以对应源程序以及输出的语法树，检查语法树的正确性。

另外，可以查看符号表，如以下源程序：

```

1 int a;
2 int b;
3 int fun(int p){}
4 int main(void)
5 {
6     fun(1);
7     return 0;
8 }

```

对应符号表如下，不同作用域之间用“====”间隔。可以看出全局有 main, fun, a, b，在另一个作用域里有 p 变量。其中每一个标识符后面第一个数字表示类型，1 为 int，第二个数字为行号。

```

=====
main|1|4
fun|1|3
a|1|1
b|1|2
=====
=====
p|1|3
=====

```

下面将根据 C-对应的语法特性进行测试。

1、首先测试使用未定义的标量：

```

1 int a;
2 int b;
3 int main(void)
4 {
5     a=1;
6     c=2;
7     return 0;
8 }

```

error:line 6:c is not a variable

2、下面测试编译阶段的错误恢复功能，三个未定义变量：

```

1 int a;
2 int b;
3 int main(void)
4 {
5     c=2;
6     d=3;
7     e=5;
8     return 0;
9 }

```

```

=====
error:line 5:c is not a variable
error:line 6:d is not a variable
error:line 7:e is not a variable

```

3、下面测试数据类型，程序通过编译，输出语法树以及符号表：

```

1 int a;
2 int b;
3 int main(void)
4 {
5     int a;
6     int b [100] ;
7     return 0;
8 }

```

```

1 DeclK
2 DeclK
3 DeclK
| 8 StmtK
|| 5 DeclK
|| 6 DeclK
|| 7 StmtK
||| 7 ExprK
|||| 7 TermK
||||| 7 FactK
=====
main|1|3
a|1|1
b|1|2
=====
a|1|5
b|1|6
=====

```

4、下面测试重复定义：

```

1 int a;
2 int a;
3 int main(void)
4 {
5     int a;
6     return 0;
7 }
8 int main(void)
9 
```

```

-----
error:line 2:redclaration of variable a
error:line 8:redclaration of function main

```

5、测试函数参数个数匹配问题:

```

1 int fun(int a)
2 {
3     return 0;
4 }
5 int main(void)
6 {
7     fun(1,2);
8 }

```

```

error:line 7:arguement number not compatible

```

6、测试函数参数类型匹配问题

```

1 int fun(int a)
2 {
3     return 0;
4 }
5 int main(void)
6 {
7     int a;
8     fun(a==1);
9 }

```

```

-----
error:line 8:arguement type not compatible

```

7、测试 if 表达式条件必须是 bool 类型（区分 int 以及 bool 类型）

```

1 int fun(int a)
2 {
3     return 0;
4 }
5 int main(void)
6 {
7     int a;
8     if(a){}
9 }

```

```

error:line 9:condition must be a boolean value

```

8、测试加法表达式操作数匹配情况，正确并输出语法树以及符号表:

```

1 int main(void)
2 {
3     int a;
4     a=1+2;
5 }

```

```

1 DeclK
|5 StmtK
||3 DeclK
||4 StmtK
|||4 ExprK
|||4 VarK
|||4 ExprK
||||4 AddexprK
|||||4 TermK
|||||4 FactK
|||||4 TermK
|||||4 FactK
=====
main|1|1
=====
a|1|3
=====

```

9、测试操作数类型不匹配情况（int 类型与 bool 类型相加）

```

1 int main(void)
2 {
3     int a;
4     a=1+(2==1);
5 }

```

error:line 4:operand type not compatible

10、测试函数参数类型为数组类型的传递问题:

```

1 int fun(int a[]){}
2 int main(void)
3 {
4     int a;
5     fun(a);
6 }

```

error:line 5:arguement type not compatible

本阶段的更多测试也体现在最终的编译器的运行结果上。

5.2 生成代码及最终测试

语句功能测试:

1. 测试单局部变量及 output() 系统函数
测试程序源码（test/t00.c）:

```

void main(void) {
    int a;
    a = 0;
    output(a);
}

```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
0

```

分析:

正确输出全局变量值，测试通过。

2. 测试单全局变量

测试程序源码 (test/t01.c):

```
int a;  
int main(void) {  
    a = 1;  
    output(a);  
    return 0;  
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh  
elf@elf-R458:~/program/compiler/compiler/test$ ./t  
1  
elf@elf-R458:~/program/compiler/compiler/test$
```

分析:

正确输出局部变量值，测试通过。

3. 测试有参数的函数调用

测试程序源码 (test/t02.c):

```
int a;  
int sum(int a, int b) {  
    return a+b;  
}  
  
int main(void) {  
    int b;  
    a = 1;  
    b = 2;  
    output(sum(a,b));  
    return 43;  
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ../a.out t02.c  
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh  
elf@elf-R458:~/program/compiler/compiler/test$ ./t  
3  
elf@elf-R458:~/program/compiler/compiler/test$
```

分析:

全局变量和局部变量均作为参数传入函数中，函数返回值正确，测试通过。

4. 测试无参数的函数调用

测试程序源码 (test/t03.c):

```
int a(void) {  
    return 4;  
}  
  
int main(void) {  
    output(a());  
    return 0;  
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./a.out t03.c
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
4
```

分析:

函数返回值正确, 测试通过。

5. 测试多个有参数函数调用

测试程序源码 (test/t04.c):

```
int a;
int b;

int sum(int a, int b) {
    return a+b;
}

int sub(int a, int b) {
    return a-b;
}

int mul(int a, int b) {
    return a*b;
}

int div(int a, int b) {
    return a/b;
}

int main(void) {
    a = 7;
    b = 3;
    output(sum(a, b));
    output(sub(a, b));
    output(mul(a, b));
    output(div(a, b));
    return 0;
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t04
10
4
21
2
```

分析:

各函数返回值正确, 测试通过。

6. 测试嵌套有参数函数调用

测试程序源码 (test/t05.c):

```
int add(int a) {
    return a + 1;
}
```

```
void main(void) {
    output (add(add(add(0))) );
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
3
```

分析:

结果正确, 测试通过。

7. 测试嵌套的作用域

测试程序源码 (test/t06.c):

```
int main(void) {
    int a; int b;
    a = 10; b = 10;
    {
        int b;
        b = 1;
        {
            int a;
            a = 2;
            output(a); output(b);
        }
        {
            int b;
            b = 3;
            output(a); output(b);
        }
        output(a); output(b);
    }
    output(a); output(b);
    return 0;
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
2
1
10
3
10
1
10
10
```

分析:

各作用域中变量值不冲突, 输出结果均正确, 测试通过。

8. 测试单 if 语句

测试程序源码 (test/t10.c):

```
void main(void) {
    int a;
    int b;
```

```

a = input();
if(a!=0)
    b = a;
else
    b = 1;

output(b);
}

```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ../a.out t10.c
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
0
1
elf@elf-R458:~/program/compiler/compiler/test$ ./t
10
10

```

分析:

当输入为 0 时，输出 1；当输入为正时，输出原值。结果正确，测试通过。

9. 测试嵌套 if 语句

测试程序源码 (test/t11.c):

```

void main(void) {
    int a;
    a = input();
    if(a >= 0)
        if(a == 0)
            output(1);
        else
            output(2);
    else
        output(0);
}

```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
5
2
elf@elf-R458:~/program/compiler/compiler/test$ ./t
0
1
elf@elf-R458:~/program/compiler/compiler/test$ ./t
-2
0

```

分析:

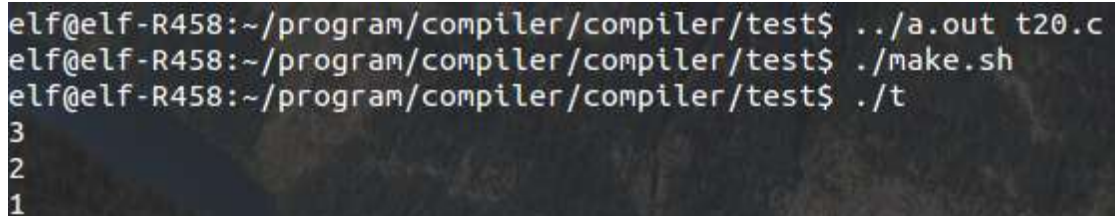
当输入正时，输出 2；当输入 0 时，输出 1；当输入负时，输出 0。结果正确，测试通过。

10. 测试 while 语句

测试程序源码 (test/t20.c):

```
void main(void) {  
    int a;  
    a = 3;  
    while(a > 0) {  
        output(a);  
        a = a - 1;  
    }  
}
```

测试结果:



```
elf@elf-R458:~/program/compiler/compiler/test$ ./a.out t20.c  
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh  
elf@elf-R458:~/program/compiler/compiler/test$ ./t  
3  
2  
1
```

分析:

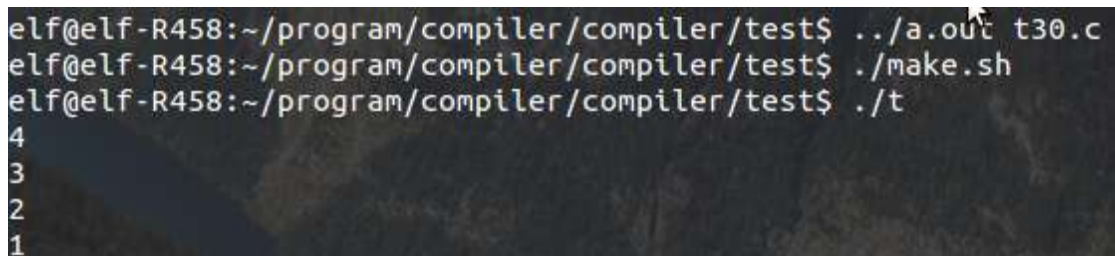
循环将 3 至 1 的数输出, 结果正确, 测试通过。

11. 测试局部数组

测试程序源码 (test/t30.c):

```
int main(void) {  
    int a[4];  
    int i;  
    i = 4;  
    while(i > 0) {  
        a[i] = i;  
        output(a[i]);  
        i = i - 1;  
    }  
    return 0;  
}
```

测试结果:



```
elf@elf-R458:~/program/compiler/compiler/test$ ./a.out t30.c  
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh  
elf@elf-R458:~/program/compiler/compiler/test$ ./t  
4  
3  
2  
1
```

分析:

局部数组值读写正确, 测试通过。

12. 测试全局数组

测试程序源码 (test/t31.c):

```
int a[4];  
int sum(void) {  
    int a[5];  
    a[3] = 33;  
    return a[3];  
}  
void main(void) {
```

```

a[3] = 3;
output(a[3]);
output(sum());
}

```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ../a.out t31.c
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
3
33

```

分析:

全局数组读写正确, 测试通过。

复杂功能测试:

1. 测试最大公约数程序

测试程序源码 (test/test1.c):

```

int gcd(int u, int v)
{
    if (v == 0)
        return u;
    else
        return gcd(v, u - u / v * v);
}

int main(void)
{
    int x;
    int y;
    x = input();
    y = input();
    output(gcd(x, y));

    return 0;
}

```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
42
72
6

```

分析:

最大公约数输出正确, 测试通过。

2. 测试选择排序程序

测试程序源码 (test/test2.c):

```

int x[10];

int minloc(int a[], int low, int high)
{
    int i;
    int x;

```

```
int k;
k = low;
x = a[low];
i = low + 1;
while (i < high) {
    if (a[i] < x) {
        x = a[i];
        k = i;
    }
    i = i + 1;
}

return k;
}

void sort(int a[], int low, int high)
{
    int i;
    int k;
    int t;
    i = low;
    while (i < high - 1) {
        k = minloc(a, i, high);
        t = a[k];
        a[k] = a[i];
        a[i] = t;
        i = i + 1;
    }
}

int main(void)
{
    int i;
    i = 0;
    while (i < 10) {
        x[i] = input();
        i = i + 1;
    }
    sort(x, 0, 10);
    i = 0;
    while (i < 10) {
        output(x[i]);
        i = i + 1;
    }
    return 0;
}
```

测试结果:

```

elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
4
2
4
5
21
6 4\nmov dword [ebp-4], 0\nmov byte [inputchar], 0\njmp
3
0 dword eax, [ebp-4]\nmov ebx, 10\nmul ebx\nxor ecx, ecx
24
9 cl, [inputchar]\nsub ecx, 48\nadd eax, ecx\nmov dword
0
2 eax, 03h\nmov ebx, 06h\nmov ecx, inputchar\nmov edx, 0
3
4 cmp byte [inputchar], 0ah\njne G5\nmov dword eax, [ebp
4
5
6
9
21
24 eax\nmov eax, 1\nint 80h");

```

分析:

数组排序正确, 测试通过。

3. 测试递归阶乘程序

测试程序源码 (test/test3.c):

```

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

int main(void) {
    int m;
    int result;
    m = input();
    if (m > 1)
        result = fact(m);
    else
        result = 1;
    output(result);
    return 0;
}

```

测试结果:


```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
0
1
elf@elf-R458:~/program/compiler/compiler/test$ ./t
6
720
```

分析:

阶乘计算正确, 测试通过。

4. 测试递归斐波那契数列程序

测试程序源码 (test/test4.c):

```
int fib(int n)
{
    int result;
    if (n < 2)
        return n;
    return fib(n-1) + fib(n-2);
}

int main(void)
{
    int n; int i;
    i = 0;
    n = input();
    while (i <= n) {
        output(fib(i));
        i = i + 1;
    }

    return 0;
}
```

测试结果:

```
elf@elf-R458:~/program/compiler/compiler/test$ ./make.sh
elf@elf-R458:~/program/compiler/compiler/test$ ./t
10
0
1
1
2
3
5
8
13
21
34
55
```

分析:

斐波那契数列输出正确, 测试通过。

6. 说明以及体会

关于程序的编译以及运行，可以参考源程序目录中的 README。我们已经将程序的编译以及运行命令写在 shell 脚本里，但是需要使用机器已经安装 gcc、flex、yacc、nasm，并且能运行 linux shell。详细命令可以参见 shell 脚本。

通过这个实验，我们对编译的基本原理有了更进一步的理解。也体会到原理与实践之间的区别，原理经过前任提炼出来的，我们在实践的过程中可能会遇到各种各样的困难，这需要我们自己去设计、解决。另外，由于后半阶段时间较紧，因此没有进行课堂展示，也是一个遗憾。