# 《编译原理课程设计》

# ZCC

## 一个简单的类 C 编译器

1452286

朱可仁

# 1. 代码使用说明

## 1.1 目标平台

ZCC 在 Linux subsystem on Windows 10 下开发。目前可生成兼容 System V x86_64 ABI 标准的 AT&T 格式汇编程序，可在 64 位 Linux 任意发行版上运行。ZCC 需要链接器生成二进制程序，请确保在有链接器的 Linux 环境下测试运行。

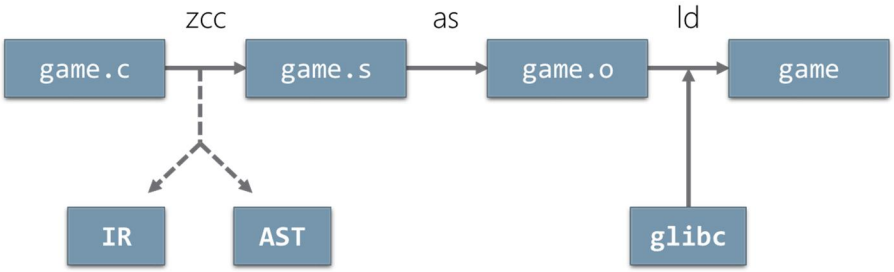


图 1. ZCC 编译流程

## 1.2 运行

ZCC 目前支持以下 3 种输入。

*屏幕打印中间代码*

```
zcc -ir src.c
```

*屏幕打印汇编代码*

```
zcc -a src.c
```

*输出汇编代码到文件*

```
Zcc -a src.c -o asm.s
```

目前 ZCC 会默认打印语法树。暂不支持通过外部参数关闭打印。

输出到汇编代码后，可使用 gcc asm.s 生成二进制程序。

## 1.3 构建

在 ZCC 源代码目录下使用 make 命令生成 zcc 可执行文件。该命令需要 GCC 支持。

# 2. 功能描述

ZCC 是尚处在开发的一个编译器，目前仅支持编译 C 的一个子集。以下是目前 ZCC 功能的支持情况。

**数据类型：**int, char*(字符串常量)
**流程控制：**函数调用、递归调用

while, for

break, continue

if, else

**运算符：** +, -, *，括号表达式

前缀后缀 ++, --

<, <=, >, >=, ==, !=

## 3. 内部实现

ZCC 是一个一趟生成、语法制导的 C 编译器，由词法分析器 lex，语法分析器 parse，符号管理模块 sym，目标代码生成模块 gen 和辅助工具 utility 模块五个部分组成。其中 sym 模块除一般意义上的符号管理外，还负责一切随解析过程中运行时环境支持。ZCC 不生成语法树，而是直接生成三地址代码。gen 模块负责将三地址代码转换为目标代码，并最后借助外部的链接器链接系统函数，生成可执行程序。除 utility 外的模块中的公共变量名、函数名，皆冠以对应的前缀。如 lex_token 代表 lex 负责生成的 token 变量，可供程序其他部分使用。

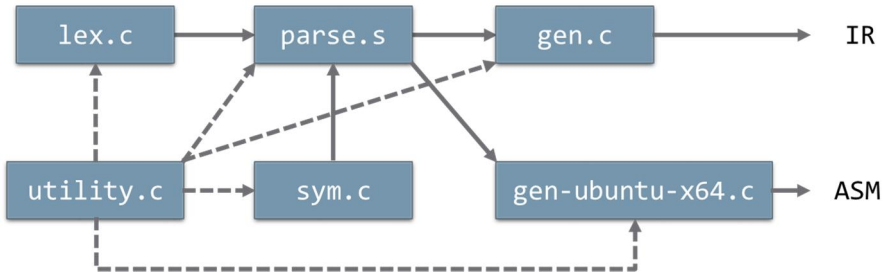


图 2. ZCC 模块

下面将就各个模块给出详细设计说明。

### 3.1 词法分析器

ZCC 的词法分析器根据 C 的语法直接写出，没有使用如正则表达式或 FDA 等技术。词法分析器读入一个文件，输出 token_st 类型的 token 流。

Lex 模块通过 token_st 类型的 lex_token 讲下一个未读的 token 暴露给程序的其他部分。Lex 提供通过 lex_next()函数，读入下一个 token；Lex_valid()函数，判断是否已读到文件末尾。

Lex 模块能够正确的处理空格、注释。但是不能处理包括 define, include 在内的，属于预编译部分的宏命令。

token_st 结构体记录了一个 token 的所有信息。其定义如下

```
typedef struct token_st {
    enum tk_class_en tk_class;
    char *tk_str;
    value_un *value;
} token_st;
```

其中，tk_class 标识该 token 的类型。tk_class 只能是下面列表中的一项。

```
TK_OP_DEREF
TK_OP_INC
TK_OP_DEC
TK_OP_LAND
TK_OP_LOR
TK_OP_EQ
TK_OP_NEQ
TK_OP_LE
TK_OP_GE
TK_OP_ADD
TK_OP_SUB
TK_OP_MUL
TK_OP_DIV
TK_OP_BAND
TK_OP_BXOR
TK_OP_BOR
TK_OP_MOD
TK_OP_LT
TK_OP_GT
TK_OP_LNOT
TK_OP_BNOT
TK_OP_REF
TK_OP_ASSIGN
TK_OP_ASSIGN_ADD
TK_OP_ASSIGN_DEL
TK_OP_ASSIGN_MUL
TK_OP_ASSIGN_DIV
TK_OP_ASSIGN_MOD
TK_OP_ASSIGN_BAND
TK_OP_ASSIGN_BXOR
TK_OP_ASSIGN_BOR
/* ^运算符===  V 非运算符*/
TK_CONST_INT
TK_CONST_STRING
TK_CONST_CHAR
TK_IDENTIFIER
TK_DELIMITER
```

tk_class 可分为运算符和非运算符两类。Lex 给整数、字符串常量以特殊的 class。对于[,(,),],;等分界作用的符号，统一赋予 TK_DELIMITER;对于关键字和可能的函数名、变量名，统一赋予 TK_IDENITFIER。

实现上，为了方便管理，ZCC 使用了一个单独的 operators.h 文件，记录运算符和其对应的 class

名，结合性和优先级。operators.h 中的一条命令如下所示。

```
yy(TK_OP_INC,       "++", ASSO_L,     14)
```

在 lex.c 中，通过#define yy(...) 和 #include “operators.h”可方便的对各种符号进行操作。ZCC 中还有多处使用了同样的技术，这里不再一一列举。

token_st 结构体中，另有一 union value_st*类型的变量 value。lex 解析到整数常量时会更新 value 的值方便 parse 模块使用。

### 3.2 语法分析器

与 bison 等自动化工具相比，手写的递归下降语法解析器速度快，易于调试，且能够提供更好的错误处理。目前主流的 CXX 编译器前端如 GCC， clang 都是使用这种写法。ZCC 同样使用一个手写的递归下降分析器。

ZCC 使用的语法主要在《LCC: A retargetable C compiler》 parser 章节上修改而来。这个语法非 LR(1)语法，有少量 LR(2)文法，少量 LR(k)文法(k 未知较大)。以下列出 ZCC 使用的 BNF 文法，并对其中特殊处理的部分做出说明。注：特殊说明的部分都可以在 parse.c 的源码中找到。

```
expression:
    assignment-expression { , assignment-expression }

assignment-expression:
    conditional-expression
    unary-expression assign-operator assignment-expression
assign-operator:
    one of = += -=  *=  /=  %=  <<= >>=  &=  ^=  |=

Note:
    It's hard to choose from one of the productions.
    So we merge them into
    => conditional-expression assign-operator assignment-expression

    This leads to incorrect expresstion like a + b = c gets accepted
    But we could leave error detection later to semantic stage
    i.e. reject input once we find a+b is not lvalue.
```

注 1：由于 conditional-expression 和 unary-expression 的 first 集有相同的终结符，且仅通过向前读取 token 很难判断应该取两者中的哪一个，ZCC 将两个生成式合并。这样的语法允许等号(=)左侧出现注入 a+b 这样的非左值表达式，不过我们可以通过 sym 模块进行语义分析，确保(=)左边是可以赋值的左值。

```
conditional-expression:
```

```
    binary-expression [ ? expression : conditional-expression ]

 binary-expression:
    unary-expression { binary-operator unary-expression }

 Note:
    prs_binary(k): parse binary expression of precedence k or higher
    precedence defined in operators.h
    trick avoiding deep recursion descripted in LCC Chap.8.6
```

注 2：纯递归下降分析法在解析二元运算符时，需要为每个优先级构造一个递归下降函数。这样会导致解析表达式时出现长串嵌套的函数调用，且其中大部分函数不会读取 token，浪费了大量的时间。为了解决这个问题，ZCC 使用了算符优先分析法。即，二元运算符解析函数 prs_binary(k)传入一优先级 k，由这层的 prs_parse 负责解析由优先级大于等于 k 的运算符连接的表达式。

```
 unary-expression:
    postfix-expression
    unary-operator unary-expression
    '(' type-name ')' unary-expression
    sizeof unary-expression
    sizeof '(' type-name ')'

postfix-expression:
    primary-expression { postfix-operator }
postfix-operator:
    '[' expression ']' // array index.   e.g. a[0]
    . identifier
    -> identifier
    ++
    --

 Note: assume primary-expression parsed by caller and passed(?) in.
```

注 3：生成式 postfix-expression 和'(' type-name ')' unary-expression 的 first 集有公共终结符’(‘，需要多读入一个 token 才能判断是那个表达式。注意到 postfix-expression 的生成式只有一项。我们可以多读一个 token，判断它是不是 type-name,若不是，则它必是 postfix-expression 生成的 primary-expression。实现中，prs_pst(var_st*)可传入一指针，表示预读取的 primary-expression。

```
primary-expression:
    identifer
    constant
    string-literal
```

```
    '(' expression ')'
    function-call

function-call:
    identifier '(' argument-list ')'


argument-list:
    assignment-expression { , assignment-expression }

statement:
    ID : statement
    case constant-expression : statement
    default : statement
    [ expression ] ;
    if '(' expression ')' statement
    if '(' expression ')' statement else statement
    switch '(' expression ') ' statement
    while '(' expression ') ' statement
    do statement while '(' expression ')' ;
    for '(' [ expression ] ; [ expression ] ; [ expression ] ')' statement
    break ;
    continue ;
    goto ID ;
    return  [ expression ] ;
    compound-statement
compound-statement:
    '{' { declaration } { statement } '}'

Note:
    Seems like declaration is only allowed at the beginning of the block.
    This is subject to change but leave it for now.
```

注 4：目前的语法只允许 ZCC 在每一个语句块的开头声明变量。

```
declarations:
    { declaration }

declaration:
    declaration-specifiers init-declarator { , init-declarator } ;
    declaration-specifiers func-declarator compound-statement

init-declarator:
    identifier
```

```
    identifier = assignment-expression

func-declarator:
    identifer '(' parameter-list ')'

parameter-list:
    parameter { , parameter } [ , ... ]

parameter:
    declaration-specifiers identifier

declaration-specifiers =>
    int
    char
```

对于所有的 expression 系列函数，它必有一个返回值，由 var_st*标记。var_st 充当了多重责任，它即可表示局部变量，也可表示实参，也可表示运算临时变量，也可承载立即数（即整数、字符串常量）。

### 3.3 语义分析

由于是语法制导编译器，语义分析由 parse 模块和 sym 模块共同负责。Parse 在解析 token 流的过程中，使用 sym 模块控制当前运行时环境。sym 模块发现有非法语义的情况，则提示错误，终止编译。

典型的几个语义分析包括

(1) 进入、退出作用域

(2) 定义局部变量（同时判断重定义）

(3) 查找局部变量

(4) 判断赋值合法性

(5) 判断运算合法性

其中，(5)判断运算合法性由于 ZCC 目前支持的数据类型很有限，基本不会涉及。

剩下的几个语义分析操作我们将在后文叙述。

### 3.4 符号管理

ZCC 的符号管理由 sym 模块负责。符号管理模块主要负责以下几块内容

(1) 管理变量作用域

(2) 管理函数栈帧变量分配（包括局部变量、实参和临时变量）

(3) 管理跳转标签(label_st)

Sym 模块管理一全局 context_st 结构体，随着解析的进行，sym 通过 context_st 管理当前环境的变化，从而实现语义分析。

### 3.4.1 管理作用域

ZCC 使用 scope_st 结构体管理作用域。Context_st 有一 scope 变量指向当前作用域。解析进入每个程序块，包括新函数、if 等区块语句时，parse 模块调用 scope_mnp_scope()进入新 scope。退出程序块时，parse 模块调用 scope_pop_scope()推出 scope。Scope 是一个串联的数据结构，可用下图表示。

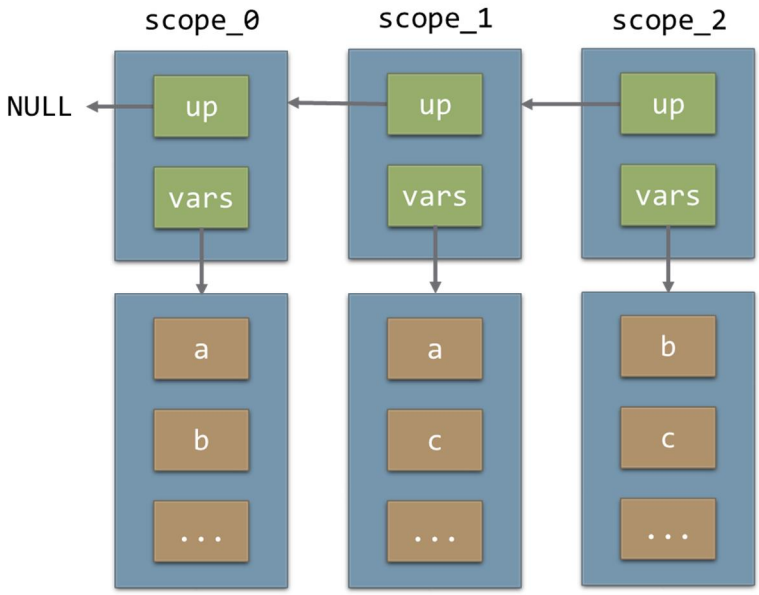


图 3. Scope_st 串联示意图

每个 scope_st 主要包含 up 和 vars 两个指针。Up 指向上一级 scope, vars 则指向一个链表（在实现中其实是一个变长数组），内存有一系列的 var_st，用以表示目前在这一级中存在的局部变量。

scope_st 还提供三个 label_st*指针, forC, forB 和 forE。他们用于 break 和 continue 的跳转。当 Parse 解析到 while 或 for 时，首先进入心的 scope，然后成三个 label_st，分别代表循环体判断条件、正文和结束的位置。此时，forB 和 forE 的位置通常不是确定的。但预先生成并保存在 scope 中使得我们可以在 for 循环体解析完毕后，向中间代码插入之前生成的 forE，从而确定 forE 的位置。注意到，这三个 label 必须放在 scope 下，而不是 context 下。因为循环可能有多层，每一层都可能有 break 和 continue。

Sym 提供 sym_redefined_var(name)在当前作用域查找名为 Name 的变量。Parse 模块用此来判断变量重定义。

### 3.4.2 管理函数栈帧分配

C 不允许嵌套函数，因此可在 context 中使用一 func_st 指针指向当前正在解析的函数。func_st 结构体维护着一个函数的所有信息，包括函数名、中间代码、返回值类型、形参和栈帧分配（注意局变由 scope 而不是 func 维护）。func_st 的定义如下。

```
typedef struct func_st {
    char    *name;
    struct type_st *rtype;
    struct list_st *pars; /* list of var_st */
```

```
    struct list_st *insts;
    struct var_st  *ret;
    /* for gen */
    int rbytes; /* bytes should reserved for local variables */
} func_st;
```

其中，rbytes 标记了这个函数在生成目标代码时应该预留的栈帧空间。这部分预留空间不仅给局部变量用，也给运算过程中的中间临时变量用。随着函数的解析，rbytes 应该是一个逐渐增大的变量。

Parse 模块使用 sym 模块提供的 sym_make_var(name, type), sym_make_temp_var(type), sym_make_imm(token)分别生成局部变量、临时变量和用于存放立即数的变量。同时，sym 模块还提供 sym_dispose_temp_var(var)函数回收临时变量。sym_make_var 一定会导致 rbytes 的增大，sym_make_temp_var 和 sym_make_imm 则会尽量使用被遗弃的临时变量。Sym 使用一个链表维护被 disposed 的临时变量。这项优化需要 parse 模块的配合。目前，ZCC 仅在临时变量使用量最大的几个区域 dispose 临时变量，因此还有很大的优化空间。

Var_st 结构体中有一 lvalue，表示该变量是否是左值。只有使用 sym_make_var 生成的变量 lvalue 才会为 1。Parse 模块用此判断非法赋值。

注：x64 平台的内存 8 字节对齐。Sym 中栈帧总是 8 字节 8 字节地分配。

### 3.5 中间代码

ZCC 使用三地址的中间代码。大部分中间指令是 OP A B C 的格式。其中 A 一般是目标，B 和 C 则是操作数。所有的代数运算、比较运算，都是此格式。以下是 ZCC 目前支持的所有中间指令。

```
ADD/SUB/MUL A, B, C
GT/GE/EQ/LT/LE/NEQ A, B, C
MOV A, B
LABEL LNAME
RET
MOV A CALL FUNC(ARGS_LIST)
CJMP COND, LNAME
JMP LNAME
INC/DEC A
```

在 ZCC 中，所有的操作数都是指针。可能是 var_st, label_st 或 func_st 指针。ZCC 的中间代码不能输出重用，仅用于调试。

以下特殊说明 2 条语句

(1) MOV A CALL FUNC(ARGS_LIST)

这是 ZCC 的函数调用代码。调用函数 FUNC，实参为 ARGS_LIST，保存返回值在 A 变量。其中 ARG_LIST 是一个保存着 var_st*类型的链表。这大概与一般的编译器很不一样（我也不知道别的编译器怎么做的）。这条语句将被拆分为多条语句，而具体的工作由 gen 模块完成，

会根据目标平台的不同而不同。

(2) `CJMP COND, LNAME`

这是 ZCC 的条件跳转语句。ZCC 不适用例如 JL, JG 等多样化的条件跳转，而是仅判断 cond 是否为 1，是则跳转。这简化了跳转语句的实现。在 x86_64 平台上，这需要 movzx(zero-extened mov)语句的辅助，讲判断语句的结果存进变量中。

### 3.6 目标代码生成

gen 模块负责生成 IR 和目标代码并写入到文件。ZCC 支持多平台目标代码。目前我实现了 linux x86_64 目标汇编代码的生成。需要编译到不同平台时，可编写 gen-x.c 文件，并与其他文件链接。

gen 根据指令集、操作系统的不同而不同。例如，System V x86_64 的函数调用遵循一下的参数传递规则。

```
long myfunc(long a, long b, long c, long d,
           long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

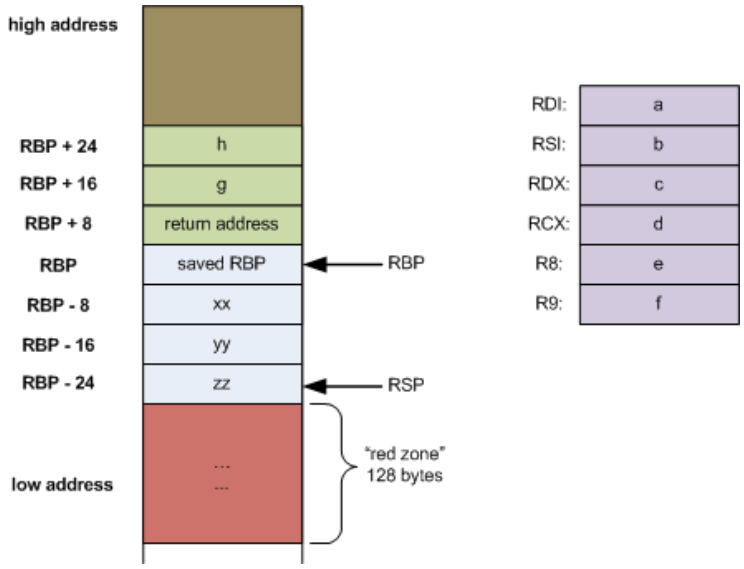


图 4. System V x86_64 参数传递示意图

注意到，前 6 个函数参数是通过寄存器传递的，余下的参数通过栈传递。并且，栈上的参数应该从参数表的后面开始压栈。

正确实现符合系统标准的函数参数传递是保证链接成功的关键。再实现 ZCC 链接 printf 的过程中，我的程序曾经出现了运行崩溃的 BUG。经过查阅资料得知，在 system V 标准中，

调用 printf 这样的可变参数函数，必须将%rax 寄存器赋值为传递的浮点数参数的个数。这样的细小的操作系统细节，只有实现过编译器的人才能亲身体会到。

## 4. 成果实测

我准备了 demo.c 测试程序，包含 4 个测试
（1）nested 测试变量作用域
（2）expr 测试复杂表达式
（3）loop 测试循环
（4）recursive() 测试递归
测试程序如下

```
int nested() {
    int a = 1;
    printf("==== Nested Test ====\n");
    printf("scope1[a=%d]\n", a);
    if(1) {
        int a = 2;
        printf("scope2[a=%d]\n", a);
        if(1) {
            int a = 3;
            printf("scope3[a=%d]\n", a);
        }
    }
    printf("\n");
    return 0;
}


int expr() {
    printf("=== Expr Test ===\n");
    printf("1 * 2 + 3 - (2 * (1 + 2)) = %d. (should be -1)\n\n", 1 * 2 + 3 - (2 *
(1 + 2)));
    return 0;
}

/* compute a*n using add */
int mul_for(int a, int n) {
    int i, r = 0;
    for(i = 0; i < n; i++)
        r = r + a;
    return r;
}
```

```c
int mul_while(int a, int n) {
    int i = 0, r = 0;
    while(1) {
        if(i >= n) break;
        r = r + a;
        i++;
        continue;
        r = 1234; /* used to test whether continue works or not */
    }
    return r;
}

int loop() {
    printf("=== Loop Test ===\n");
    printf("4 * 5: \n");
    printf("    mul_for(4, 5) = %d\n", mul_for(4, 5));
    printf("  mul_while(4, 5) = %d\n", mul_while(4, 5));
    printf("          4 * 5  = %d\n", 4 * 5);
    printf("\n");
    return 0;
}

int fac(int n) {
    if(n == 1) return 1;
    else return mul_for(fac(n - 1), n);
}

int recursive() {
    printf("=== Recursive Test ===\n");
    printf("10! = %d. (should be %d)\n\n", fac(10), 3628800);
    return 0;
}

int main() {
    nested();
    expr();
    loop();
    recursive();
    return 0;
}
```

运行结果截图

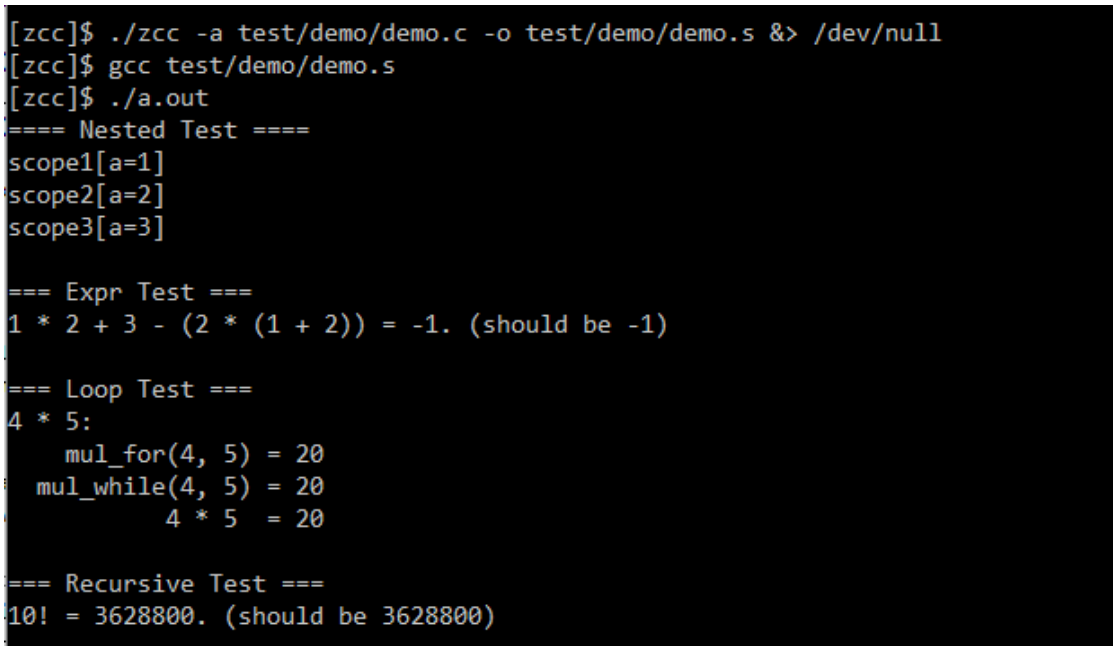


图 5. 测试程序运行结果

## 5. 心得体会

编译器的编写是一次真正的历练，这一次能够成功完成 ZCC 编译器，也让我终于解开一年来的心结。我曾经无比纠结，如此多的生成式，几十上百个非终结符，first/follow 集的计算并不是容易之事，把它写成代码，就更让人头痛。然而此次我并未计算哪怕一个 first/follow 集。在完成了 parse 模块之后，我更发现，parse 是编译器中相当简单的一部分。这其中的转变究竟为何，我也难说清楚。但是我认为，无论是龙书虎书还是陈火旺，都太过于重理论而轻实际。这几本经典教材，曾让我感觉编译器就是计算机科学的圣地，是软件工程的顶端和集大成者。这样的心态让我想得多，做得少。然而实际写下来，发现完全不是这么回事。编译器的实现是逐步的、渐进的，而不是一气呵成的。看着自己的代码能够完成越来越多的功能，真的是一件很开心的事。另外，我的代码全程附带 git log。欢迎查看我的编译器之路。