实验二、编译器认知实验

> *学号：1120180207*
>
> *姓名：唐小娟*
>
> *班级：07111801*

1. 实验目的

本实验的目的是了解工业界常用的编译器 `GCC` 和 `LLVM`，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容，了解编译器的优化效果，为编译器的学习和构造奠定基础。

2. 实验内容

GCC:

• 查看编译器的版本

• 使用编译器编译单个文件

• 使用编译器编译链接多个文件

• 查看预处理结果：gcc -E hello.c -o hello.i

• 查看语法分析树：gcc -fdump-tree-all hello.c

• 查看中间代码生成结果：Code generation result: gcc -fdump-rtl-all hello.c

• 查看生成的目标代码（汇编代码）：gcc –S hello.c –o hello.s

LLVM:

• 查看编译器的版本

• 使用编译器编译单个文件

• 使用编译器编译链接多个文件

• 查看编译流程和阶段：clang -ccc-print-phases test.c -c

• 查看词法分析结果：clang test.c -Xclang -dump-tokens

• 查看词法分析结果2： clang test.c -Xclang -dump-raw-tokens

• 查看语义分析结果：clang test.c -Xclang -ast-dump

• 查看语义分析结果 2：clang test.c -Xclang -ast-view

• 查看编译优化的结果：clang test.c -S -mllvm -print-after-all

• 查看生成的目标代码结果：Target code generation:clang test.c –S

3. 实验环境

名称	信息
操作系统版本	Ubuntu 20.04.1 LTS
Linux内核版本	5.4.0-42-generic
GCC版本	9.3.0
Clang版本	10.0.0-4ubuntu1
LLVM版本	10.0.0

4. 实验过程

4.1 GCC

4.1.1 查看编译器版本

执行命令：gcc -version

```
txx@txx-virtual-machine:~/Desktop$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

由上图可知GCC编译器的版本为：（Ubuntu 9.3.0-17ubuntu1~20.04）9.3.0

4.1.2 使用编译器编译单个文件

编译文件hello.c，hello.c代码内容如下：

```c
#include<stdio.h>
int main()
{
    printf("hello world!\n");
    return 0;
}
```

执行命令：gcc hello.c -o hello

```
txx@txx-virtual-machine:~/studycodes$ gcc hello.c -o hello
txx@txx-virtual-machine:~/studycodes$ ./hello
hello world!
```

编译hello.c后得到可执行文件hello，执行hello文件输出“hello world!"

4.1.3 使用编译器编译链接多个文件

此实验中，我创建了三个文件，分别是myfunc.h、myfunc.c、main.c，具体如下：

```c
//myfunc.h：声明myfunc函数
#include<stdio.h>
void myfunc();

//myfunc,c：定义了myfunc函数
#include"myfunc.h"
void myfunc()
{
    printf("hello,myfunc!\n");
}

//main.c：使用myfunc.c中的myfunc函数
#include"myfunc.h"
int main()
{
    myfunc();
    return 0;
}
```

执行命令：gcc -c myfunc.c //编译myfunc.c文件为myfunc.o目标文件

gcc -c main.c //编译main.c文件为main.o目标文件

gcc myfunc.o main.o -o mytest //链接myfunc.o和main.o为mytest可执行文件。

```
txx@txx-virtual-machine:~/studycodes$ gcc -c myfunc.c
txx@txx-virtual-machine:~/studycodes$ gcc -c main.c
txx@txx-virtual-machine:~/studycodes$ gcc myfunc.o main.o -o mytest
txx@txx-virtual-machine:~/studycodes$ ./mytest
hello,myfunc!
txx@txx-virtual-machine:~/studycodes$
```

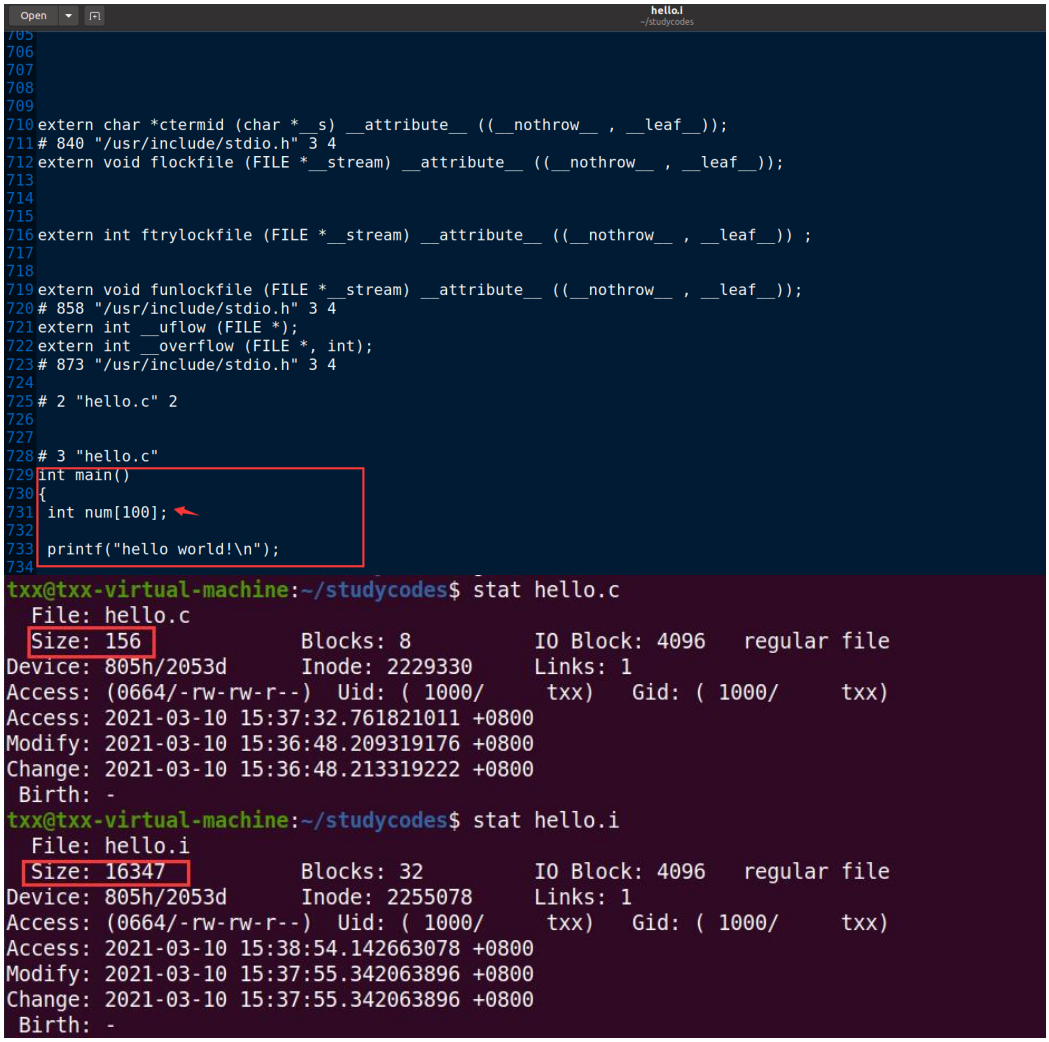
4.1.4 查看预处理结果

为了方便查看预处理的效果，我们在hello.c源文件中，做一些改动，添上宏命令和条件编译。如下：

```c
#include<stdio.h>
#define maxn 100
int main()
{
    int num[maxn];
    #ifdef maxn
    printf("hello world!\n");
    #else
    printf("welcome !\n");
    #endif
    return 0;
}
```

执行命令：gcc -E hello.c -o hello.i，得到预处理后的文件hello.i

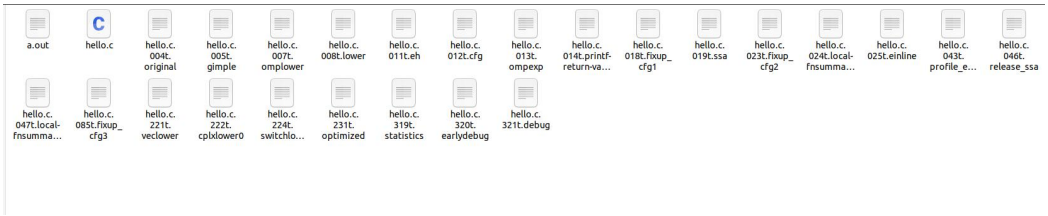
下图是hello.i文件中的部分内容：



我们可以看到，预处理后的文件体积要远远大于源文件，除此之外，根据main.i中的文件内容，文件头（#include<stdio.h>)已经被替换了,宏（#define maxn 100）也在主函数中替换了，条件编译也只保留了运行的部分代码。这样一来，虽然与源文件的内容有一些不一样，但是处理的事务是一样的，为后续的编译节省了不必要的操作。

4.1.5 查看语法分析树：

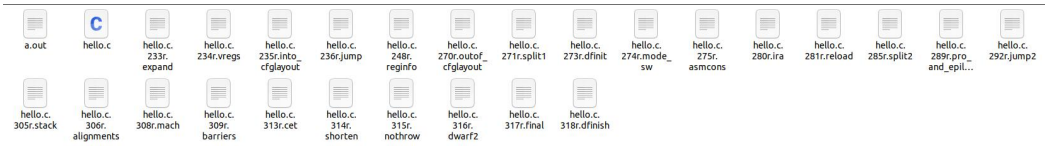
执行命令：gcc -fdump-tree-all hello.c，生成下列文件：



gcc的语法分析树，它的表现形式是一系列文件。GCC编译器的前端将高级语言源码经过词法分析、语法分析生成语法分析树GENERIC。GCC编译器的抽象语法树是源程序的一种中间表示形式，比较直观的表示出源程序的语法结构，并含有源程序结构显示所需要的全部静态信息。GCC 格式的 AST 文件是 GCC 编译源程序时产生的，以文本方式记录源程序抽象语法树的文件，但是由于每种前端语言词法语法分析后形成的GENERIC是异构的，需要转化成一种统一的中间形式进行后续的处理，这种统一的中间表示形式就是GIMPLE。

4.1.6 查看中间代码生成结果

执行命令：gcc -fdump-rtl-all hello.c，生成下列文件：



GCC的中间表示RTL是一种以虚拟寄存器的方式来叙述计算机行为的语言。它接近机器指令，既有指令序列组成的内部形式，又有机器描述和调试信息组成的文本形式。

4.1.7 查看生成的目标代码

执行命令：gcc –S hello.c –o hello.s ，生成如下代码：

```
    .file   "hello.c"
    .text
    .section    .rodata
.LC0:
    .string "hello world!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
```

```
18      subq    $416, %rsp
19      movq    %fs:40, %rax
20      movq    %rax, -8(%rbp)
21      xorl    %eax, %eax
22      leaq    .LC0(%rip), %rdi
23      call    puts@PLT
24      movl    $0, %eax
25      movq    -8(%rbp), %rdx
26      xorq    %fs:40, %rdx
27      je  .L3
28      call    __stack_chk_fail@PLT
29  .L3:
30      leave
31      .cfi_def_cfa 7, 8
32      ret
33      .cfi_endproc
34  .LFE0:
35      .size   main, .-main
36      .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
37      .section    .note.GNU-stack,"",@progbits
38      .section    .note.gnu.property,"a"
39      .align 8
40      .long    1f - 0f
41      .long    4f - 1f
42      .long    5
43  0:
44      .string  "GNU"
45  1:
46      .align 8
47      .long    0xc0000002
48      .long    3f - 2f
49  2:
50      .long    0x3
51  3:
52      .align 8
53  4:
```

根据上面代码，简单分析有：

第1行是GCC留下的文件信息，第2行表示下面是代码段，第3行标识下面是数据段，第5行标识了函数main要用的字符串常量，第7、8行定义了main函数的入口，8行为main入口标号，10-34行是main函数体的内容，后面的内容是GCC留下的某些信息。

4.2 LLVM

4.2.1 查看编译器的版本

执行命令：clang --version

```
txx@txx-virtual-machine:~/Desktop$ clang --version
clang version 10.0.0-4ubuntu1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

由上图可知，clang编译器版本为：10.0.0-4ubuntu1

4.2.2 使用编译器编译单个文件

执行命令：clang hello.c -o hello

```
txx@txx-virtual-machine:~/studycodes/clang$ clang hello.c -o hello
txx@txx-virtual-machine:~/studycodes/clang$ ./hello
hello world!
txx@txx-virtual-machine:~/studycodes/clang$
```

编译hello.c后得到可执行文件hello，执行hello文件输出“hello world!"

4.2.3 使用编译器编译链接多个文件

执行命令：clang -c myfunc.c //编译myfunc.c文件为myfunc.o目标文件

clang -c main.c //编译main.c文件为main.o目标文件

clang myfunc.o main.o -o mytest //链接myfunc.o和main.o为mytest可执行文件。

```
txx@txx-virtual-machine:~/studycodes/clang$ clang -c main.c
txx@txx-virtual-machine:~/studycodes/clang$ clang -c myfunc.c
txx@txx-virtual-machine:~/studycodes/clang$ clang main.o myfunc.o -o mytest
txx@txx-virtual-machine:~/studycodes/clang$ ./mytest
hello,myfunc!
txx@txx-virtual-machine:~/studycodes/clang$
```

4.2.4 查看编译流程和阶段

执行命令：clang -ccc-print-phases main.c -c

```
txx@txx-virtual-machine:~/studycodes/clang$ clang -ccc-print-phases main.c -c
         +- 0: input, "main.c", c
      +- 1: preprocessor, {0}, cpp-output
   +- 2: compiler, {1}, ir
+- 3: backend, {2}, assembler
4: assembler, {3}, object
```

4.2.5 查看词法分析结果

执行命令：clang hello.c -Xclang -dump-tokens 后发现报错

```
(.text+0x24): undefined reference to `main'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

此时应当添上“-c"保证只执行预处理、编译和汇编，而不链接，并且重定向输出到tokens.txt文件中。即：

clang hello.c -Xclang -dump-tokens -c &>tokens.txt

```
typedef 'typedef'       [StartOfLine]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:1>
long 'long'     [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:23>>
unsigned 'unsigned'     [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:28>>
int 'int'       [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:37>>
identifier 'size_t'     [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:23>
semi ';'        Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:29>
typedef 'typedef'       [StartOfLine]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:1>
identifier '__builtin_va_list'  [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:9>
identifier 'va_list'    [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:27>
semi ';'        Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:34>
typedef 'typedef'       [StartOfLine]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:1>
identifier '__builtin_va_list'  [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:9>
identifier '__gnuc_va_list'     [LeadingSpace]  Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:27>
semi ';'        Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:41>
typedef 'typedef'       [StartOfLine]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:1>
unsigned 'unsigned'     [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:9>
char 'char'     [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:18>
identifier '__u_char'   [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:23>
semi ';'        Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:31>
typedef 'typedef'       [StartOfLine]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:1>
unsigned 'unsigned'     [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:9>
short 'short'   [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:18>
int 'int'       [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:24>
identifier '__u_short'  [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:28>
semi ';'        Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:37>
typedef 'typedef'       [StartOfLine]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:1>
unsigned 'unsigned'     [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:9>
int 'int'       [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:18>
identifier '__u_int'    [LeadingSpace]  Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:22>
```

这是其中的一分内容，文件展示了源代码作为长长的字符串利用词法分析后生成一系列tokens。单词被分隔出来，产生相应的属性字。

4.2.6 查看词法分析结果2

同理执行命令：clang test.c -Xclang -dump-raw-tokens -c &>tokens2.txt,结果如下：

```
hash '#'        [StartOfLine]  Loc=<hello.c:1:1>
raw_identifier 'include'                Loc=<hello.c:1:2>
less '<'                Loc=<hello.c:1:9>
raw_identifier 'stdio'          Loc=<hello.c:1:10>
period '.'              Loc=<hello.c:1:15>
raw_identifier 'h'              Loc=<hello.c:1:16>
greater '>'             Loc=<hello.c:1:17>
unknown '
'               Loc=<hello.c:1:18>
raw_identifier 'int'    [StartOfLine]  Loc=<hello.c:3:1>
unknown ' '             Loc=<hello.c:3:4>
raw_identifier 'main'           Loc=<hello.c:3:5>
l_paren '('             Loc=<hello.c:3:9>
r_paren ')'             Loc=<hello.c:3:10>
unknown '
'               Loc=<hello.c:3:11>
l_brace '{'     [StartOfLine]  Loc=<hello.c:4:1>
unknown '
    '           Loc=<hello.c:4:2>
raw_identifier 'int'    [StartOfLine]  Loc=<hello.c:5:2>
unknown ' '             Loc=<hello.c:5:5>
raw_identifier 'b'              Loc=<hello.c:5:6>
unknown ' '             Loc=<hello.c:5:7>
equal '='               Loc=<hello.c:5:8>
unknown ' '             Loc=<hello.c:5:9>
numeric_constant '10'           Loc=<hello.c:5:10>
semi ';'                Loc=<hello.c:5:12>
unknown '
'               Loc=<hello.c:5:13>
raw_identifier 'int'    [StartOfLine]  Loc=<hello.c:6:2>
```

我们发现该命令生成的词法分析结果中的单词只有我们源代码的部分，这个和上述的词法分析结果有些许不一样。

4.2.7 查看语义分析结果

为了更好的理解语义分析结果，我在源代码添上int a = b + c。

执行命令：clang hello.c -Xclang -ast-dump -c &>ast.txt。

```c
#include<stdio.h>

int main()
{
        int b = 10;
        int c = 20;
        int a = b+c;

        return 0;
}
```

```
`-FunctionDecl 0x1e99f10 <hello.c:3:1, line:10:1> line:3:5 main 'int ()'
  `-CompoundStmt 0x1e9a278 <line:4:1, line:10:1>
    |-DeclStmt 0x1e9a050 <line:5:2, col:12>
    | `-VarDecl 0x1e99fc8 <col:2, col:10> col:6 used b 'int' cinit
    |   `-IntegerLiteral 0x1e9a030 <col:10> 'int' 10
    |-DeclStmt 0x1e9a108 <line:6:2, col:12>
    | `-VarDecl 0x1e9a080 <col:2, col:10> col:6 used c 'int' cinit
    |   `-IntegerLiteral 0x1e9a0e8 <col:10> 'int' 20
    |-DeclStmt 0x1e9a230 <line:7:2, col:13>
    | `-VarDecl 0x1e9a138 <col:2, col:12> col:6 a 'int' cinit
    |   `-BinaryOperator 0x1e9a210 <col:10, col:12> 'int' '+'
    |     |-ImplicitCastExpr 0x1e9a1e0 <col:10> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x1e9a1a0 <col:10> 'int' lvalue Var 0x1e99fc8 'b' 'int'
    |     `-ImplicitCastExpr 0x1e9a1f8 <col:12> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x1e9a1c0 <col:12> 'int' lvalue Var 0x1e9a080 'c' 'int'
    `-ReturnStmt 0x1e9a268 <line:9:2, col:9>
      `-IntegerLiteral 0x1e9a248 <col:9> 'int' 0
```

我们将两个文件进行对比得到，下图就是上图中main函数的语法分析结果，其中

FunctionDecl表示方法定义，也就是main函数；

CompoundStmt表示{}，代表了main函数的开始和结尾；

DeclStmt表示局部变量声明，也就是int b = 10；

VarDel表示变量定义；

BinaryOperator表示运算表达式，也就是代码中的b+c；

ImplicitCastExpr表示隐式类型转换；

总上，可以看到，b+c是一个运算表达式，同时int a = b+c则是一个关于a局部变量的声明。

4.2.8 查看语义分析结果2

执行命令：clang hello.c -Xclang -ast-view -c 后，一直报错，我下载了gv和Graphviz仍然是这样，后来查阅资料还是无法解决，所以该步骤没有完成。

4.2.9 查看编译优化的结果

执行命令：clang hello.c -S -mllvm -print-after-all &>optim.txt，得到如下文件内容（截取部分）：

```
418 bb.0 (%ir-block.0):
419   %0:gr32 = MOV32r0 implicit-def $eflags
420   MOV32mi %stack.0, 1, $noreg, 0, $noreg, 0 :: (store 4 into %ir.1)
421   MOV32mi %stack.1, 1, $noreg, 0, $noreg, 10 :: (store 4 into %ir.2)
422   MOV32mi %stack.2, 1, $noreg, 0, $noreg, 20 :: (store 4 into %ir.3)
423   %5:gr32 = MOV32rm %stack.1, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
424   %4:gr32 = ADD32rm killed %5:gr32(tied-def 0), %stack.2, 1, $noreg, 0, $noreg, implicit-def $eflags :
425   MOV32mr %stack.3, 1, $noreg, 0, $noreg, killed %4:gr32 :: (store 4 into %ir.4)
426   $eax = COPY %0:gr32
427   RETQ implicit $eax
428
429 # End machine code for function main.
430
431 # *** IR Dump After Eliminate PHI nodes for register allocation ***:
432 # Machine code for function main: NoPHIs, TracksLiveness
433 Frame Objects:
434   fi#0: size=4, align=4, at location [SP+8]
435   fi#1: size=4, align=4, at location [SP+8]
436   fi#2: size=4, align=4, at location [SP+8]
437   fi#3: size=4, align=4, at location [SP+8]
438
439 bb.0 (%ir-block.0):
440   %0:gr32 = MOV32r0 implicit-def $eflags
441   MOV32mi %stack.0, 1, $noreg, 0, $noreg, 0 :: (store 4 into %ir.1)
442   MOV32mi %stack.1, 1, $noreg, 0, $noreg, 10 :: (store 4 into %ir.2)
443   MOV32mi %stack.2, 1, $noreg, 0, $noreg, 20 :: (store 4 into %ir.3)
444   %5:gr32 = MOV32rm %stack.1, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
```

4.2.10 查看生成的目标代码结果

执行命令：clang -S hello.c

```
    .text
    .file   "hello.c"
    .globl  main                    # -- Begin function main
    .p2align    4, 0x90
    .type   main,@function
main:                                   # @main
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $432, %rsp              # imm = 0x1B0
    movl    $0, -4(%rbp)
    movabsq $.L.str, %rdi
    movb    $0, %al
    callq   printf
    xorl    %ecx, %ecx
    movl    %eax, -420(%rbp)        # 4-byte Spill
    movl    %ecx, %eax
    addq    $432, %rsp              # imm = 0x1B0
```

```
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc
                                        # -- End function
    .type   .L.str,@object          # @.str
    .section    .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz  "hello world!\n"
    .size   .L.str, 14

    .ident  "clang version 10.0.0-4ubuntu1 "
    .section    ".note.GNU-stack","",@progbits
    .addrsig
    .addrsig_sym printf
```

从上述代码我们可以看到main函数入口，以及L.str是main函数所需要用的字符串常量。

5. GCC和LLVM对比分析

5.1 优化效率

我们利用第一个实验中的矩阵运算来看待GCC和LLVM优化效果上的对比：

```
txx@txx-virtual-machine:~/studycodes/matrix$ gcc -O0 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00305s 0.00290s 0.00307s 0.00333s 0.00287s 0.00324s 0.00340s 0.00281s 0.00311s 0.00291s  | average_time : 0.00307s
matrix size: 500    0.40219s 0.39468s 0.39471s 0.40011s 0.40610s 0.40445s 0.40967s 0.40660s 0.40496s 0.40210s  | average_time : 0.40256s
matrix size: 1000    3.56915s 3.62430s 3.42663s 3.65542s 3.73245s 4.00123s 3.87441s 3.65503s 3.65439s 3.63382s  | average_time : 3.68268s
txx@txx-virtual-machine:~/studycodes/matrix$ gcc -O1 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00056s 0.00054s 0.00047s 0.00054s 0.00047s 0.00047s 0.00047s 0.00049s 0.00047s 0.00061s  | average_time : 0.00051s
matrix size: 500    0.08897s 0.08295s 0.08132s 0.08264s 0.07793s 0.07172s 0.07061s 0.07111s 0.07001s 0.07093s  | average_time : 0.07682s
matrix size: 1000    0.71064s 0.70837s 0.71869s 0.71843s 0.71680s 0.71450s 0.71913s 0.72125s 0.71963s 0.71489s  | average_time : 0.71623s
txx@txx-virtual-machine:~/studycodes/matrix$ gcc -O2 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00055s 0.00062s 0.00045s 0.00046s 0.00045s 0.00046s 0.00045s 0.00048s 0.00045s 0.00061s  | average_time : 0.00050s
matrix size: 500    0.07217s 0.07040s 0.07075s 0.06953s 0.06933s 0.06957s 0.06896s 0.06900s 0.06993s 0.07038s  | average_time : 0.07000s
matrix size: 1000    0.72653s 0.70350s 0.72713s 0.70961s 0.70884s 0.70860s 0.70888s 0.71429s 0.70524s 0.70643s  | average_time : 0.71191s
txx@txx-virtual-machine:~/studycodes/matrix$ gcc -O3 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00036s 0.00028s 0.00043s 0.00028s 0.00029s 0.00029s 0.00027s 0.00027s 0.00028s 0.00028s  | average_time : 0.00030s
matrix size: 500    0.03991s 0.03772s 0.03735s 0.03834s 0.03777s 0.03762s 0.03750s 0.03789s 0.03750s 0.03745s  | average_time : 0.03791s
matrix size: 1000    0.34351s 0.34447s 0.33445s 0.33226s 0.35210s 0.32686s 0.33458s 0.32904s 0.33021s 0.33049s  | average_time : 0.33580s
```

```
txx@txx-virtual-machine:~/studycodes/matrix$ clang -O0 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00217s 0.00197s 0.00204s 0.00202s 0.00211s 0.00260s 0.00223s 0.00234s 0.00201s 0.00216s  | average_time : 0.00217s
matrix size: 500    0.27061s 0.26323s 0.26706s 0.26550s 0.26898s 0.27281s 0.27878s 0.27373s 0.27775s 0.27276s  | average_time : 0.27112s
matrix size: 1000    2.32078s 2.30115s 2.42818s 2.40070s 2.31248s 2.43556s 2.41705s 2.71807s 2.85890s 2.72567s  | average_time : 2.49185s
txx@txx-virtual-machine:~/studycodes/matrix$ clang -O1 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00059s 0.00049s 0.00058s 0.00050s 0.00062s 0.00051s 0.00049s 0.00062s 0.00049s 0.00050s  | average_time : 0.00054s
matrix size: 500    0.07499s 0.07242s 0.07105s 0.07129s 0.07119s 0.07117s 0.07178s 0.07115s 0.07156s 0.07105s  | average_time : 0.07176s
matrix size: 1000    0.72263s 0.71695s 0.71822s 0.72386s 0.72139s 0.71747s 0.71703s 0.71393s 0.71802s 0.72014s  | average_time : 0.71896s
txx@txx-virtual-machine:~/studycodes/matrix$ clang -O2 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00058s 0.00050s 0.00049s 0.00061s 0.00049s 0.00050s 0.00049s 0.00050s 0.00049s 0.00059s  | average_time : 0.00052s
matrix size: 500    0.07361s 0.07042s 0.07095s 0.06949s 0.06961s 0.06960s 0.06969s 0.07282s 0.07102s 0.07107s  | average_time : 0.07083s
matrix size: 1000    0.71242s 0.70960s 0.70811s 0.73697s 0.70481s 0.71063s 0.70740s 0.70815s 0.70900s 0.70957s  | average_time : 0.71167s
txx@txx-virtual-machine:~/studycodes/matrix$ clang -O3 dot.c -o dot
txx@txx-virtual-machine:~/studycodes/matrix$ ./dot
matrix size: 10    0.00001s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s 0.00000s  | average_time : 0.00000s
matrix size: 100    0.00058s 0.00058s 0.00048s 0.00060s 0.00049s 0.00049s 0.00049s 0.00049s 0.00049s 0.00061s  | average_time : 0.00053s
matrix size: 500    0.07093s 0.06946s 0.06956s 0.06929s 0.06881s 0.06844s 0.06867s 0.06857s 0.06962s 0.06830s  | average_time : 0.06916s
matrix size: 1000    0.70265s 0.72387s 0.73700s 0.74068s 0.74440s 0.73588s 0.73152s 0.73171s 0.73102s 0.72170s  | average_time : 0.73004s
```

统计有表格：（单位：s）

	10	100	500	1000
GCC-O0	0.00000	0.00307	0.40256	3.68268
GCC-O1	0.00000	0.00051	0.07682	0.71623
GCC-O2	0.00000	0.00050	0.07000	0.71191
GCC-O3	0.00000	0.00030	0.03791	0.33580
clang-O0	0.00000	0.00217	0.27112	2.49185
clang-O1	0.00000	0.00054	0.07176	0.71896
clang-O2	0.00000	0.00052	0.07083	0.71167
clang-O3	0.00000	0.00053	0.06916	0.73004

由上面表格可知：在没有优化的情况下，clang编译运行效率要优于GCC，但是clang的优化效果要略差于GCC。

5.2 语义分析结果

从GCC和clang的语义分析结果来看，clang生成的词法分析结果和语义分析结果易读性更加强，参考clang的语义分析结果，能够更好的理解语法分析树的概念。

5.3 命令行接口

clang和GCC具有高度的兼容性，许多在GCC中适用的命令同样适用于clang。

6. 实验感想

本次实验中，我明白了一份源文件是如何生成可执行文件，而编译器在其中扮演着什么样的角色。从预处理——词法分析——语法分析——语义分析——生成中间代码——代码优化——生成目标代码，贯穿着整个编译原理的过程，也是制作编译器的要点所在，为我学习后续的编译课程打下了概括性认识的基础。

除此之外，在实验过程中，我更加深刻认识了语法树的构成，以及预处理阶段GCC做的主要工作。

尽管在某些过程中，我还不是很能理解某些文件内容，但是我相信，在后续的学习中，我会对编译器有更深的认识和了解。