南　开　大　学

计　算　机　学　院

并行程序设计期末实验报告

预备工作 1：了解编译器 & 简单的 LLVM IR 编程

王浩

学号：2013287

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 2 日

目录

一、 了解编译器

（一） 预处理器

预处理阶段执行的内容就是对 include 后的内容进行代替。执行命令：

```
g++ -E prod prod.cpp
```

可对 prod.cpp 执行预编译。执行命令：

```
g++ -E prod prod.cpp --verbose > /dev/null
```

可以查看预编译阶段的日志，从而可以查看头文件的搜索顺序：

```
#include "..." search starts here:
#include <...> search starts here:
 /usr/include/c++/11
 /usr/include/x86_64-linux-gnu/c++/11
 /usr/include/c++/11/backward
 /usr/lib/gcc/x86_64-linux-gnu/11/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/11/:/usr/lib/gcc/x86_64-linux-gnu/11/:/usr/lib/gcc/x86_64
-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/11/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/11/:/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux
-gnu/:/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/
x86_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/11/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-E' '-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64'
g++: warning: prod: linker input file unused because linking not done
wanghao@wanghao-Lenovo-Legion-Y7000-2020H:~/Coding/编译$
```

另外，通过阅读 g++ 手册：

```
You can specify any number or combination of these options on the command line to
search for header files in several directories.  The lookup order is as follows:

1.  For the quote form of the include directive, the directory of the current file is
    searched first.

2.  For the quote form of the include directive, the directories specified by -iquote
    options are searched in left-to-right order, as they appear on the command line.

3.  Directories specified with -I options are scanned in left-to-right order.

4.  Directories specified with -isystem options are scanned in left-to-right order.

5.  Standard system directories are scanned.

6.  Directories specified with -idirafter options are scanned in left-to-right order.

You can use -I to override a system header file, substituting your own version, since
these directories are searched before the standard system header file directories.
However, you should not use this option to add directories that contain vendor-supplied
system header files; use -isystem for that.
```

可以知道添加头文件的顺序是先添加引号指令的头文件，在添加标准头文件。最后一段也有写，可以通过-I 指令用自己版本的头文件替换系统头文件。

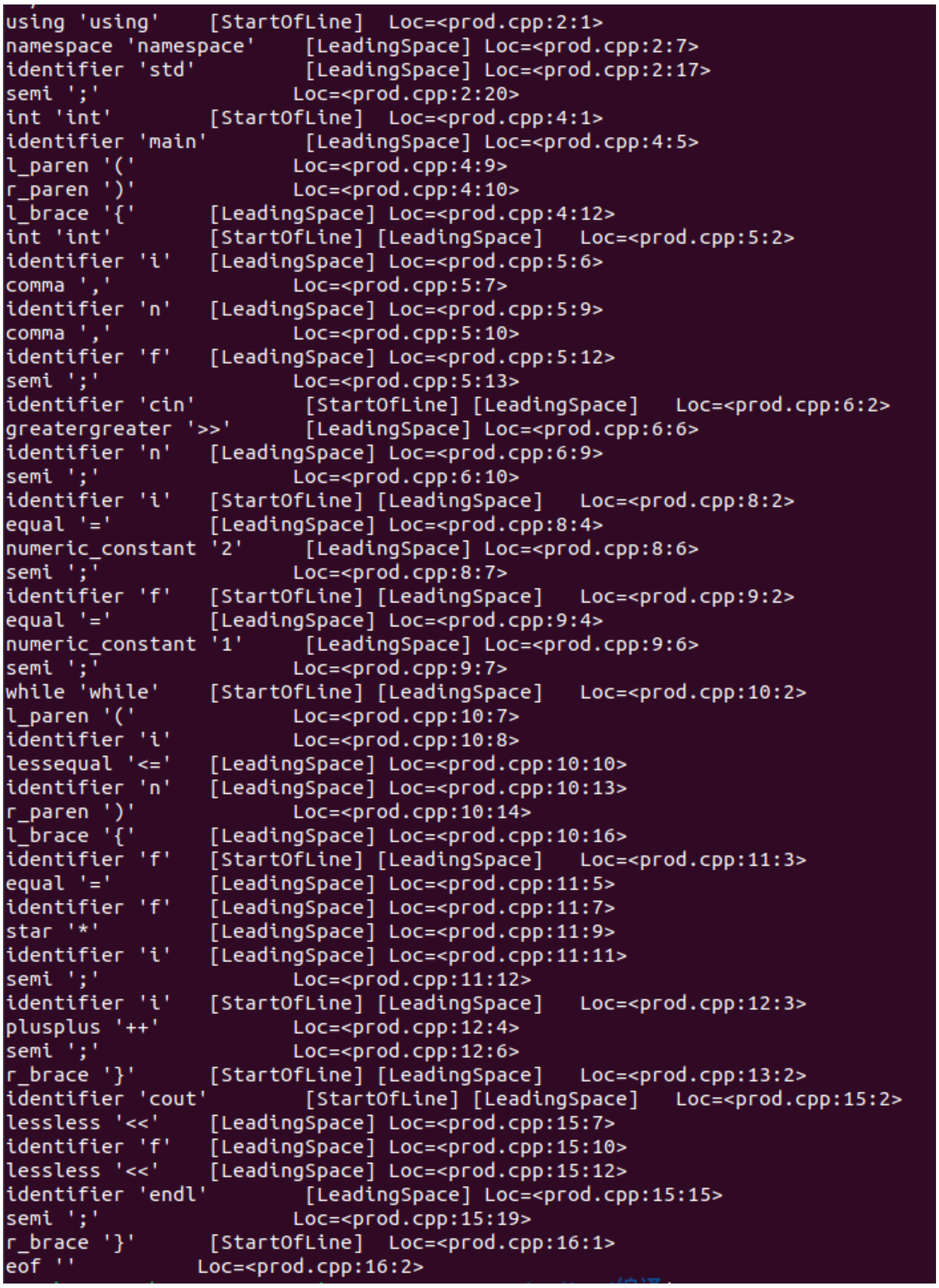
（二） 编译器

1. 词法分析

词法分析阶段，编译器将源程序转换为单词序列。用以下命令对源程序进行词法分析，获得每一个 token 的位置：

```
clang -fsyntax-only -Xclang -dump-tokens prod.cpp
```

结果如下图所示：



```
using 'using'    [StartOfLine]  Loc=<prod.cpp:2:1>
namespace 'namespace'    [LeadingSpace] Loc=<prod.cpp:2:7>
identifier 'std'         [LeadingSpace] Loc=<prod.cpp:2:17>
semi ';'                Loc=<prod.cpp:2:20>
int 'int'        [StartOfLine]  Loc=<prod.cpp:4:1>
identifier 'main'        [LeadingSpace] Loc=<prod.cpp:4:5>
l_paren '('             Loc=<prod.cpp:4:9>
r_paren ')'             Loc=<prod.cpp:4:10>
l_brace '{'      [LeadingSpace] Loc=<prod.cpp:4:12>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:5:2>
identifier 'i'   [LeadingSpace] Loc=<prod.cpp:5:6>
comma ','               Loc=<prod.cpp:5:7>
identifier 'n'   [LeadingSpace] Loc=<prod.cpp:5:9>
comma ','               Loc=<prod.cpp:5:10>
identifier 'f'   [LeadingSpace] Loc=<prod.cpp:5:12>
semi ';'                Loc=<prod.cpp:5:13>
identifier 'cin'         [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:6:2>
greatergreater '>>'      [LeadingSpace] Loc=<prod.cpp:6:6>
identifier 'n'   [LeadingSpace] Loc=<prod.cpp:6:9>
semi ';'                Loc=<prod.cpp:6:10>
identifier 'i'   [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:8:2>
equal '='        [LeadingSpace] Loc=<prod.cpp:8:4>
numeric_constant '2'     [LeadingSpace] Loc=<prod.cpp:8:6>
semi ';'                Loc=<prod.cpp:8:7>
identifier 'f'   [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:9:2>
equal '='        [LeadingSpace] Loc=<prod.cpp:9:4>
numeric_constant '1'     [LeadingSpace] Loc=<prod.cpp:9:6>
semi ';'                Loc=<prod.cpp:9:7>
while 'while'    [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:10:2>
l_paren '('             Loc=<prod.cpp:10:7>
identifier 'i'          Loc=<prod.cpp:10:8>
lessequal '<='   [LeadingSpace] Loc=<prod.cpp:10:10>
identifier 'n'   [LeadingSpace] Loc=<prod.cpp:10:13>
r_paren ')'             Loc=<prod.cpp:10:14>
l_brace '{'      [LeadingSpace] Loc=<prod.cpp:10:16>
identifier 'f'   [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:11:3>
equal '='        [LeadingSpace] Loc=<prod.cpp:11:5>
identifier 'f'   [LeadingSpace] Loc=<prod.cpp:11:7>
star '*'         [LeadingSpace] Loc=<prod.cpp:11:9>
identifier 'i'   [LeadingSpace] Loc=<prod.cpp:11:11>
semi ';'                Loc=<prod.cpp:11:12>
identifier 'i'   [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:12:3>
plusplus '++'           Loc=<prod.cpp:12:4>
semi ';'                Loc=<prod.cpp:12:6>
r_brace '}'      [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:13:2>
identifier 'cout'        [StartOfLine] [LeadingSpace]   Loc=<prod.cpp:15:2>
lessless '<<'    [LeadingSpace] Loc=<prod.cpp:15:7>
identifier 'f'   [LeadingSpace] Loc=<prod.cpp:15:10>
lessless '<<'    [LeadingSpace] Loc=<prod.cpp:15:12>
identifier 'endl'        [LeadingSpace] Loc=<prod.cpp:15:15>
semi ';'                Loc=<prod.cpp:15:19>
r_brace '}'      [StartOfLine]  Loc=<prod.cpp:16:1>
eof ''          Loc=<prod.cpp:16:2>
```

2. 语法分析

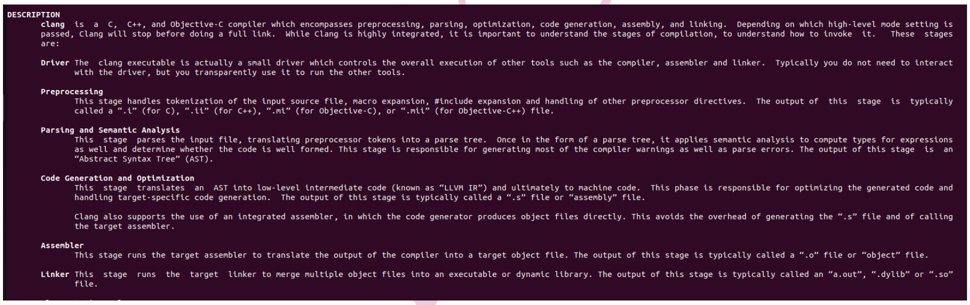
语法分析阶段，编译器将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree，即 AST）。LLVM 可以通过如下命令获得相应的 AST：

```
clang -fsyntax-only -Xclang -ast-dump prod.cpp
```

得到的语法树如下图所示:

3. 语义分析

在语义分析阶段中, 编译器使用语法树和符号表中信息来检查源程序是否与语言定义语义一致, 进行类型检查等。但是大多数编译器并没有把词法分析, 语法分析, 语义分析严格按阶段进行。clang 的-ast-dump 把语义信息也一起输出了。



```
DESCRIPTION
       clang  is  a  C,  C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking.  Depending on which high-level mode setting is
       passed, Clang will stop before doing a full link.  While Clang is highly integrated, it is important to understand the stages of compilation, to understand how to invoke  it.   These  stages
       are:

       Driver The  clang executable is actually a small driver which controls the overall execution of other tools such as the compiler, assembler and linker.  Typically you do not need to interact
              with the driver, but you transparently use it to run the other tools.

       Preprocessing
              This stage handles tokenization of the input source file, macro expansion, #include expansion and handling of other preprocessor directives.  The output of  this  stage  is  typically
              called a “.i” (for C), “.ii” (for C++), “.mi” (for Objective-C), or “.mii” (for Objective-C++) file.

       Parsing and Semantic Analysis
              This  stage  parses the input file, translating preprocessor tokens into a parse tree.  Once in the form of a parse tree, it applies semantic analysis to compute types for expressions
              as well and determine whether the code is well formed. This stage is responsible for generating most of the compiler warnings as well as parse errors. The output of this stage  is  an
              “Abstract Syntax Tree” (AST).

       Code Generation and Optimization
              This  stage  translates  an  AST into low-level intermediate code (known as “LLVM IR”) and ultimately to machine code.  This phase is responsible for optimizing the generated code and
              handling target-specific code generation.  The output of this stage is typically called a “.s” file or “assembly” file.

              Clang also supports the use of an integrated assembler, in which the code generator produces object files directly. This avoids the overhead of generating the “.s” file and of calling
              the target assembler.

       Assembler
              This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a “.o” file or “object” file.

       Linker This  stage  runs  the  target  linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an “a.out”, “.dylib” or “.so”
              file.
```

读 clang 的文档可知, 语法分析 (parsing analysis) 和语义分析 (semantic analysis) 是同时进行的。

4. 中间代码生成

中间代码 (也称中间表示, IR) 是一种编译器定义的, 面向编译场景的指令集。用如下命令生成中间代码:

```
clang -S -emit-llvm prod.cpp
```

5. 代码优化

```
llc -print-before-all -print-after-all a.ll > a.log 2>&1
```

6. 目标代码生成

在目标代码生成阶段，编译器将中间代码翻译成目标指令集：

- 将中间代码的变量映射到寄存器/内存
- 将中间代码的操作映射到指令
- 同时进行目标指令集相关的优化

```
llc prod.ll -o prod.S # LLVM 生成目标代码
```

（三） 汇编器

通过阅读文档，可以知道一个编译器的后端会执行以下步骤：

1. 指令选择：将单独的指令映射到指令集架构中的指令
2. 寄存器分配：为向量分配寄存器；将虚拟寄存器连接到一个（或几个）物理寄存器
3. 指令调度
4. 指令编码

使用以下命令对程序汇编，翻译成二进制目标文件，再反汇编阅读：

```
clang -c prod.cpp # 得到二进制文件prod.o
objdump -d prod.o # 反汇编，得到汇编语言
```

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   f:   48 8b 3d 00 00 00 00    mov    0x0(%rip),%rdi        # 16 <main+0x16>
  16:   48 8d 75 f4             lea    -0xc(%rbp),%rsi
  1a:   e8 00 00 00 00          call   1f <main+0x1f>
  1f:   c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
  26:   c7 45 f0 01 00 00 00    movl   $0x1,-0x10(%rbp)
  2d:   8b 45 f8                mov    -0x8(%rbp),%eax
  30:   3b 45 f4                cmp    -0xc(%rbp),%eax
  33:   0f 8f 18 00 00 00       jg     51 <main+0x51>
  39:   8b 45 f0                mov    -0x10(%rbp),%eax
  3c:   0f af 45 f8             imul   -0x8(%rbp),%eax
  40:   89 45 f0                mov    %eax,-0x10(%rbp)
  43:   8b 45 f8                mov    -0x8(%rbp),%eax
  46:   83 c0 01                add    $0x1,%eax
  49:   89 45 f8                mov    %eax,-0x8(%rbp)
  4c:   e9 dc ff ff ff          jmp    2d <main+0x2d>
  51:   8b 75 f0                mov    -0x10(%rbp),%esi
  54:   48 8b 3d 00 00 00 00    mov    0x0(%rip),%rdi        # 5b <main+0x5b>
  5b:   e8 00 00 00 00          call   60 <main+0x60>
  60:   48 89 c7                mov    %rax,%rdi
  63:   48 8b 35 00 00 00 00    mov    0x0(%rip),%rsi        # 6a <main+0x6a>
  6a:   e8 00 00 00 00          call   6f <main+0x6f>
  6f:   8b 45 fc                mov    -0x4(%rbp),%eax
  72:   48 83 c4 10             add    $0x10,%rsp
  76:   5d                      pop    %rbp
  77:   c3                      ret

Disassembly of section .text.startup:

0000000000000000 <__cxx_global_var_init>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b <__cxx_global_var_init+0xb>
   b:   e8 00 00 00 00          call   10 <__cxx_global_var_init+0x10>
  10:   48 8b 3d 00 00 00 00    mov    0x0(%rip),%rdi        # 17 <__cxx_global_var_init+0x17>
  17:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 1e <__cxx_global_var_init+0x1e>
  1e:   48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx        # 25 <__cxx_global_var_init+0x25>
  25:   e8 00 00 00 00          call   2a <__cxx_global_var_init+0x2a>
  2a:   5d                      pop    %rbp
  2b:   c3                      ret
  2c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000000030 <_GLOBAL__sub_I_prod.cpp>:
  30:   55                      push   %rbp
  31:   48 89 e5                mov    %rsp,%rbp
  34:   e8 c7 ff ff ff          call   0 <__cxx_global_var_init>
  39:   5d                      pop    %rbp
  3a:   c3                      ret
```

反汇编后得到汇编代码。

（四） 链接器

使用如下命令对 'prod.o'文件进行链接：

```
g++ prod.o -o prod
```

二、 LLVM IR 编程

在本次实验中，需要编写一个包含 SysY 语言特性的 C 语言程序，并写出它的中间代码形式（LLVM IR 语言），因为 LLVM IR 语言的语法是之后的实验过程中要用到的，所以本次实验我们小组并没有采取每个人写一部分的做法，而是各自写了完整的 C 程序和 LLVM IR 程序，以便能更熟练地掌握 LLVM IR 语法。接下来呈现我的 C 程序和 LLVM IR 程序：

计算 Fibonacci 数列

```c
void fibonacci(int n) {
        int a, b, i, t;
        a = 0, b = 1, i = 1;
        putint(b);
        putchar(10);

        while (i < n) {
                t = a + b;
                putint(t);
                putchar(10);
                a = b;
                b = t;
                i = i + 1;
        }
}

int main() {
        int n;
        n = getint();

        if (n >= 1)
                fibonacci(n);

        return 0;
}
```


LLVM IR 程序

```llvm
define void @fibonacci(i32 %0) #0 {
        %2 = alloca i32, align 4 ; a
        %3 = alloca i32, align 4 ; b
        %4 = alloca i32, align 4 ; i
        %5 = alloca i32, align 4 ; t
        %6 = alloca i32, align 4 ; n

        store i32 0, i32* %2, align 4
        store i32 1, i32* %3, align 4
        store i32 1, i32* %4, align 4
        store i32 %0, i32* %6, align 4

        %7 = load i32, i32* %3, align 4
```

```
    %8 = call i32 (i32, ...) bitcast (i32 (...)* @putint to i32 (i32, ...)*)(i32 noundef %7)
    %9 = call i32 (i32, ...) bitcast (i32 (...)* @putchar to i32 (i32, ...)*)(i32 noundef 10)
    br label %10

10:
    %11 = load i32, i32* %4, align 4
    %12 = load i32, i32* %6, align 4
    %13 = icmp slt i32 %11, %12
    br i1 %13, label %14, label %23

14:
    %15 = load i32, i32* %2, align 4 ; a
    %16 = load i32, i32* %3, align 4 ; b
    %17 = add nsw i32 %15, %16
    store i32 %17, i32* %5, align 4
    %18 = load i32, i32* %5 ; t, align 4 ; t
    %19 = call i32 (i32, ...) bitcast (i32 (...)* @putint to i32 (i32, ...)*)(i32 %18)
    %20 = call i32 (i32, ...) bitcast (i32 (...)* @putchar to i32 (i32, ...)*)(i32 noundef 10)
    store i32 %16, i32* %2, align 4
    store i32 %17, i32* %3, align 4
    %21 = load i32, i32* %4, align 4 ; i
    %22 = add nsw i32 1, %21 ; i = i + 1
    store i32 %22, i32* %4, align 4 ; i
    br label %10

23:
    ret void
}

declare i32 @putint(...) #1

declare i32 @putchar(...) #1

define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4 ; n
    store i32 0, i32* %1, align 4
    %3 = call i32 (...) @getint()
    store i32 %3, i32* %2 ; n = getint()
    %4 = load i32, i32* %2, align 4
    %5 = icmp sge i32 %4, 1
    br i1 %5, label %6, label %8

6:
```

```
        %7 = load i32, i32* %2, align 4
        call void @fibonacci(i32 %7)
        br label %8

8:
        ret i32 0
}

declare i32 @getint(...) #1
```

Makefile 文件

```
ir-llvm:
    clang -emit-llvm -S main.c -o main.ll
    clang -emit-llvm -S sylib.c -o sylib.ll

exe:
    clang main.ll sylib.c -o out

.PHONY: clean
clean:
        rm main.ll
```