南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

预备工作

刘迪乘　袁田

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 9 月 25 日

目录

一、 了解你的编译器

(一) 问题描述

作业本小节的主要任务是以 GCC，LLVM 等为研究对象，深入地探究语言处理系统的完整工作流程:

1. 完整的编译过程都有什么?
2. 预处理器都做了什么?
3. 编译器都做了什么?
4. 汇编器都做了什么?
5. 链接器都做了什么?

具体而言，我们通过一个简单的 C(C++) 源程序为例，调整编译器的程序选项获得**各阶段的输出**，研究他们与源程序的关系。

我们选取 C++ 中计算阶乘的代码作为本实验的源程序。

阶乘计算算法

```
#include<iostream>
#define NUM 2
int main()
{
  int i,n,f;
  //cin>>n;
  n = NUM;
  i = 2;
  f = 1;
  while(i <= n )
  {
      f = f * i;
      i = i + 1;
  }
  cout << f << endl;
}
```


(二) 刘迪乘 part

1. 编译流程综述

如图1所示，一个 c++ 语言的程序，从源代码到可执行文件，需要经历预处理，编译，汇编，链接四个主要阶段。

2. 预处理阶段

预处理阶段主要由预处理负责。在这个阶段，源程序被处理为一个**纯净的，可以交给编译器语法分析的源文件**。这个阶段的主要工作有:

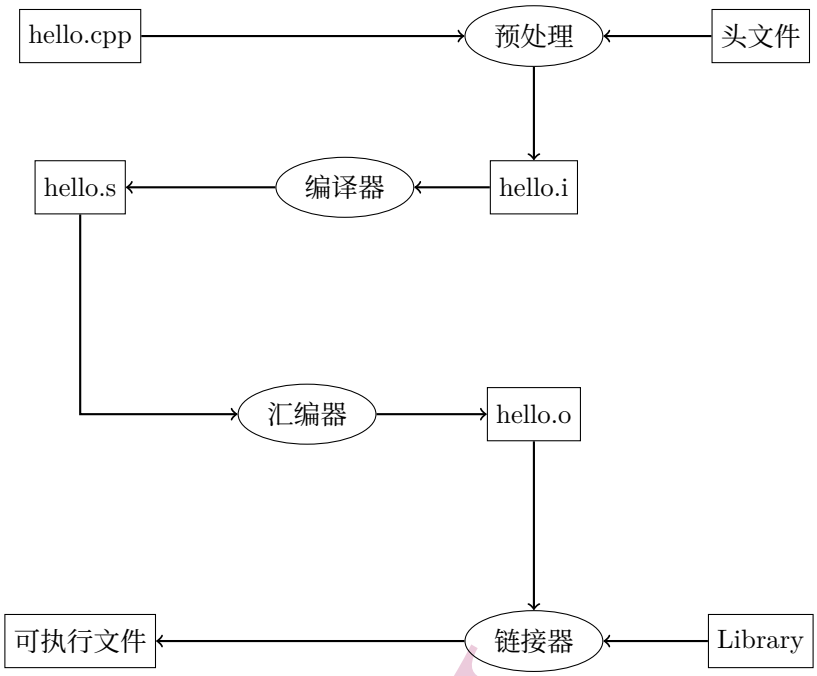


图 1: 编译流程图

- 删除所有注释
- 宏扩展
- 文件包含

宏是使用#define指令定义的一些常量值或表达式。宏调用会导致宏扩展。预处理器创建一个中间文件，其中一些预先编写的汇编级指令替换定义的表达式或常量 (基本上是匹配的标记)。为了区分原始指令和宏扩展产生的程序集指令，在每个宏展开语句中添加了一个”+” 号。

C 语言中的文件包含是在预处理期间将另一个包含一些预写代码的文件添加到当前的程序中它是使用#include指令完成的。在预处理期间包含文件会导致在源代码添加文件名的全部内容，从而替换#include指令，从而创建新的中间文件。

通过编译指令gcc jc.cpp -E -o jc.i，我们可以看到仅仅经过预处理后得到的预处理文件。这个文件的长度远大于源文件，这是因为预处理进行了文件包含的处理，将头文件进行了替换。而主函数部分经过预处理后的代码如下:

阶乘计算算法

```
1  int main()
2  {
3    int i,n,f;
4
5    n = 2;
6    i = 2;
7    f = 1;
8    while(i <= n)
9    {
10       f = f * i;
11       i = i + 1;
```

```
12      }
13      std::cout << f << std::endl;
14    }
```

与源程序相比，经过预处理后的代码进行了注释删除 (cin >> n), 宏扩展 (n = num→ n=2).

此外预处理器还会在预处理文件开头插入**行控制指令**。通过这些指令，在源程序出现错误的情况下，编译器能知道错误在程序的第几行，而不是在预处理展开后的巨大.i 文件里的某个行号。并且编译器可以借此在.i 文件里知道哪些代码来自源代码，哪些来自于头文件。

3. 编译阶段

编译阶段将预处理文件处理为汇编语言程序代码，主要分为六个阶段:

1. 词法分析：将源程序转化为单词序列，我们通过 LLVM 展示，使用指令

```
1       clang -E -Xclang -dump-tokens jc.cpp
```

查看词法分析后的 token 序列。

```
int 'int'       [StartOfLine]  Loc=<jc.cpp:4:1>
identifier 'main'       [LeadingSpace] Loc=<jc.cpp:4:5>
l_paren '('             Loc=<jc.cpp:4:9>
r_paren ')'             Loc=<jc.cpp:4:10>
l_brace '{'     [StartOfLine]  Loc=<jc.cpp:5:1>
int 'int'       [StartOfLine] [LeadingSpace]   Loc=<jc.cpp:6:5>
identifier 'i'  [LeadingSpace] Loc=<jc.cpp:6:9>
comma ','               Loc=<jc.cpp:6:10>
identifier 'n'          Loc=<jc.cpp:6:11>
comma ','               Loc=<jc.cpp:6:12>
identifier 'f'          Loc=<jc.cpp:6:13>
semi ';'                Loc=<jc.cpp:6:14>
identifier 'n'  [StartOfLine] [LeadingSpace]   Loc=<jc.cpp:8:5>
equal '='       [LeadingSpace] Loc=<jc.cpp:8:7>
numeric_constant '2'    [LeadingSpace] Loc=<jc.cpp:8:9 <Spelling=jc.cpp:3:13>>
semi ';'                Loc=<jc.cpp:8:12>
identifier 'i'  [StartOfLine] [LeadingSpace]   Loc=<jc.cpp:9:5>
equal '='       [LeadingSpace] Loc=<jc.cpp:9:7>
numeric_constant '2'    [LeadingSpace] Loc=<jc.cpp:9:9>
```

图 2: 词法分析

根据图2可以看到，此时源程序被解析为了单词序列，序列中的每个元素是最小不可拆分的 token，每个 Token 的表示格式为”类型 值 位置”

2. 语法分析/语义分析，用词法分析生成的词法单元来构建抽象语法树，我们可以通过指令

```
1       clang -E -Xclang -ast-dump jc.cpp
```

获取对应的 AST

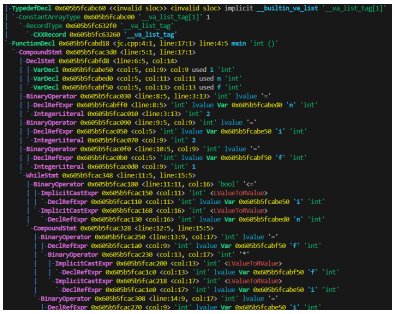


图 3: 语法分析

根据图3可以看到，此时源程序被解析为了 AST，AST 的每个节点通常包含了节点类型，值或名字，子节点列表，位置信息。与此同时，编译器也会进行语义分析，利用语法树和符号表中信息来检查源程序是否和语言定义语义一致，进行类型检查等。

3. 中间代码生成：完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。

   可以通过指令`-fdump-tree-all-graph`和`-fdump-rtl-all-graph`两个 gcc flag 获得中间代码生成的多阶段输出。在此基础上，可以使用 graphviz 工具对这两个阶段的输出进行可视化，从而得到控制流程图 (CFG) 以及编译过程中 CFG 的变化。

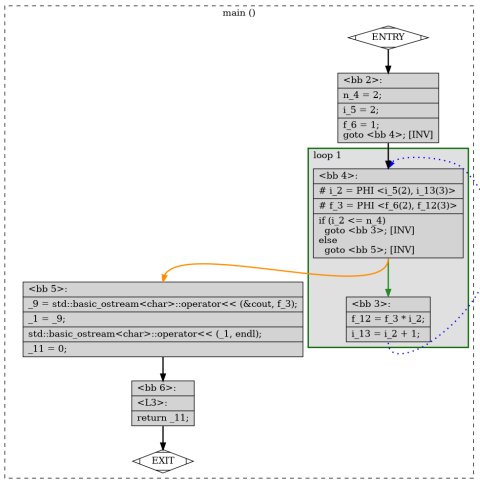


图 4: tree 阶段 CFG 图

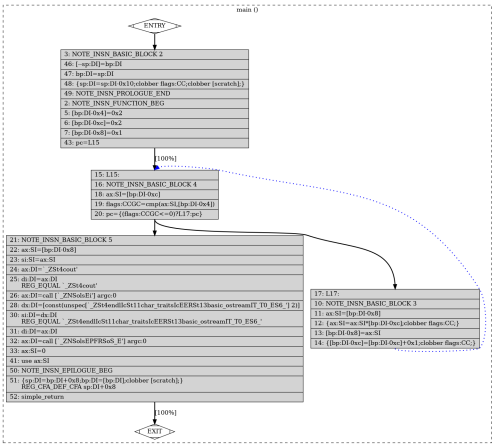


图 5: rtl 阶段 CFG 图

可以看到虽然同处于中间代码生成阶段，但二者的控制流程图存在一定差异，这个差异主要体现在 tree 阶段是编译的高层中间表示，更接近源代码的 AST，保留了源代码的语义，便于进行各种高级优化；而 rtl 阶段则是编译的底层中间表示更加接近汇编，便于寄存器分配和最终代码生成。

此外我们还可以通过指令

```
clang -S -emit-llvm main.c
```

生成 LLVM IR

4. 代码优化：进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。代码优化是通过一系列的 Pass(编译过程中的” 编”) 来完成的。LLVM 官网对所有 pass 的分类中，一共分为三种：Analysis Passes、Transform Passes 和 UtilityPasses。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

   可以通过指令

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

生成每个 pass 后生成的 LLVM IR。这个阶段干的事情简单来说就是优化代码的效率，分为早期优化 (指令级别的 peephole 等)，中级优化 (消除冗余计算等)，循环优化 (循环展开等)，后期优化 (函数内联等) 等。

5. 代码生成：以中间表示形式作为输入，将其映射到目标语言.

   例如我们想要生成 risc-v 格式目标代码，可以采用指令

```
riscv64-unknown-elf-gcc main.c -S -o main.S
```

   如图6所示，编译器生成了 riscv 格式的目标代码，它将作为汇编器的输入，最终被转化为机器码

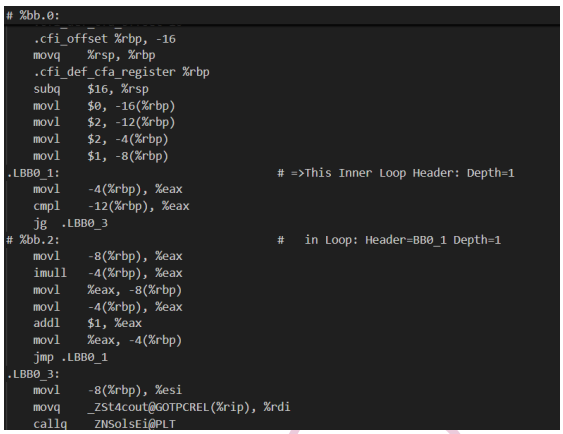


图 6: RISCV 代码生成

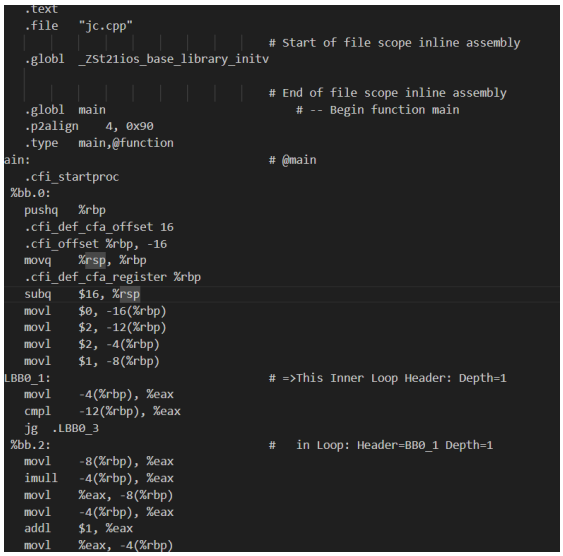
4. 汇编阶段

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”，这一步主要负责把编译器生成的汇编语言程序转化为机器码.

它的主要工作可以分为以下几个步骤：

1. 指令翻译：将汇编源文件 (.s) 中的汇编指令，逐条翻译成对应的机器指令 (即二进制操作码)。
2. 符号处理：汇编代码中可能包含符号，因此汇编器需要维护一个符号表，记录每个符号对应的地址或偏移。对于当前文件可以确定的符号，汇编器会直接替换为具体地址；对于外部符号 (如调用库函数等) 会在目标文件中生成未解析符号信息，留给后续的链接器来处理
3. 伪指令展开: 汇编源文件中存在一些高级汇编伪指令，这并不是硬件真正支持的指令，汇编器会把他们展开成若干条真正的机器指令
4. 数据分布：处理`.data`,`.bss`,`.rodata`等段中的数据，分配合适的存储位置
5. 目标文件生成：最终输出目标文件 (.o/.obj), 里面包含：机器指令的二进制编码，符号表，重定位信息，段表信息等

具体而言我们在上一步中得到了源代码的汇编源代码 (.s)



```
  .text
  .file   "jc.cpp"
                                        # Start of file scope inline assembly
  .globl  _ZSt21ios_base_library_initv

                                        # End of file scope inline assembly
  .globl  main                              # -- Begin function main
  .p2align    4, 0x90
  .type   main,@function
ain:                                    # @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    $16, %rsp
  movl    $0, -16(%rbp)
  movl    $2, -12(%rbp)
  movl    $2, -4(%rbp)
  movl    $1, -8(%rbp)
LBB0_1:                                 # =>This Inner Loop Header: Depth=1
  movl    -4(%rbp), %eax
  cmpl    -12(%rbp), %eax
  jg  .LBB0_3
 %bb.2:                                 #   in Loop: Header=BB0_1 Depth=1
  movl    -8(%rbp), %eax
  imull   -4(%rbp), %eax
  movl    %eax, -8(%rbp)
  movl    -4(%rbp), %eax
  addl    $1, %eax
  movl    %eax, -4(%rbp)
```

图 7: 汇编源代码

如图7所示，其中有多种符号以及标签 (如.global main是全局符号，声明main是全局可见的)，汇编器接下来会把这些符号和标签都记录在符号表里，并生成对应的地址或重定位信息。

同时还可以注意到在源汇编文件中存在指令.size main, .Lfunc_end0-main，这是一个指定函数大小的高级汇编指令，但是在机器指令中并不存在，因此汇编器接下来要将他做一些展开。

另外在汇编源文件中，也可以看到诸如.text之类的代码段信息，汇编器会把他们的地址加入到符号表中.

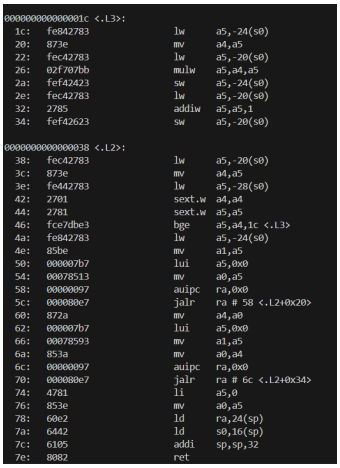
通过指令

```
riscv64-unknown-elf-gcc test.S -c -o test.o
```

可以对汇编源文件执行汇编器的工作。此时得到的是一个二进制文件，我们无法通过文本编辑器读取，因此需要借助反汇编工具，通过指令

```
riscv64-unknown-elf-objdump -d jc.o
```

可以得到对应的反汇编文件，得到二进制机器码对应的指令，部分结果如图8



```
000000000000001c <.L3>:
  1c:   fe842783            lw      a5,-24(s0)
  20:   873e                mv      a4,a5
  22:   fec42783            lw      a5,-20(s0)
  26:   02f707bb            mulw    a5,a4,a5
  2a:   fef42423            sw      a5,-24(s0)
  2e:   fec42783            lw      a5,-20(s0)
  32:   2785                addiw   a5,a5,1
  34:   fef42623            sw      a5,-20(s0)

0000000000000038 <.L2>:
  38:   fec42783            lw      a5,-20(s0)
  3c:   873e                mv      a4,a5
  3e:   fe442783            lw      a5,-28(s0)
  42:   2781                sext.w  a4,a4
  44:   2781                sext.w  a5,a5
  46:   fce7dbe3            bge     a5,a4,1c <.L3>
  4a:   fe842783            lw      a5,-24(s0)
  4e:   85be                mv      a1,a5
  50:   000007b7            lui     a5,0x0
  54:   00078513            mv      a0,a5
  58:   00000097            auipc   ra,0x0
  5c:   000080e7            jalr    ra # 58 <.L2+0x20>
  60:   872a                mv      a4,a0
  62:   000007b7            lui     a5,0x0
  66:   00078593            mv      a1,a5
  6a:   853a                mv      a0,a4
  6c:   00000097            auipc   ra,0x0
  70:   000080e7            jalr    ra # 6c <.L2+0x34>
  74:   4781                li      a5,0
  76:   853e                mv      a0,a5
  78:   60e2                ld      ra,24(sp)
  7a:   6442                ld      s0,16(sp)
  7c:   6105                addi    sp,sp,32
  7e:   8082                ret
```

图 8: 反汇编

5. 链接阶段

由汇编程序生成的目标文件无法直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接在一起，最终形成真正可以在机器上运行的可执行文件.

同样对生成的可执行文件进行反汇编，一个最直观的感觉是这个反汇编得到的文件长度显著大于之前我们得到的文件，这主要是因为链接器将不同的库文件进行了链接。另外就是所有的符号都已解析或通过 PLT 重定向，跳转地址都是最终绝对/相对地址。段信息也进行了合并。

(三) 袁田 part

1. 完整的编译过程都有什么?

在程序从源代码变为可执行文件的过程中，编译系统一般会经历预处理，编译，汇编，链接这四个主要阶段。从代码的角度来看，则是从最初源代码，经过预处理阶段变为调整后的代码，经过编译器生成汇编语言代码，经过汇编器生成可调整的机器语言代码，经过链接器或加载器生成最终可执行程序。以下将简单概述各个阶段的实际功能。

首先是预处理阶段，这一步由预处理器完成，主要负责处理源代码中的宏定义（define），文件包含（include），条件编译指令（ifdef 等）等操作。经过预处理后，代码会被展开成一个纯净的源代码文件，去掉了注释并替换了宏，包含的头文件内容也会被插入进来。

接下来是编译阶段，编译器会将预处理过的源代码翻译成对应的汇编代码。在这一过程中，编译器会进行课上讲述的六个阶段：词法分析，语法分析，语义分析，中间代码生成，代码优化，目标代码生成。编译阶段会将预处理调整后的代码，转变成为汇编语言代码，生成汇编语言文件。

然后是汇编阶段，汇编器会把汇编语言文件转换成机器指令，生成目标文件（通常是.o 型文件）。目标文件包含了机器码和一些符号表信息，但此时还不是一个完整的可执行程序，因为里面的函数调用、全局变量可能引用了外部符号。

最后是链接阶段，链接器会把多个目标文件以及需要用到的库文件（如标准库、第三方库）整合在一起，解决符号引用问题，把函数调用和变量引用与它们的真实地址对应起来。链接完成后，生成一个完整的可执行文件（在 Linux 系统下常见为.out 型文件，在 Windows 系统下则通常为 exe 文件）。

2. 预处理器都做了什么?

我们知道，在预处理阶段，编译器会先调用预处理器对源代码进行一系列文本替换和整理工作。最常见的任务如下，在这一阶段后，源文件会被展开为一个“纯净”的中间文件，方便后续编译器进行下一步处理。

- 处理 include 指令，把被包含的头文件内容直接插入到源文件中
- 处理宏定义（define），将宏名替换为对应的值或代码片段
- 处理条件编译指令（如 ifdef、ifndef、if），根据条件决定代码是否保留
- 删除源代码中的注释，使代码更简洁

本次实验中，我们选取 C++ 中计算阶乘的代码作为本实验的源程序。我们使用指令 **g++ -E test.cpp -o test.i** 对源程序进行预处理。当我们对不添加头文件，没有宏定义的最基础源程序进行预处理后，test.i 文件下的代码如下:

预处理后代码

```
# 0 "test.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "test.cpp"
int main()
{
    int i,n,f;
    cin>>n;
    i=2;
    f=1;
    while(i <= n)
    {
        f=f*i;
        i=i+1;
    }

    cout<<f<<endl;
}
```

我们可以看到，相对于源程序外，在预处理后的.i 文件开头出现了这些以开头的行，这是预处理器插入的**行控制指令 (line markers)**。通过这些行控制指令，在源程序出现错误的情况下，编译器能知道错误在源程序的第几行，而不是在预处理展开后的巨大.i 文件里的某个行号。并且编译器可以借此在.i 文件里知道哪些代码来自源程序，哪些来自 iostream、stdlib.h 等系统头文件。

随后，我们简单修改程序，增加宏定义 define NUM 2，并注释 cin«n，将其改为 n=NUM。对其进行预处理后，可以看到处理后的文件中，直接将 n=NUM 转化为了 n=2，并且将注释部分删除。同样，增加头文件包含后，可以发现文件长度远大于源文件，这就是将代码中的头文件进行了替代导致的结果。

3. 编译器都做了什么?

我们知道，将源程序从人类可读的文本变成计算机可执行的代码，通常会经历一系列连续的阶段，这个过程被称为编译过程。大多数编译器会将这些阶段抽象为前端和后端两大主要部分。其中，前端部分主要负责分析源代码，确保其语法和语义正确，并生成一个与源语言和目标机器都无关的中间表示，这部分主要依赖于源语言的规则（如 C++, Java 的语法）；而后端部分则负责接收前端生成的中间表示，并针对目标机器（如 x86, ARM 架构）进行优化和代码生成。

编译过程分为六个阶段：词法分析，语法分析，语义分析，中间代码生成，代码优化，目标代码生成。接下来，我们将依次获取每个阶段的运行成果并对其进行解释。

1. 词法分析

   词法分析是编译过程的第一个阶段，负责对源代码进行“分词”。其输入是原始的字符序列（包括字母、数字、符号等）。词法分析器会逐个字符地扫描这些源代码，根据预定义的构词规则，识别出一系列有意义的、不可再分的最小单元，称为词法单元。在此过程中，它会忽略空格、制表符、换行符等无关紧要的内容。该阶段的输出是一个规整的 Token 流，为后

续的语法分析做好准备。我们通过指令 **clang -E -Xclang -dump-tokens test.cpp**，获取输出的 token 流。可以看到，词法分析器将源程序分词并输出了 token 流，以 **i=2;** 为例，词法分析器对其进行分词，变为标识符 i，等号 =，数字 2，分号;

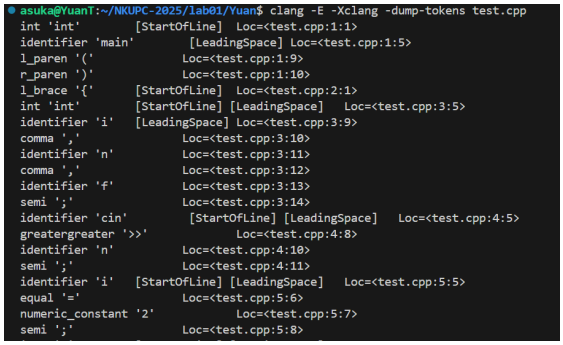


图 9: 词法分析结果

2. 语法分析和语义分析

语法分析阶段将会接收上一阶段产生的 Token 流，并检查这些 Token 的组合是否符合源语言的语法规则。这个过程由语法分析器完成，它使用上下文无关文法来描述编程语言的语法结构。语法分析器会将 Token 流转化为抽象语法树。这棵树清晰地表达了程序的语法结构，其中内部节点代表操作符（如循环、条件、赋值），而叶子节点则代表操作数（如常量、变量）。如果 Token 序列无法根据语法规则构成任何有效的结构，语法分析器将报告语法错误，例如括号不匹配、缺少分号等。

我们首先使用指令 **clang -E -Xclang -ast-dump test.cpp** 获取抽象语法树，为了简化输出，我们注释掉头文件包含，这会导致 cin 和 cout 出现错误，语法树的部分结构如下图:

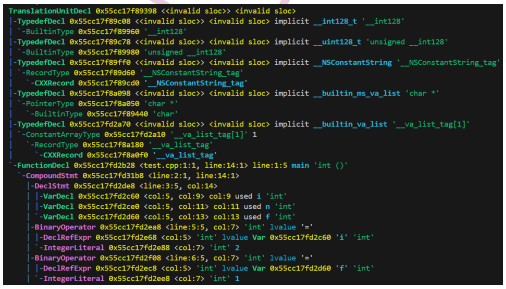
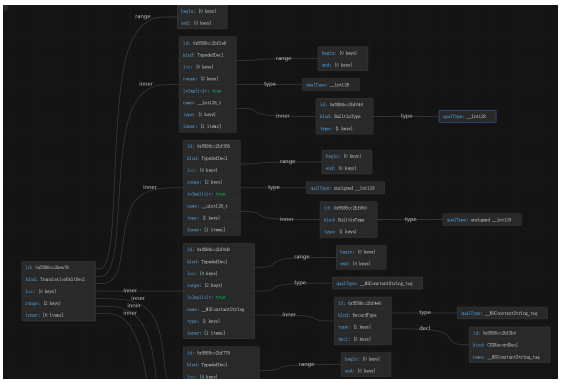


图 10: 语法分析结果

考虑到可视化的问题，我们使用指令 **clang -Xclang -ast-dump=json -fsyntax-only test.c > ast.json**，获取 JSON 格式的 AST，随后使用 jsoncrack 查看树状结构，其中部分分支如下图:



在语法分析结束生成 AST 后，语义分析阶段将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。

3. 中间代码生成和代码优化

在完成前面的工作后，很多编译器会生成一个明确的低级或类机器语言的中间表示。之所以要引入中间代码，是因为引入后可以减轻多种高级程序语言在多种平台下可运行的压力。在实验中我们通过-fdump-tree-all-graph 和-fdump-rtl-all-graph 这两个 gcc flag 获得中间代码生成的多阶段的输出，随后使用 graphviz 对输出的.dot 文件进行可视化。具体使用的指令为 **gcc -O0 -fdump-rtl-all-graph=rtl test.cpp** 和 **dot -Tpng rtl.dot -o rtl.png**。可视化的图形输出如下：

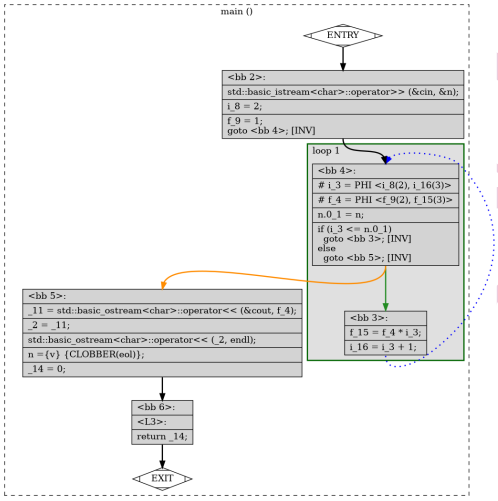


图 11: tree 阶段 CFG 图

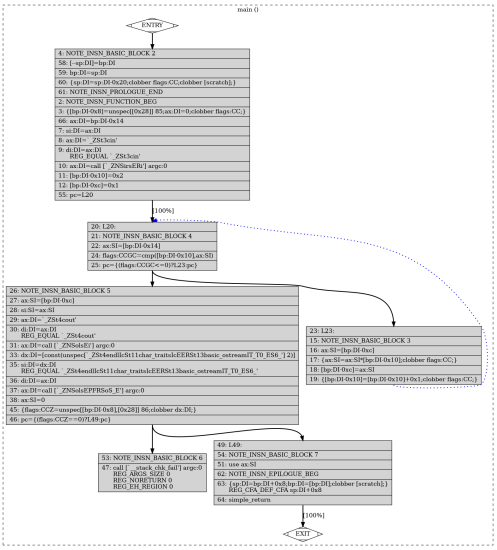


图 12: rtl 阶段 CFG 图

可以看出，tree 阶段将会保留 if/while/for，函数调用等高级语义，相对来说更接近于源程序；而 RTL 则是相对低级的中间表示，更加接近汇编语言，用伪指令加上寄存器操作的形式来表达计算。随后，使用指令 **clang -S -emit-llvm test.cpp** 获取中间代码，保存在.ll 文件中。

随后，代码优化阶段将会基于中间代码进行与机器无关的代码优化步骤，以此改进中间代码，生成更好的目标代码。LLVM 官网将代码优化使用的 pass 分为三种：Analysis Passes、Transform Passes 和 UtilityPasses。其中 Analysis Passes 用于分析或计算某些信息，以便

给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 则会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

我们可以通过指令 **llc -print-before-all -print-after-all test.ll > test.log 2>1** 来观察每一个 pass 前后 IR 代码的状态。我们可以通过寻找类似于 IR Dump Before <pass-name> 和 IR Dump After <pass-name> 这两种标签，找到进行某种 pass 前后的 IR 代码对其比较。例如在 test.log 中的第一个 pass：Pre-ISel Intrinsic Lowering (pre-isel-intrinsic-lowering)，这并非做性能优化的 Pass，而是一个代码转换 pass，它的工作是：在指令选择之前，LLVM 将把一些不适合直接映射到机器指令的 intrinsic(LLVM 提供的一些扩展操作) 替换成等价的、更底层的 IR。

4. 目标代码生成

   该阶段的任务是将优化后的中间表示映射到特定的目标机器指令集上，其输入为经过代码优化后的中间代码，输出则是不同格式的目标代码。本次实验中我们需要将其映射到 risc-v 格式，使用指令 **riscv64-unknown-elf-gcc test.cpp -S -o test.S**，则可以获取编译器最终输出：汇编语言代码，其被保存在.S 文件中，以下是部分代码：

```
    .file   "test.cpp"
    .option nopic
    .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zmmul1p0_zaamo1p0_zalrsc1p0_zca1p0_zcd1p0"
    .attribute unaligned_access, 0
    .attribute stack_align, 16
    .text
    .align  1
    .globl  main
    .type   main, @function
main:
.LFB1957:
    .cfi_startproc
    addi    sp,sp,-32
    .cfi_def_cfa_offset 32
    sd  ra,24(sp)
    sd  s0,16(sp)
    .cfi_offset 1, -8
    .cfi_offset 8, -16
    addi    s0,sp,32
    .cfi_def_cfa 8, 0
    addi    a5,s0,-28
    mv  a1,a5
    lui a5,%hi(_ZSt3cin)
    addi    a0,a5,%lo(_ZSt3cin)
    call    _ZNSirsERi
    li  a5,2
    sw  a5,-20(s0)
    li  a5,1
    sw  a5,-24(s0)
    j   .L2
```

图 13: Risc-V 格式目标代码

4. 汇编器都做了什么?

我们从两个方面来回答这个问题：汇编器的具体功能是什么？汇编器处理的结果是什么?

首先，关于汇编器的具体功能。汇编器的主要功能是：把汇编代码翻译成目标文件中的机器指令和数据，并为链接器准备信息。例如，汇编器会将编译器生成的汇编语言代码进行翻译，将常见的指令 add,move 等翻译成对应的机器指令。另外，汇编器还有包括地址计算，符号处理等功能。

在知道汇编器具体功能后，自然可以发现，汇编器接受来自编译器的汇编语言程序，输出.o 型的目标文件。这个文件并非最终可执行程序，而是二进制的中间文件。其中包括：

1. 机器指令代码段：将汇编中的指令如 add,move 等翻译为对应的机器码
2. 数据段：存放全局变量和静态变量
3. 符号表：记录函数名，全局变量名等信息，用于之后的链接阶段
4. 重定位信息：在程序应用外部函数或变量时，汇编器无法确定对应地址，将在目标文件 (.o 文件) 中留下重定位信息，后面链接阶段可以借此定位函数或变量位置

我们可以使用指令：**riscv64-unknown-elf-gcc test.S -c -o test.o** 获取目标文件，由于二进制机器码无法在文本编辑器中展开，我们需要使用反编译工具进行反编译。首先使用指令

file test.o，可以获取输出：**test.o: ELF 64-bit LSB relocatable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), not stripped**，其中 not stripped 表明符号表并未被删除，相对于 stripped 而言反编译难度较低。

由于架构为 Risc-V，我们使用交叉编译工具链中的 objdump 进行反汇编操作，输入指令 **riscv64-unknown-elf-objdump -d test.o** 可以在终端中获取 test.o 的反汇编信息，部分如下图：

```
test.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <main>:
   0:   1101                    addi    sp,sp,-32
   2:   ec06                    sd      ra,24(sp)
   4:   e822                    sd      s0,16(sp)
   6:   1000                    addi    s0,sp,32
   8:   fe440793                addi    a5,s0,-28
   c:   85be                    mv      a1,a5
   e:   000007b7                lui     a5,0x0
  12:   00078513                mv      a0,a5
  16:   00000097                auipc   ra,0x0
  1a:   000080e7                jalr    ra # 16 <main+0x16>
  1e:   4789                    li      a5,2
  20:   fef42623                sw      a5,-20(s0)
  24:   4785                    li      a5,1
  26:   fef42423                sw      a5,-24(s0)
  2a:   a839                    j       48 <.L2>
```

图 14: 反编译结果部分显示

5. 链接器都做了什么?

由汇编程序生成的目标文件不能够直接执行。在汇编器部分提到，汇编器输出的目标文件中会包含重定位信息。一般来说，大型程序经常会被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。

我们使用指令 **riscv64-unknown-elf-gcc test.o -o test** 获取最终可执行程序，这也是编译阶段的最后一步，获取.exe 文件后，则可进行程序运行 (实际上由于 cpp 文件中包含有 cin 和 cout 等 C++ 标准库符号，而我们使用的工具链 riscv64-unknown-elf-gcc 并不带有完整 c++ 标准库，所以会报 undefined reference，使用 printf 等来代替即可)。

二、 LLVM IR 编程汇编编程

(一) 问题描述

熟悉 LLVM IR 中间语言和汇编代码:

- 设计 SysY 示例程序涵盖编译器要支持的语言特性 (各种数值运算，赋值，条件分支，循环等语句);
- 编写 LLVM IR 程序以及 RISC-V 汇编程序，与 SysY 源程序等价，且能够链接 SysY 运行库，用 LLVM/Clang，汇编器编译成目标程序，验证运行结果是否正确。

我们采用下面用 SysY 语言编写的代码作为研究对象进行 LLVM IR,RISC-V 语言的编写。

```
1  int calculateFactorial(int n) {
2  if (n == 0 || n == 1) {
3      return 1;
```

```
    }

    int factorial = 1;
    int i = 2;
    while (i <= n) {
        factorial = factorial * i;
        i = i + 1;
    }

    return factorial;
}

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

    if (number < 0) {
        putstr("error\n");
    } else {
        int result;
        result = calculateFactorial(number);
        putint(result);
    }

    return 0;
}
```

(二) LLVM IR 编程

根据我们的初始源代码 sysy 文件，我们首先尝试使用三地址码风格写中间代码。三地址码是最常见最直观的一种 IR，这种风格下，每条指令最多有三个操作数，其中控制流用条件跳转 (if goto) 和无条件跳转 (goto) 来实现，函数调用则使用 call 和 return 等来实现。具体代码如下:

三地址码形式的中间代码

```
func calculateFactorial(n):
    if n == 0 goto L1
    if n == 1 goto L1
    goto L2
L1:
   return 1
L2:
    factorial = 1
    i = 2
L3:
    if i <= n goto L4
    goto L5
L4:
    t1 = factorial * i
```

```
    factorial = t1
    t2 = i + 1
    i = t2
    goto L3
L5:
    return factorial


func main():
    number = call getint()
    if number < 0 goto L6
    goto L7
L6:
    call putint(-1)
    goto L8
L7:
    result = calculateFactorial(number)
    call putint(result)
L8:
    return 0
```

随后，按照 LLVM IR 的风格完成中间代码，代码如下：

三地址码形式的中间代码

```
declare i32 @getint()
declare void @putint(i32)

; int calculateFactorial(int n)
;函数定义会以define开头，i32指的是返回类型为int

define i32 @calculateFactorial(i32 %n){
entry:
    %cmp0 = icmp eq i32 %n, 0
    %cmp1 = icmp eq i32 %n, 1
    %or = or i1 %cmp0,%cmp1
    ;将虚拟寄存器cmp0和cmp1的或结果保存于%or中
    br i1 %or,label %ret1,label %loop_entry
    ;此处br为有条件分支，若i1型的%or为真，则跳转到%ret1

ret1:
    ret i32 1

loop_entry:
    ;先分配空间，再将常数1存入
    %factorial = alloca i32, align 4
    store i32 1,i32* %factorial
    %i = alloca i32, align 4
    store i32 2,i32* %i

```

```
    br label %loop_cond
    ;此处的br为无条件分支，跳到loop_cond下

loop_cond:
    %i_val = load i32, i32* %i
    %cmp = icmp sle i32 %i_val, %n
    br i1 %cmp, label %loop_body, label %loop_end
    ;若i<=n，则%cmp为真，跳转至loop_body

loop_body:
    %f_val = load i32,i32* %factorial
    %i_val2 = load i32, i32* %i
    %t1 = mul i32 %f_val, %i_val2
    ;进行乘法运算，输出类型为i32
    store i32 %t1,i32* %factorial

    %t2 = add i32 %i_val2,1
    store i32 %t2,i32* %i
    br label %loop_cond

loop_end:
    %res = load i32,i32* %factorial
    ret i32 %res
}

define i32 @main(){
entry:
    %number = alloca i32, align 4
    %num_val = call i32 @getint()
    store i32 %num_val,i32* %number

    %cmp_m = icmp slt i32 %num_val,0
    br i1 %cmp_m,label %sit_error,label %main_body

sit_error:
    call void @putint(i32 -1)
    br label %main_end

main_body:
    %number_val = load i32,i32* %number
    %result = call i32 @calculateFactorial(i32 %number_val)
    call void @putint(i32 %result)
    br label %main_end

main_end:
    ret i32 0
}
```

完成 LLVM IR 编程后，我们将其编译为可执行文件来查看运行结果。使用指令 **clang -o f f.ll /home/asuka/NKUPC-2025/lib/libsysy_x86.a** 获取可执行文件 f，随后运行该文件进行验证。验证情况如下图，可以看到结果正确。

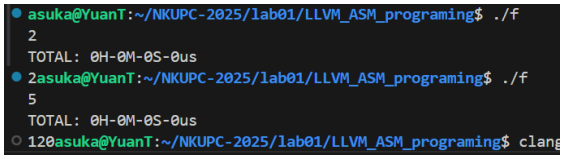


图 15: 结果验证

(三) RISC-V 汇编编程

在本小节中，我们采用 RISC-V 指令编写汇编代码。

```
        .text
    .globl calculateFactorial

calculateFactorial:
    addi sp,sp,-16
    sw ra,12(sp)
    sw s0,8(sp)
    mv s0,a0

    #if (n == 0 || n == 1) return 1;
    li t0,0
    beq s0,t0,.Lreturn1
    li t0,1
    beq s0,t0,.Lreturn1

    #int factorial = 1;
    li t1,1

    #int i = 2
    li t2,2

.Lwhile:
    #while(i <= n)
    bgt t2,s0,.Lwhile_end

    #factorial = factorial * i;
    mul t1,t1,t2

    #i++;
    addi t2,t2,1

    j .Lwhile

.Lwhile_end:
```

```
    mv a0,t1
    j .Lepilogue

.Lreturn1:
    li a0,1

.Lepilogue:
    lw ra,12(sp)
    lw s0,8(sp)
    addi sp,sp,16
    ret

    .globl main
main:
    addi sp,sp,-16
    sw ra,12(sp)
    sw s0,8(sp)

    #number = getint()
    call getint
    mv s0,a0

    #if(number < 0)
    bltz s0,.Lif_negative

    #else result = calculateFactorial(number)
    mv a0,s0
    call calculateFactorial
    call putint
    j .Lend
.Lif_negative:
    li a0, -1
    call putint

.Lend:
    lw ra,12(sp)
    lw s0,8(sp)
    addi sp,sp,16
    li a0,0
    ret
```

然后通过如下指令，编译为可执行文件

```
    riscv64-unknown-elf-gcc main.s -c -o main.o -w
    riscv64-unknown-elf-gcc main.o -o main\
    -L./lib -lsysy_riscv\
    -static -mcmodel=medany\
    -Wl,--no-relax,-Ttext=0x90000000
```

生成可执行文件后，由于电脑架构并非 RISC-V 架构，因此需要通过 qemu 执行，使用指令

```
qemu-riscv64 main
```

经检验，最终结果与源程序输出一致

图 16: RISC-V 汇编编译结果

参考文献