

四、中间代码生成 (10. LLVM IR 简介)

魏恒峰

hfwei@nju.edu.cn

2024 年 04 月 26 日



Chris Lattner @ Homepage



lattner @ GitHub

<https://llvm.org/>

The LLVM Compiler Infrastructure



P:
W
S
at
ion
Gu
ide
ons

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

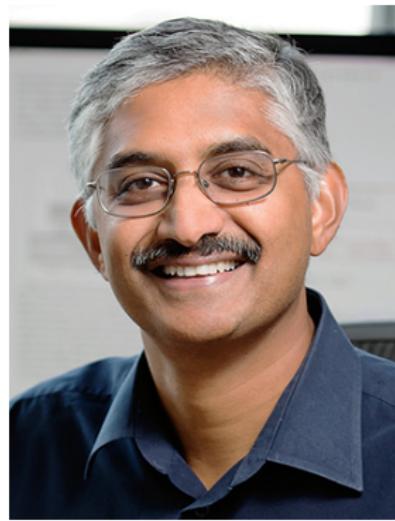
Latest LLVM Release!

17 Apr 2024: LLVM 18.1.4 is now available for download! LLVM is publicly available under an open

“Low Level Virtual Machine”



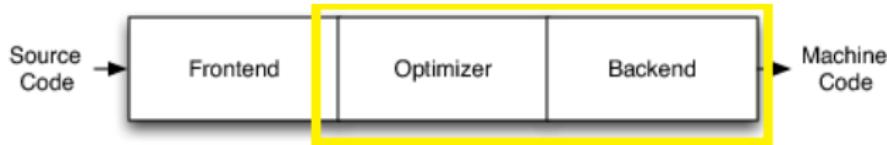
Chris Lattner (1978); UIUC 2000



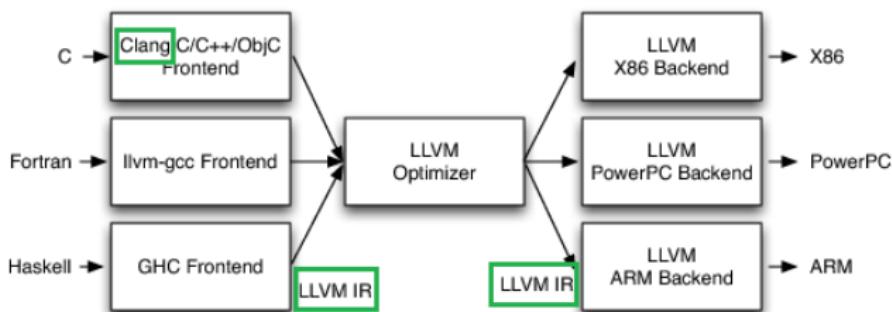
Vikram Adve (1966)



ACM Software System Award (历年获奖名单)



LLVM IR (Intermediate Representation)

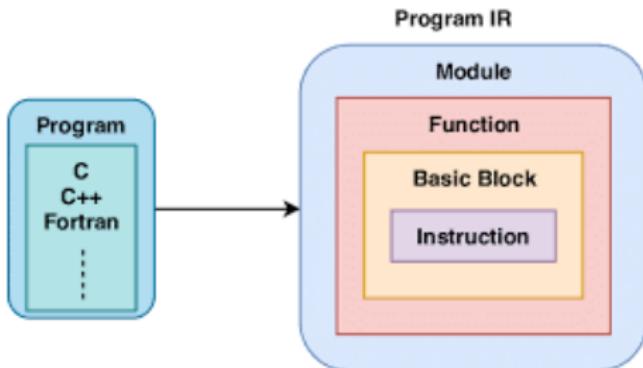


“IR 设计的优秀与否决定着整个编译器的好坏”



8 章技术内容, 其中 4 章介绍 Maple IR, 另外 4 章基于 Maple IR

LLVM Language Reference Manual



IR: Intermediate Representation

LLVM IR: 带类型的、介于高级程序设计语言与汇编语言之间
(LLVM Assembly Language)

"TALK IS
CHEAP.
SHOW ME THE
CODE."
-LINUS TORVALDS



顺序语句 (函数调用)、选择语句、循环语句

factorial0.c @ Compiler Explorer

```
int factorial(int val);

int main(int argc, char **argv) {
    return factorial(val: 2) * 7 == 42;
}
```

```
6 ; Function Attrs: nounwind uwtable
7 define dso_local i32 @main(i32 %0, i8** nocapture readnone %1)
8     %3 = call i32 @factorial(i32 2) #2
9     %4 = mul nsw i32 %3, 7
10    %5 = icmp eq i32 %4, 42
11    %6 = zext i1 %5 to i32
12    ret i32 %6
13 }
```

```
clang -S -emit-llvm -fno-discard-value-names
factorial0.c -o f0-opt0.ll -O1 -g0
```

Three Address Code (TAC)

```
6 ; Function Attrs: nounwind uwtable
7 define dso_local i32 @main(i32 %0, i8** nocapture readnone %1)
8     %3 = call i32 @factorial(i32 2) #2
9     %4 = mul nsw i32 %3, 7
10    %5 = icmp eq i32 %4, 42
11    %6 = zext i1 %5 to i32
12    ret i32 %6
13 }
```

```
clang -S -emit-llvm -fno-discard-value-names
factorial0.c -o f0-opt0.ll -O1 -g0
```

Three Address Code (TAC)

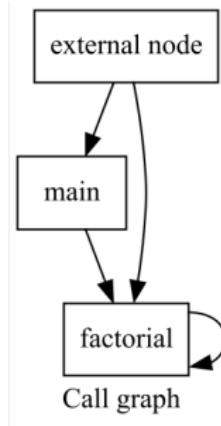
Static Single Assignment (SSA)

```
6 ; Function Attrs: nounwind uwtable
7 define dso_local i32 @main(i32 %0, i8** nocapture readnone %1)
8     %3 = call i32 @factorial(i32 2) #2
9     %4 = mul nsw i32 %3, 7
10    %5 = icmp eq i32 %4, 42
11    %6 = zext i1 %5 to i32
12    ret i32 %6
13 }
```

```
clang -S -emit-llvm -fno-discard-value-names
factorial0.c -o f0-opt0.ll -O1 -g0
```

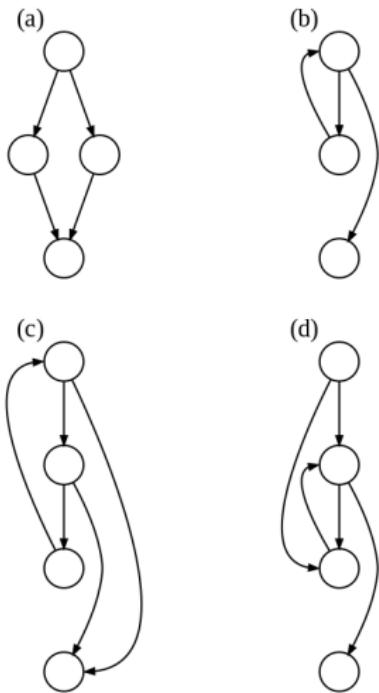
```
int factorial(int val);  
  
int main(int argc, char **argv) {  
    return factorial(val: 2) * 7 == 42;  
}  
  
// precondition: val is non-negative  
int factorial(int val) {  
    if (val == 0) {  
        return 1;  
    }  
  
    return val * factorial(val: val - 1);  
}
```

factorial1.c





Frances Elizabeth Allen
(1932 ~ 2020; 2006 Turing Award)



(Intra-procedure) Control Flow Graph (CFG)

Control Flow Graph (CFG)

Definition (CFG)

Each **node** represents a *basic block*, i.e. a straight-line code sequence with no **branches/jumps** in except to the **entry point** and no **branches/jumps** out except at the **exit point**.

Control Flow Graph (CFG)

Definition (CFG)

Each **node** represents a *basic block*, i.e. a straight-line code sequence with no **branches/jumps** in except to the **entry point** and no **branches/jumps** out except at the **exit point**.

Jump targets start a block, and jumps end a block.

Control Flow Graph (CFG)

Definition (CFG)

Each **node** represents a *basic block*, i.e. a straight-line code sequence with no **branches/jumps** in except to the **entry point** and no **branches/jumps** out except at the **exit point**.

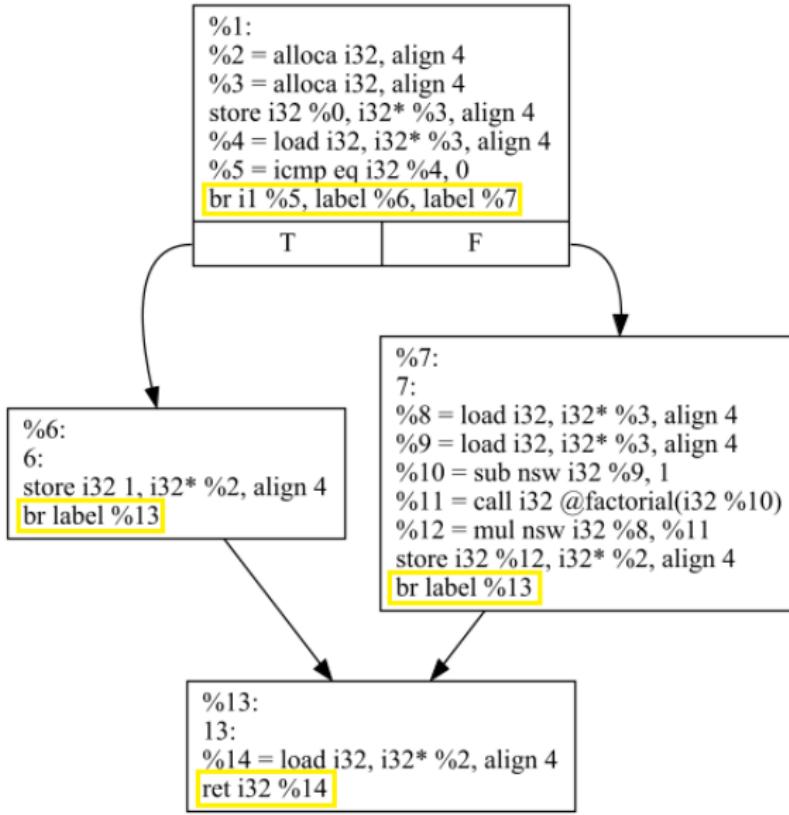
Jump targets start a block, and jumps end a block.

Directed **edges** are used to represent jumps in the control flow.

factorial1.c @ Compiler Explorer

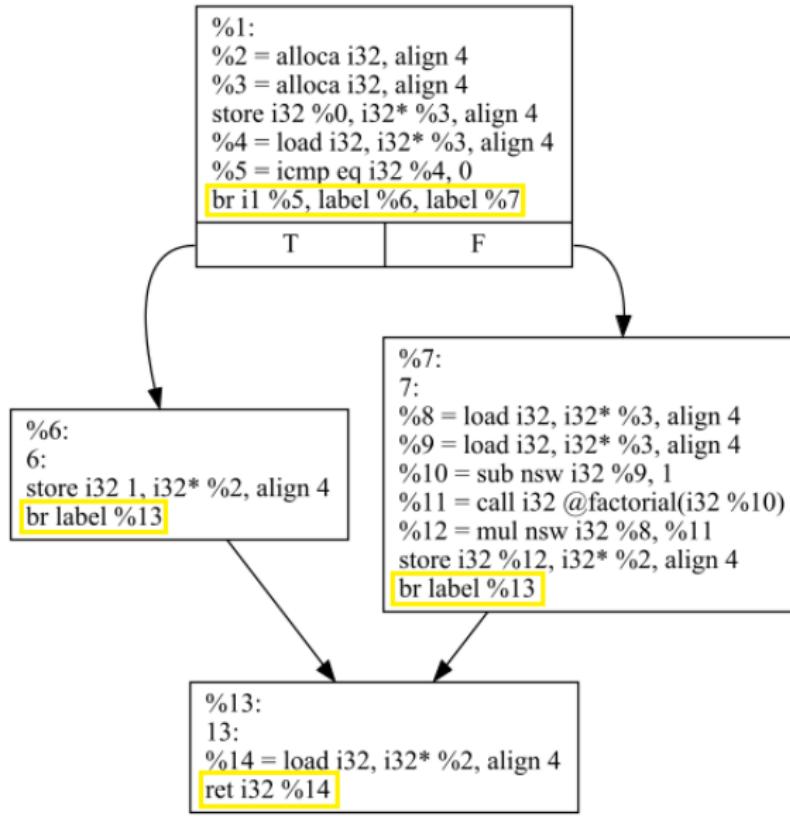
```
int factorial(int val) {
    if (val == 0) {
        return 1;
    }

    return val * factorial(val: val - 1);
}
```



CFG for 'factorial' function

%2: store the return value (in different branches)



CFG for 'factorial' function

Instruction Reference

- Terminator Instructions

- ‘ret’ Instruction
- ‘br’ Instruction
- ‘switch’ Instruction
- ‘indirectbr’ Instruction
- ‘invoke’ Instruction
- ‘callbr’ Instruction
- ‘resume’ Instruction
- ‘catchswitch’ Instruction
- ‘catchret’ Instruction
- ‘cleanupret’ Instruction
- ‘unreachable’ Instruction



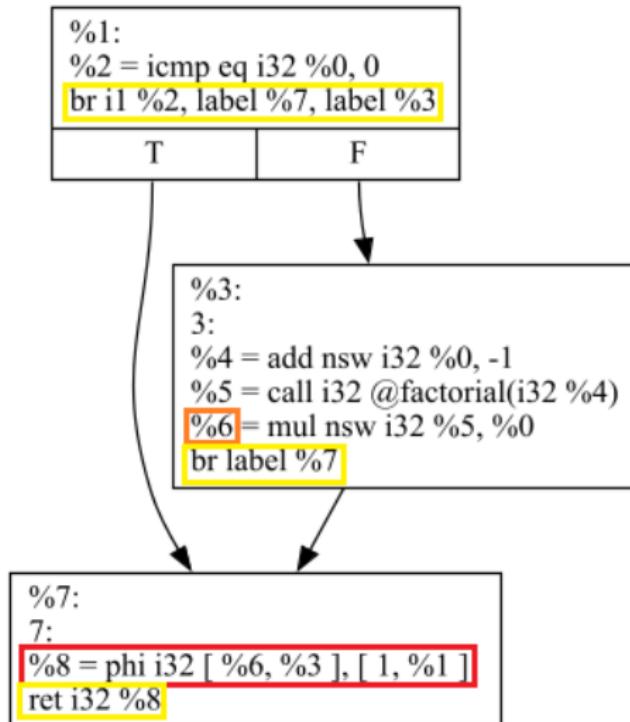
为什么基本块的中间某条指令可以是 call 指令？



为什么基本块的中间某条指令可以是 call 指令？

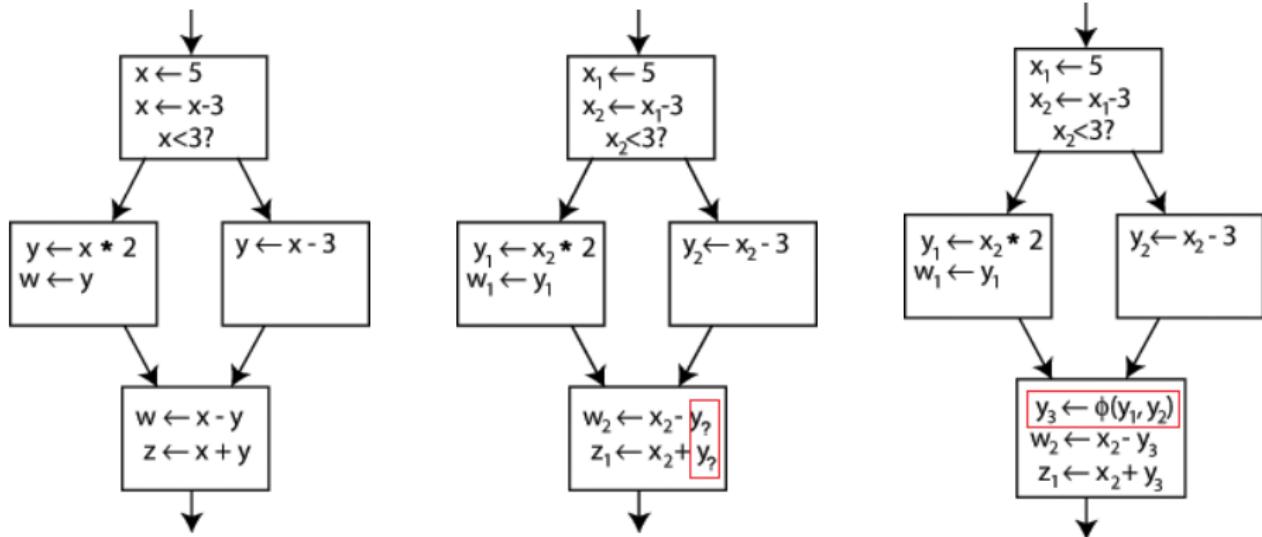
Terminator Instructions

As mentioned [previously](#), every basic block in a program ends with a “Terminator” instruction, which indicates which block should be executed after the current block is finished. These [terminator instructions](#) typically yield a ‘void’ value: they produce control flow, not values (the one exception being the [‘invoke’](#) instruction).

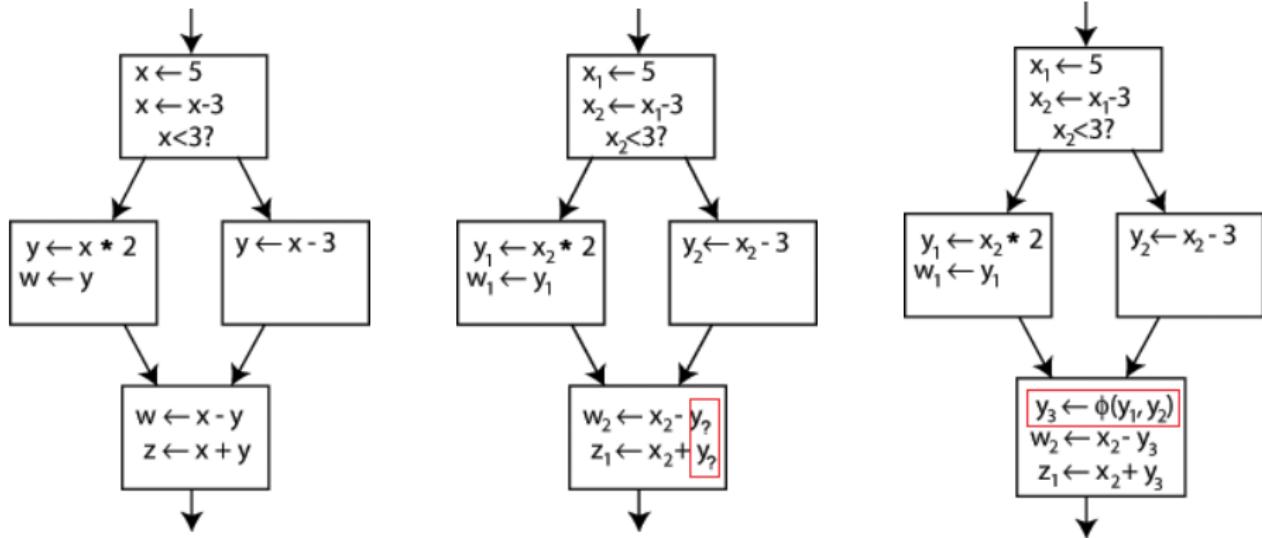


CFG for 'factorial' function

ϕ 根据控制流决定选择 y_1 还是 y_2

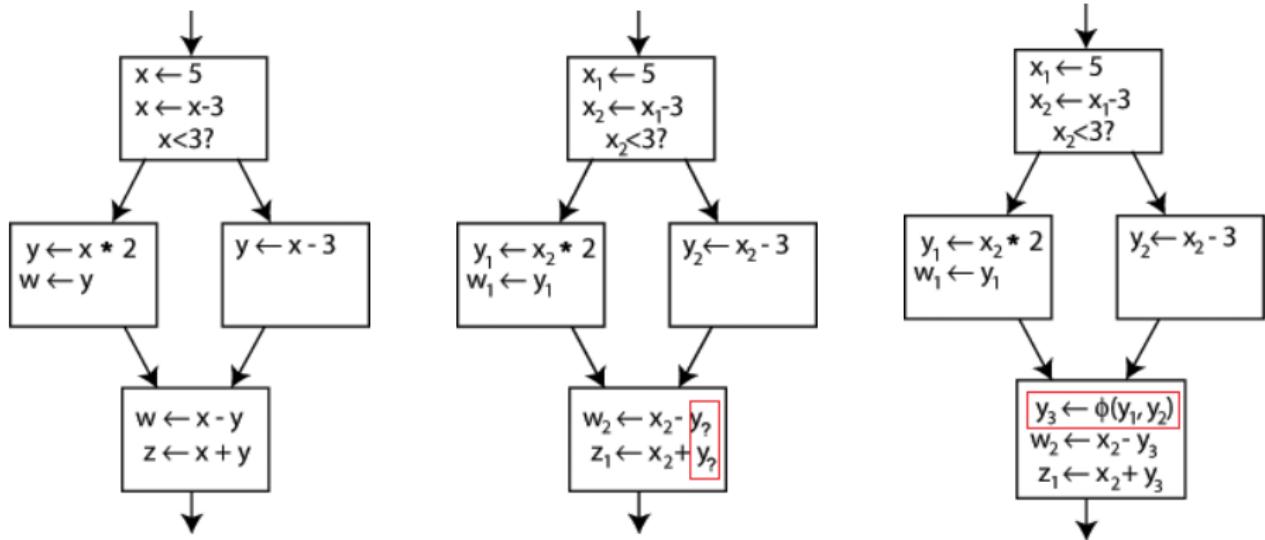


ϕ 根据控制流决定选择 y_1 还是 y_2



How to implement ϕ instruction?

ϕ 根据控制流决定选择 y_1 还是 y_2



How to implement ϕ instruction?

基本思想：将 ϕ 指令转换成若干赋值指令，上推至前驱基本块中

SSA 形式的构建 (Construction)、消去 (Destruction)、重建 (Reconstruction)



Section 4.3

SSA 形式的构建 (Construction)、消去 (Destruction)、重建 (Reconstruction)



Section 4.3



Section 9.3

SSA 形式的构建 (Construction)、消去 (Destruction)、重建 (Reconstruction)



Efficiently Computing Static Single Assignment Form and the Control Dependence Graph

RON CYTRON, JEANNE FERRANTE, BARRY K. ROSEN, and
MARK N. WEGMAN
IBM Research Division
and
F. KENNETH ZADECK
Brown University

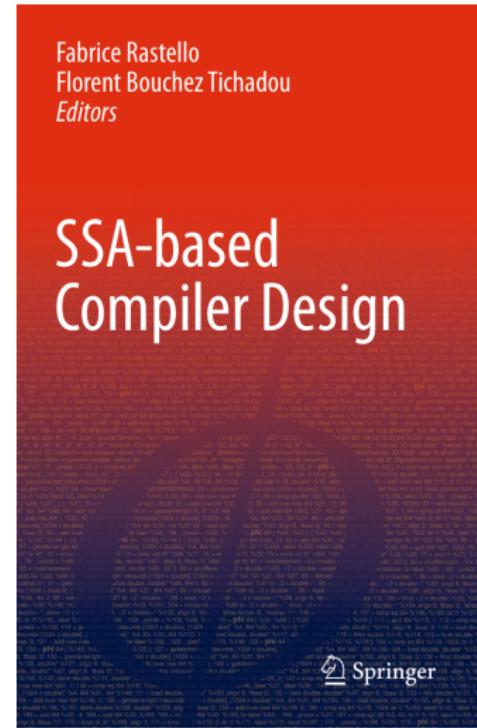
TOPLAS1991 @
compilers-papers-we-love

Section 4.3

Section 9.3



“The SSA Book”



Springer

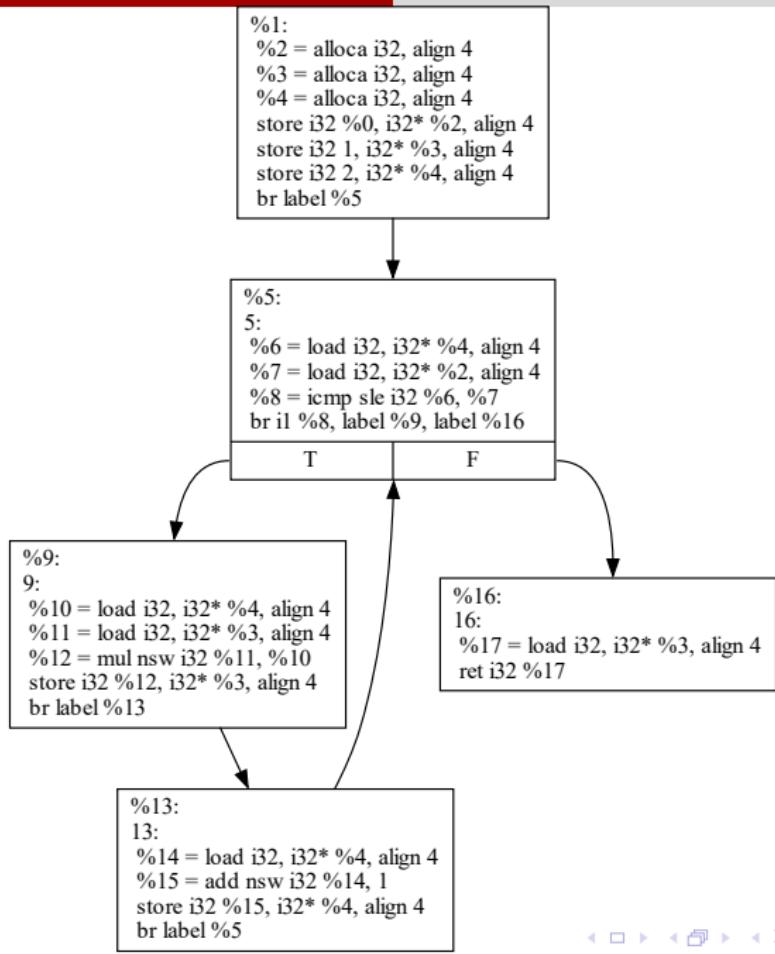
<https://github.com/courses-at-nju-by-hfwei/compilers-resources/tree/master/books/Classic%20Textbooks>

factorial2.c @ Compiler Explorer

```
int factorial(int val) {
    int temp = 1;

    for (int i = 2; i <= val; i++) {
        temp *= i;
    }

    return temp;
}
```



```

int factorial(int val) {
    int temp = 1;

    for (int i = 2; i <= val; i++) {
        temp *= i;
    }

    return temp;
}

```

		%1:
		%2 = icmp slt i32 %0, 2
T	F	br i1 %2, label %3, label %5

		%5:
		5:
		%6 = phi i32 [%9, %5], [2, %1]
		%7 = phi i32 [%8, %5], [1, %1]
		%8 = mul nsw i32 %6, %7
		%9 = add nuw i32 %6, 1
		%10 = icmp eq i32 %6, %0
T	F	br i1 %10, label %3, label %5

		%3:
		3:
		%4 = phi i32 [1, %1], [%8, %5]
		ret i32 %4

CFG for 'factorial' function

LLVM's Intermediate Representation

LLVM IR Animation



LLVM IR Tutorial @ Bilibili

Instruction Reference @ LLVM Language Reference Manual

https://llvm.org/docs/LangRef.html



[LLVM Home](#) | [Documentation](#) » [Reference](#) »

LLVM Language Reference Manual

Instruction Reference

The LLVM instruction set consists of several different classifications of instructions: [terminator instructions](#), [binary instructions](#), [bitwise binary instructions](#), [memory instructions](#), and [other instructions](#). There are also [debug records](#), which are not instructions themselves but are printed interleaved with instructions to describe changes in the state of the program's debug information at each position in the program's execution.

Program Visualization using LLVM @ Bilibili



Program Visualization



使用命令行生成 LLVM IR 与控制流图 (Control Flow Graph)

LLVM Programmer's Manual



A screenshot of a web browser displaying the LLVM Programmer's Manual. The page has a white header bar with a red logo on the left and a search bar on the right. Below the header is the LLVM logo, which features a stylized blue and grey bird-like creature next to the text "LLVM COMPILER INFRASTRUCTURE". The main navigation bar below the logo includes links for "LLVM Home", "Documentation", "Getting Started/Tutorials", and "LLVM Programmer's Manual". The title "LLVM Programmer's Manual" is prominently displayed in a large, dark blue font at the top of the main content area.

如何用编程的方式生成 LLVM IR?

Bytedeco/javacpp @ github

JavaCPP Presets Platform For LLVM



LLVM JAVA API使用手册
准备工作

课程实验中具体如何生成 ϕ 指令？



主打一个“课上不讲，课后自学”吗？

课程实验中具体如何生成 ϕ 指令？



主打一个“课上不讲，课后自学”吗？

课程实验可以使用 `load`, `store` 指令, 不需要采用 ϕ 指令格式

[llvm/factorial/ @ GitHub](#)



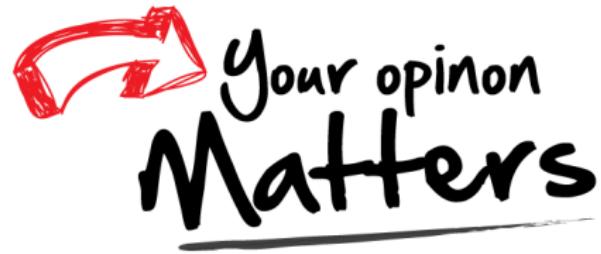
Kaleidoscope: Implementing a Language with LLVM (in C++)

The screenshot shows the LLVM Documentation Tutorial page at <https://llvm.org/docs/tutorial/>. The main content area displays the "Kaleidoscope: Implementing a Language with LLVM" tutorial. The page includes the LLVM logo, a table of contents, and several sections of the tutorial text.

The screenshot shows a GitHub repository page for [courses-at-nju-by-hfwei / kaleidoscope-in-java](#). The repository is public and contains code for the Kaleidoscope project. A pull request titled "finish chapter 3" has been merged, and the commit message indicates it covers files like BinaryExprAST.java, CallExprAST.java, ExprAST.java, FunctionAST.java, NumberExprAST.java, PrototypeAST.java, and VariableExprAST.java, all finished for chapter 3.

kaleidoscope-in-java @ GitHub

Thank You!



Office 926

hfwei@nju.edu.cn