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:
ew
es
ation
Guide
ons

LLVM Overview

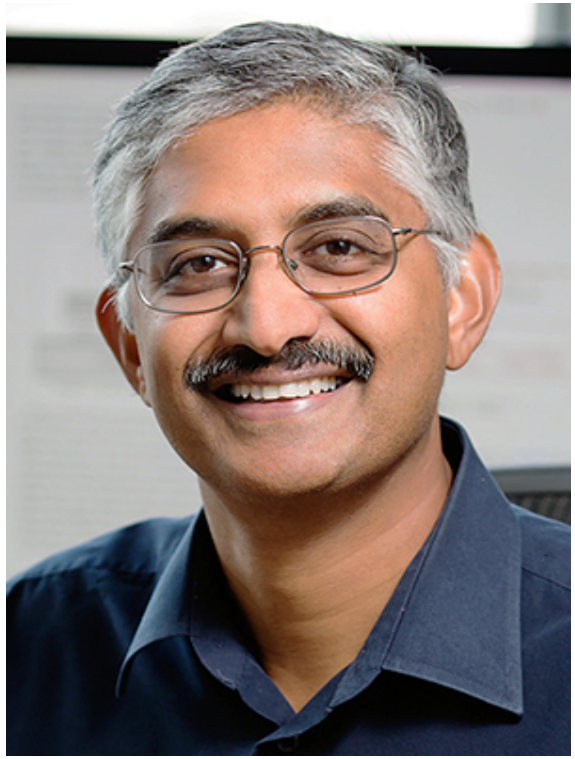
The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

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"

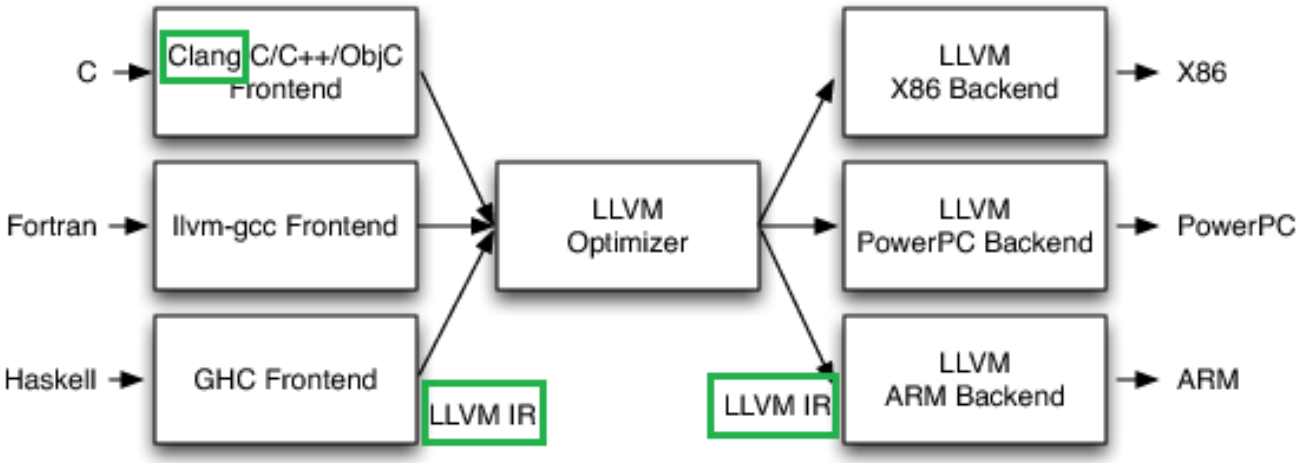
Chris Lattner (1978); UIUC 2000



Vikram Adve (1966)

ACM Software System Award (历年获奖名单)

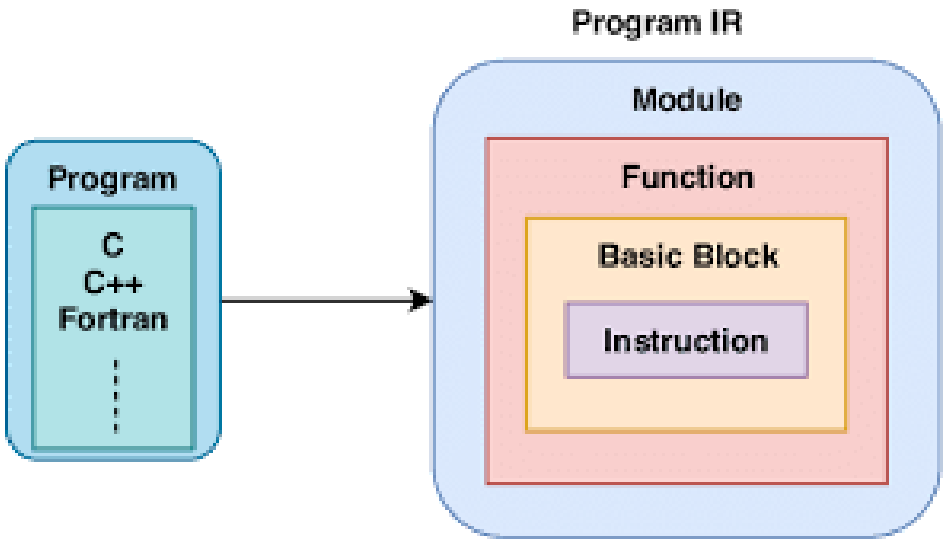
LLVM IR (Intermediate Representation)



“IR 设计的优秀与否决定着整个编译器的好坏”

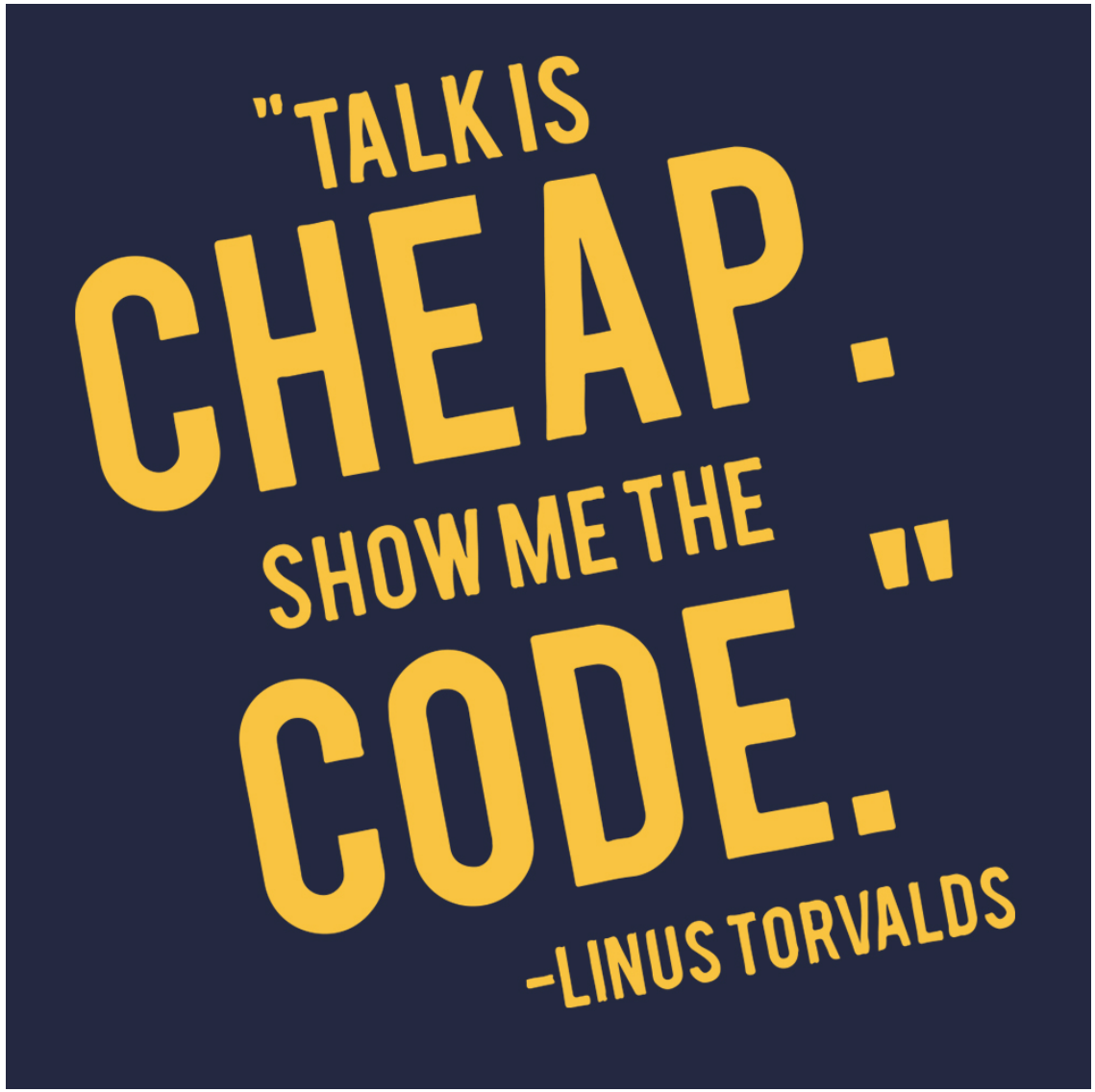
8 章技术内容, 其中 4 章介绍 Maple IR, 另外 4 章基于 Maple IR

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

```c
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  }
```

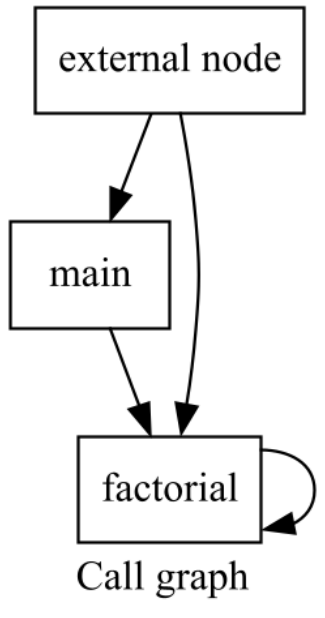
```
clang -S -emit-llvm -fno-discard-value-names
    factorial0.c -o f0-opt0.ll -O1 -g0
```

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
```

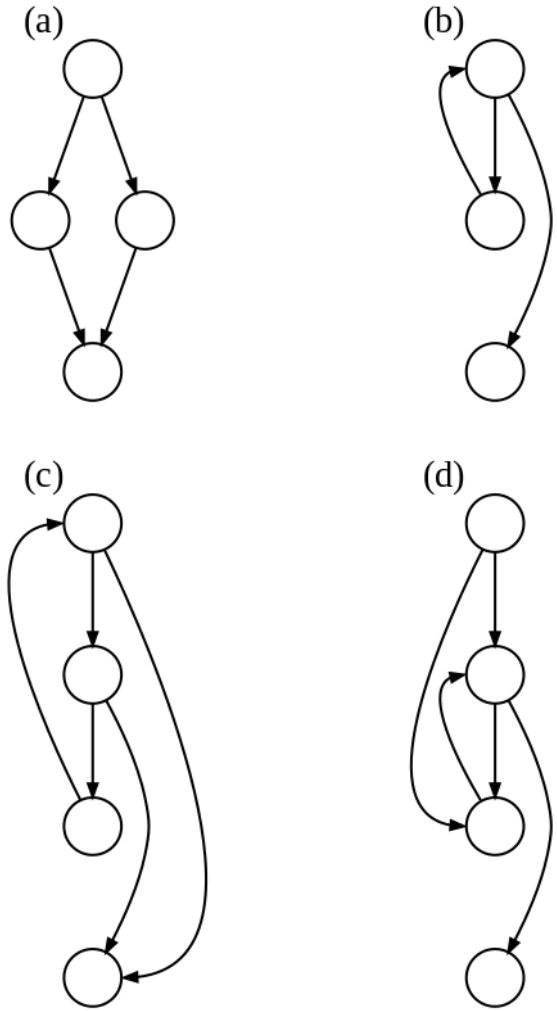
```c
int factorial(int val);

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);
}
```



Call graph

factorial1.c

Frances Elizabeth Allen
(1932 $\sim$ 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)

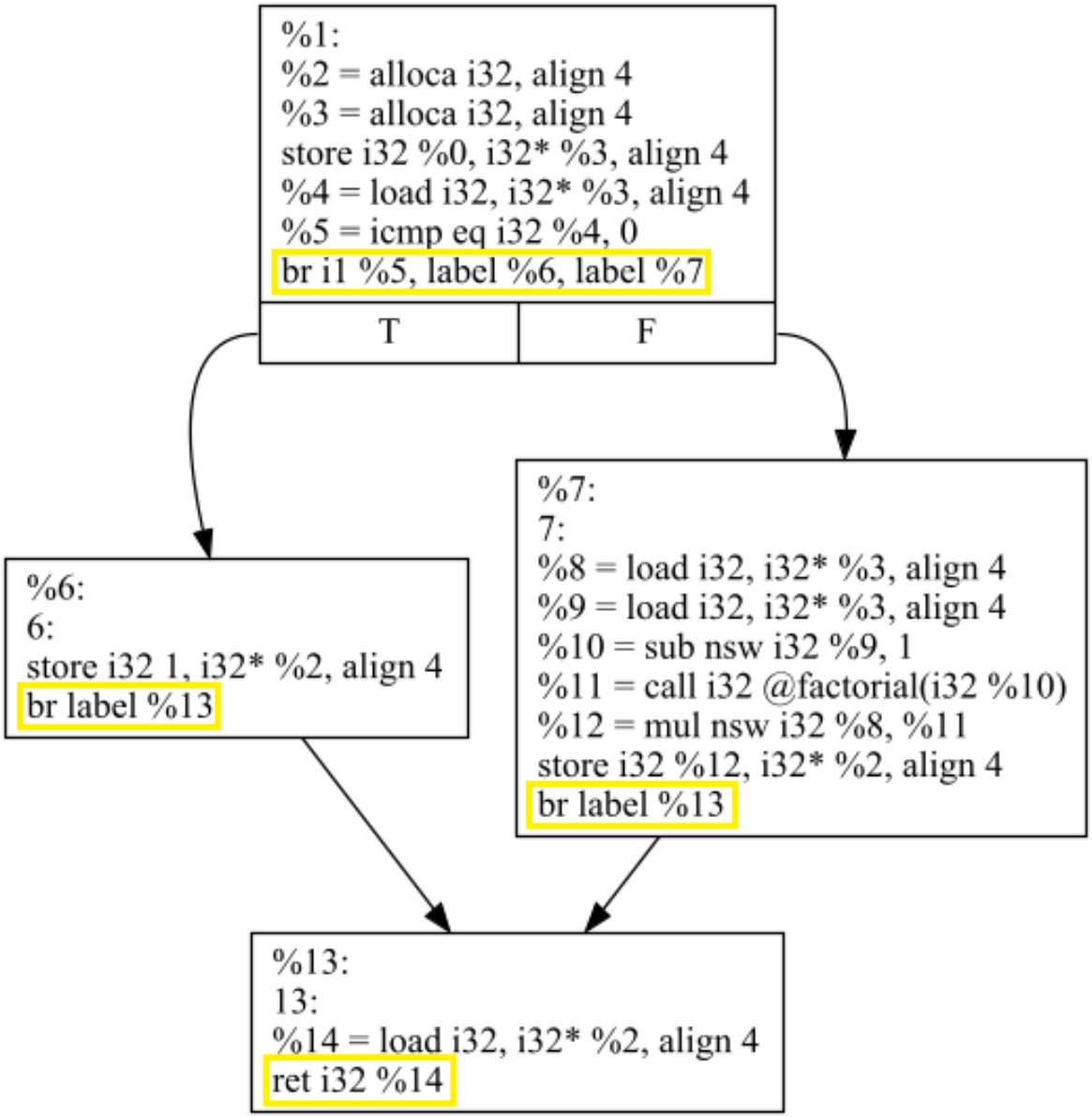
Each node represents a *basic block*, i.e. a straight-line code sequence with no **branches/jumps** in except to the entry point and no **branches/jumps** out except at the exit point.

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

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

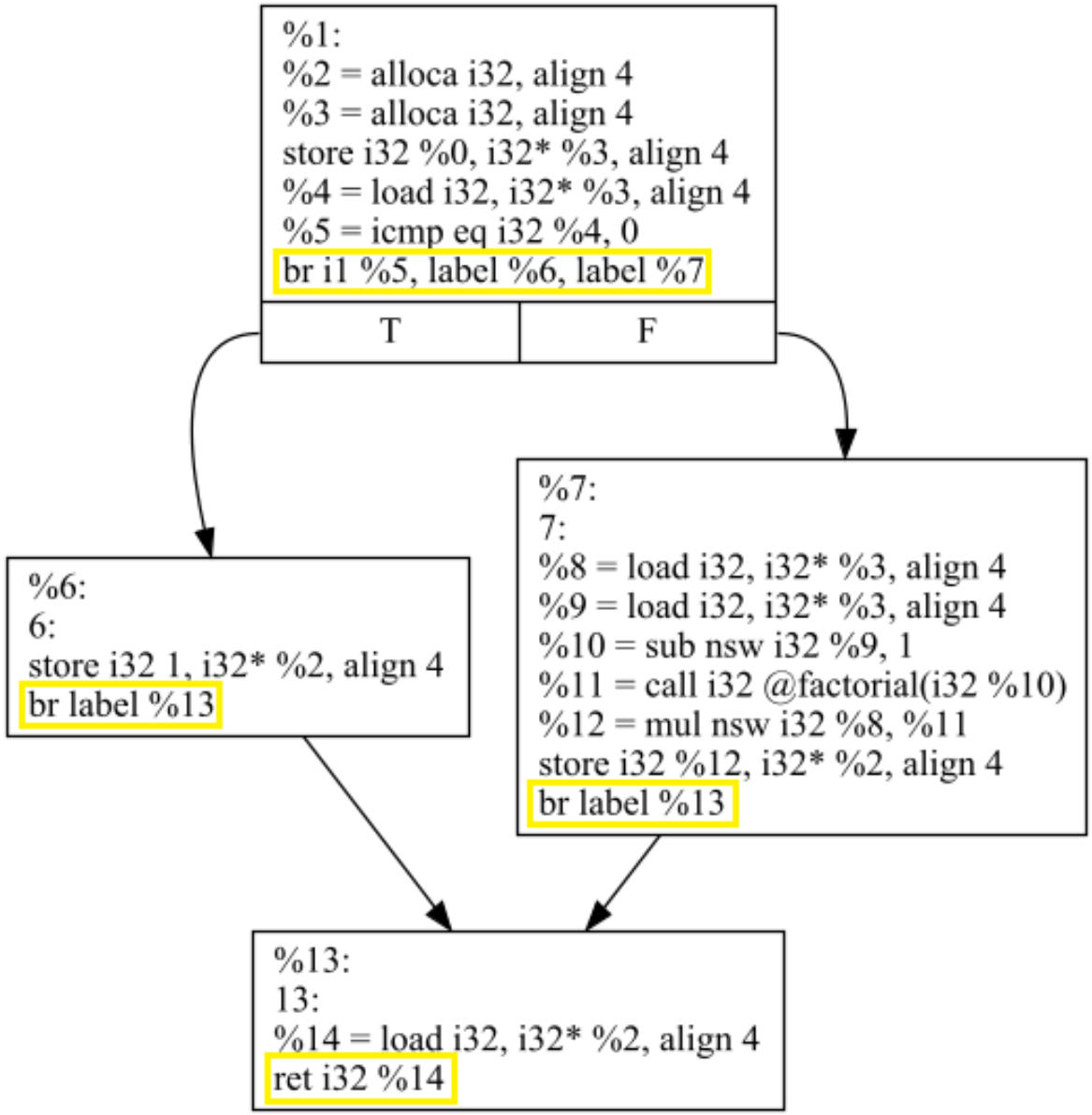
factorial1.c @ Compiler Explorer

```c
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

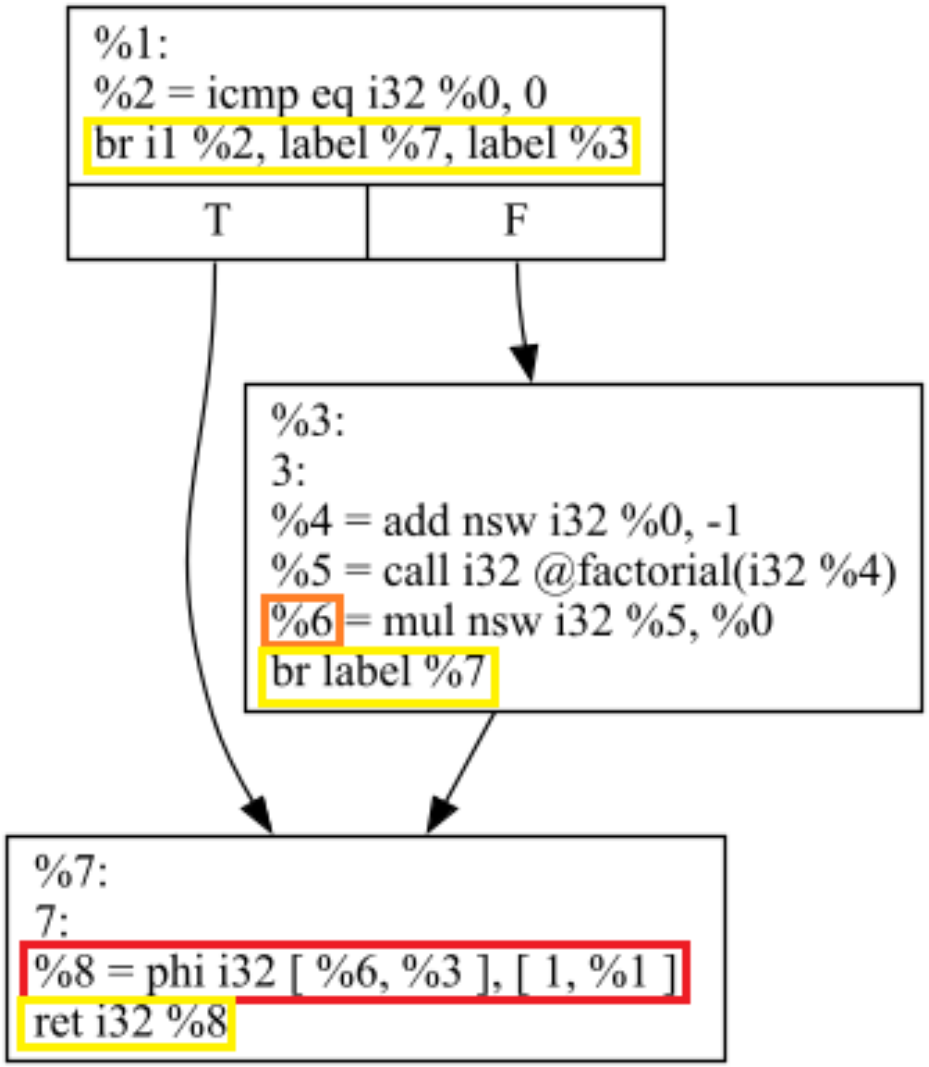
- Instruction Reference
  - Terminator Instructions
    - ‘`ret`’ Instruction
    - ‘`br`’ Instruction
    - ‘`switch`’ Instruction
    - ‘`indirectbr`’ Instruction
    - ‘`invoke`’ Instruction
    - ‘`callbr`’ Instruction
    - ‘`resume`’ Instruction
    - ‘`catchswitch`’ Instruction
    - ‘`catchret`’ 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?

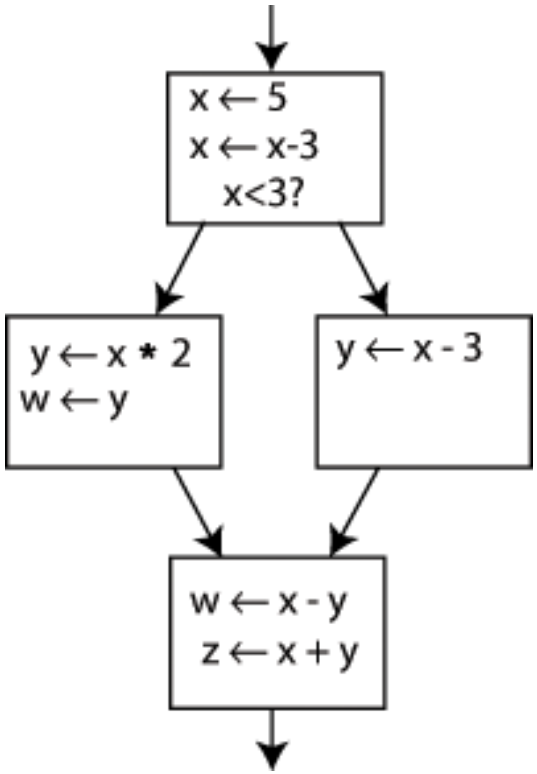
Terminator Instructions

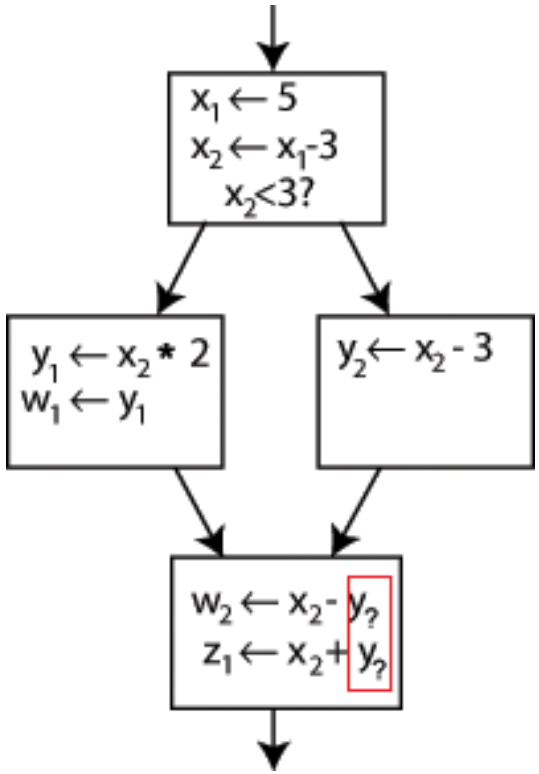
As mentioned previously, every basic block in a program ends with a "Terminator" instruction, which indicates which block should be executed after the current block is finished. These terminator instructions typically yield a 'void' value: they produce control flow, not values (the one exception being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

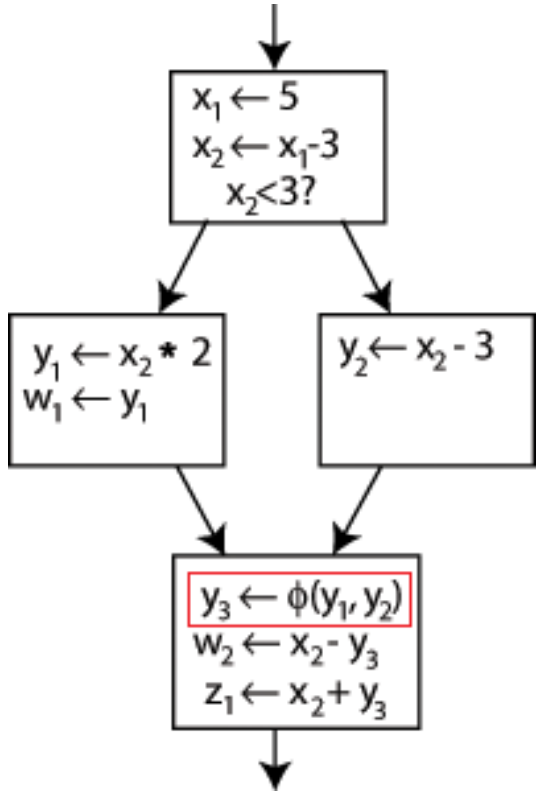


CFG for 'factorial' function

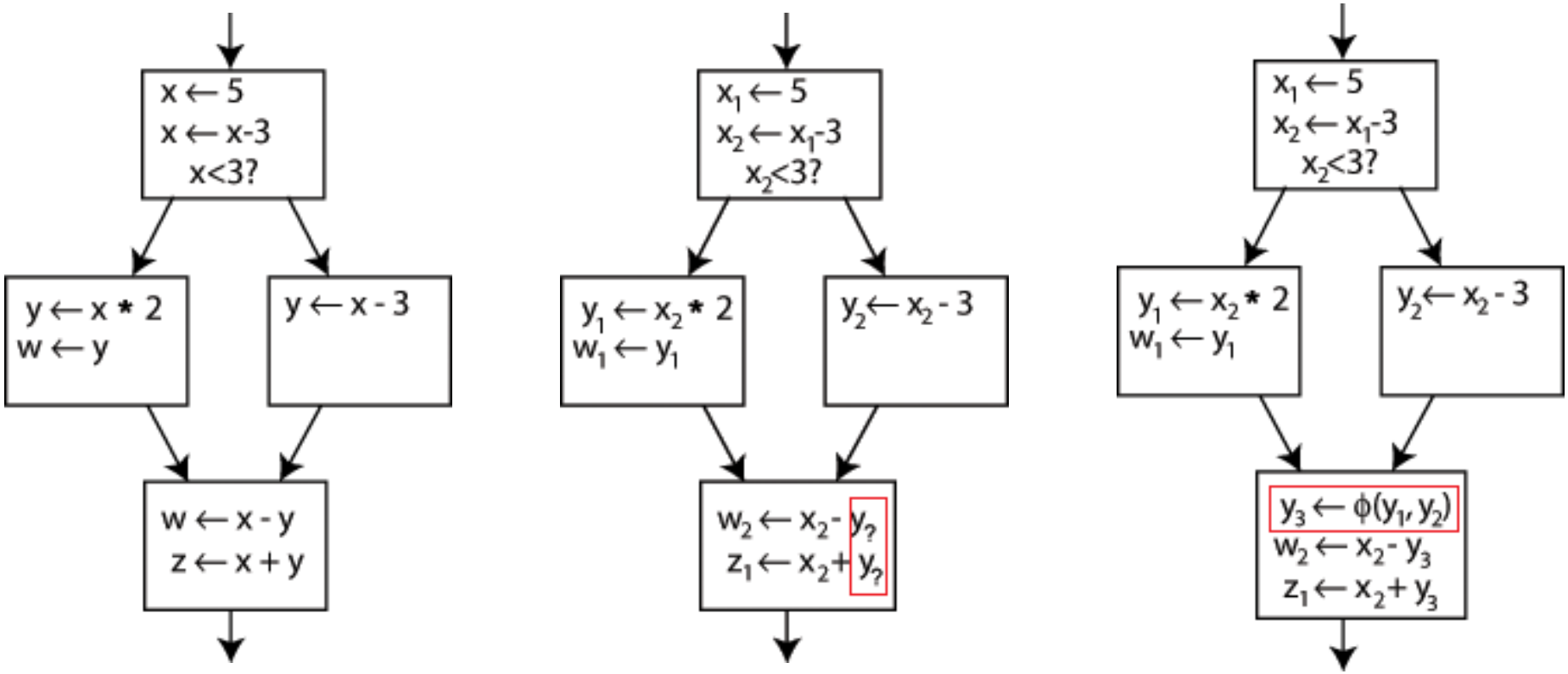
ϕ 根据控制流决定选择 y_1 还是 y_2





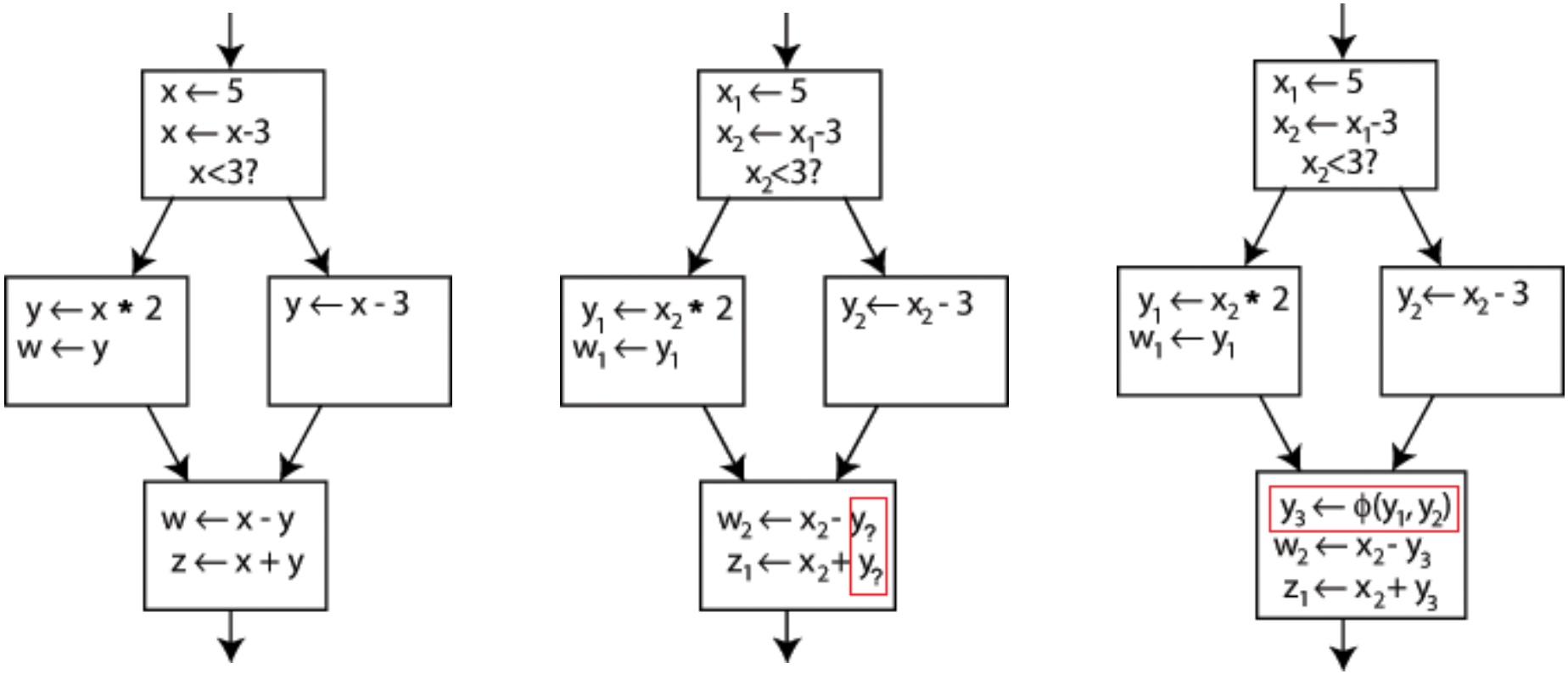


ϕ 根据控制流决定选择 y_1 还是 y_2



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

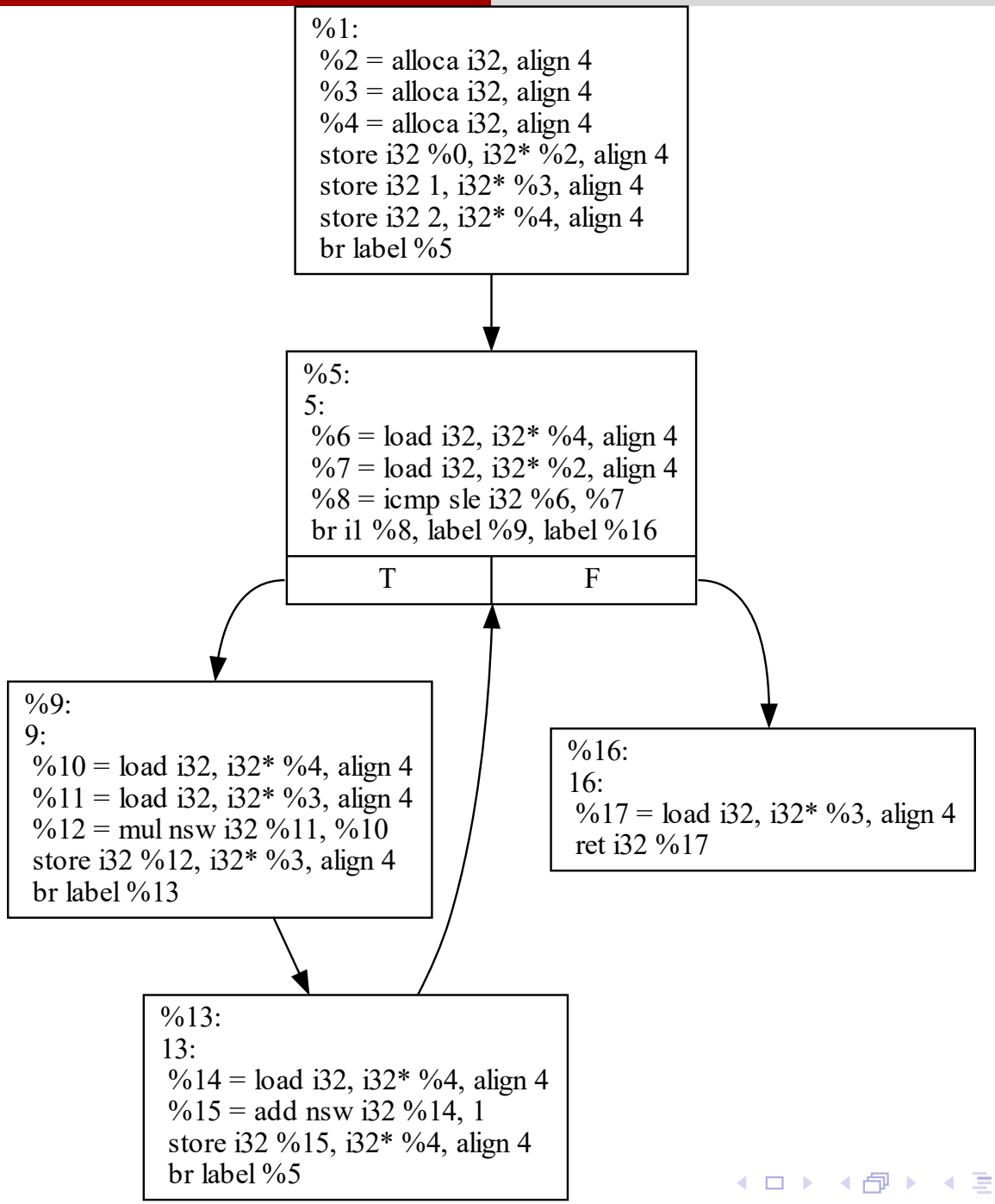
Section 9.3

“The SSA Book”

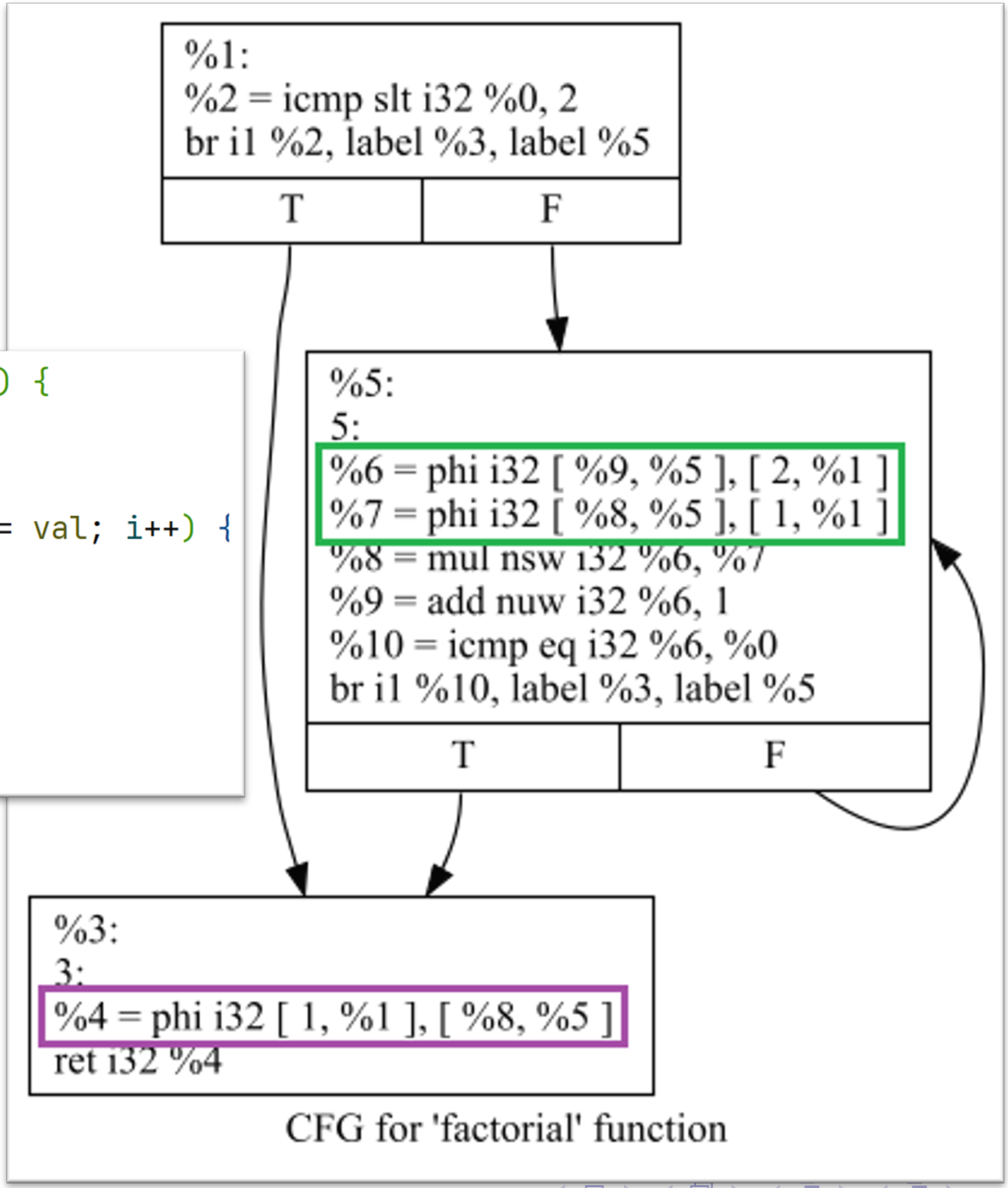
https://github.com/courses-at-nju-by-hfwei/compilers-resources/tree/master/books/Classic%20Textbooks

factorial2.c @ Compiler Explorer

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

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

  return temp;
}
```


%1:
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 %0, i32* %2, align 4
store i32 1, i32* %3, align 4
store i32 2, i32* %4, align 4
br label %5
%5:
5:
%6 = load i32, i32* %4, align 4
%7 = load i32, i32* %2, align 4
%8 = icmp sle i32 %6, %7
br i1 %8, label %9, label %16
T
F
%9:
9:
%10 = load i32, i32* %4, align 4
%11 = load i32, i32* %3, align 4
%12 = mul nsw i32 %11, %10
store i32 %12, i32* %3, align 4
br label %13
%16:
16:
%17 = load i32, i32* %3, align 4
ret i32 %17
%13:
13:
%14 = load i32, i32* %4, align 4
%15 = add nsw i32 %14, 1
store i32 %15, i32* %4, align 4
br label %5

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

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

  return temp;
}
```



CFG for 'factorial' function

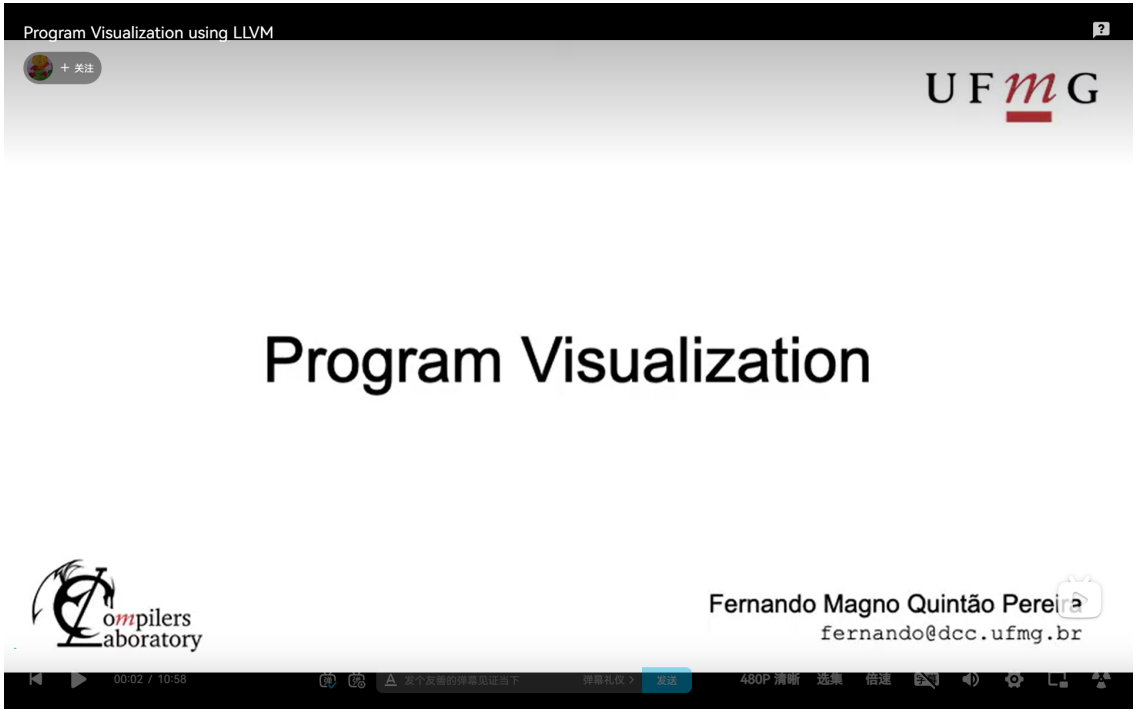
LLVM's Intermediate Representation

LLVM IR Animation

LLVM IR Tutorial @ Bilibili

Instruction Reference @ LLVM Language Reference Manual

Program Visualization using LLVM @ Bilibili



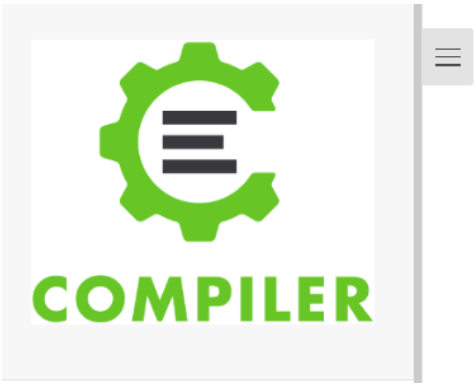
使用命令行生成 LLVM IR 与控制流图 (Control Flow Graph)

LLVM Programmer's Manual

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?

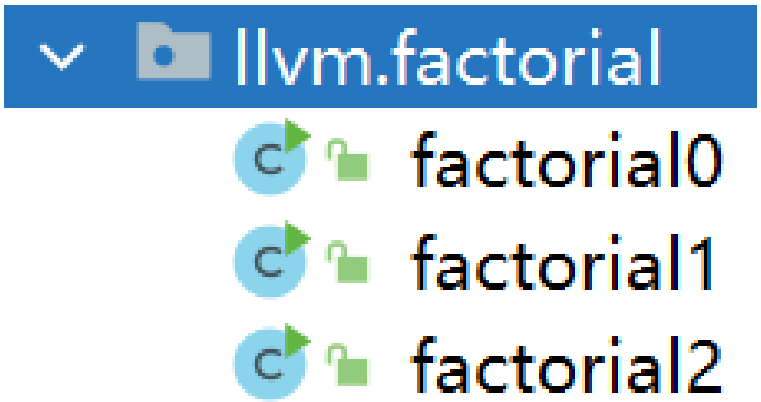
主打一个 “课上不讲, 课后自学”吗?

课程实验中具体如何生成 ϕ 指令?

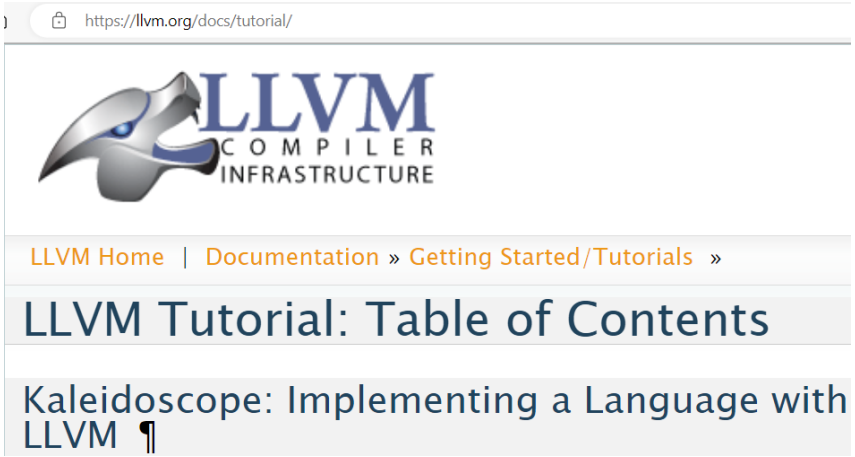
主打一个 “课上不讲, 课后自学”吗?

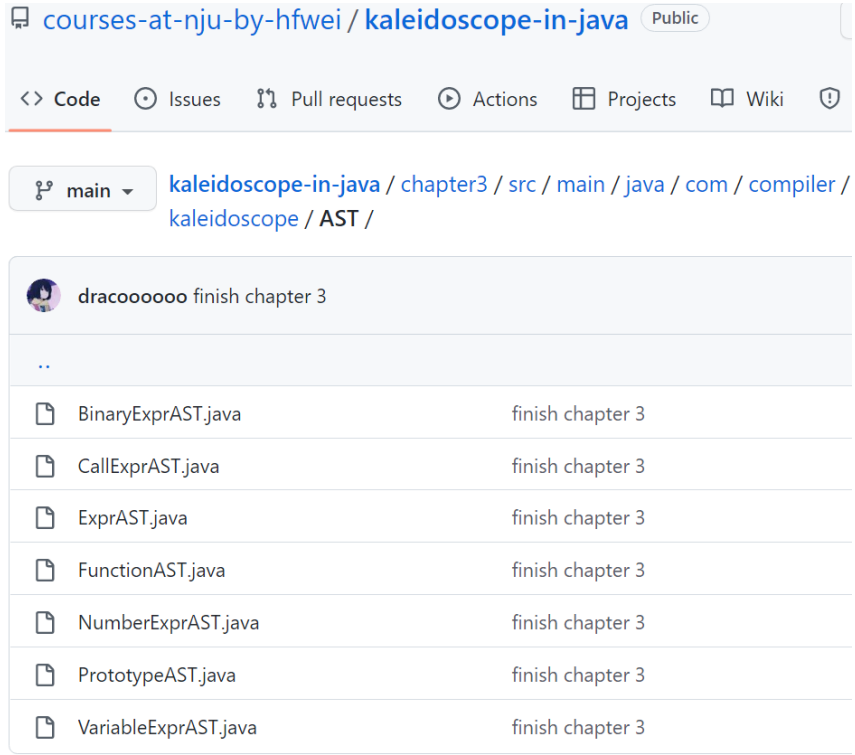
课程实验可以使用 `load`, `store` 指令, 不需要采用 ϕ 指令格式

llvm/factorial/ @ GitHub



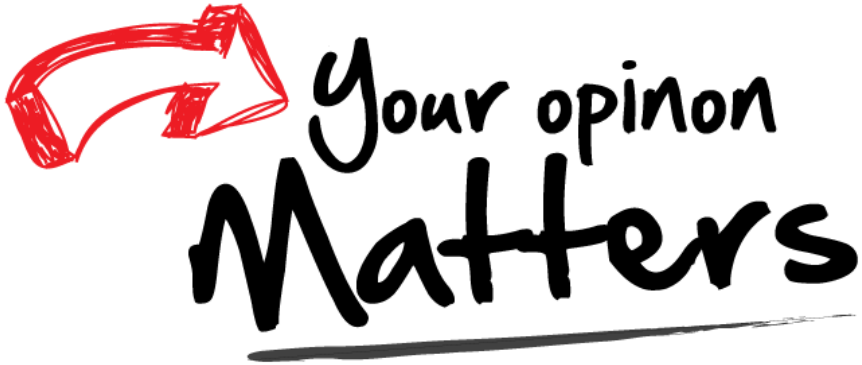
Kaleidoscope: Implementing a Language with LLVM (in C++)





kaleidoscope-in-java @ GitHub


Thank
You!



Office 926

hfwei@nju.edu.cn