LLVM编译器简介

LLVM (Low-Level Virtual Machine) 中间码

钮鑫涛

LLVM编译系统

- LLVM编译器基础设施
  - 提供了可以构建编译器的可复用部件（模块化）
    - 大大减少了新的编译器的开发成本和时间
    - 可以用于开发不同种类的编译器
    - 有很多部件：JITs、trace-based optimizers…
- 基于LLVM的编译器框架
  - 很多端到端（end-to-end）的编译器使用LLVM的基础设施
    - 最早支持C/C++,现在支持JAVA、Python、Rust、Haskell等

LLVM巨大的贡献

- 由Chris Lattner于2000年在UIUC创建 (advised by Vikram Adve)
- 该项目获得了2012年的ACM Software System Award

Specific Types of Contributions

ACM Software System Award

Institutions/individuals who developed software systems with lasting influence on computing

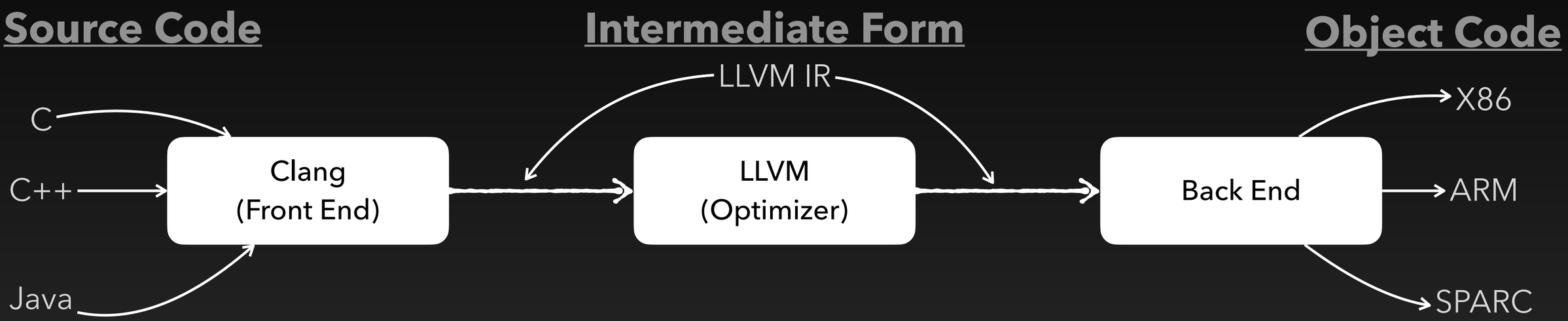
历年名单

Chris Lattner

Vikram Adve

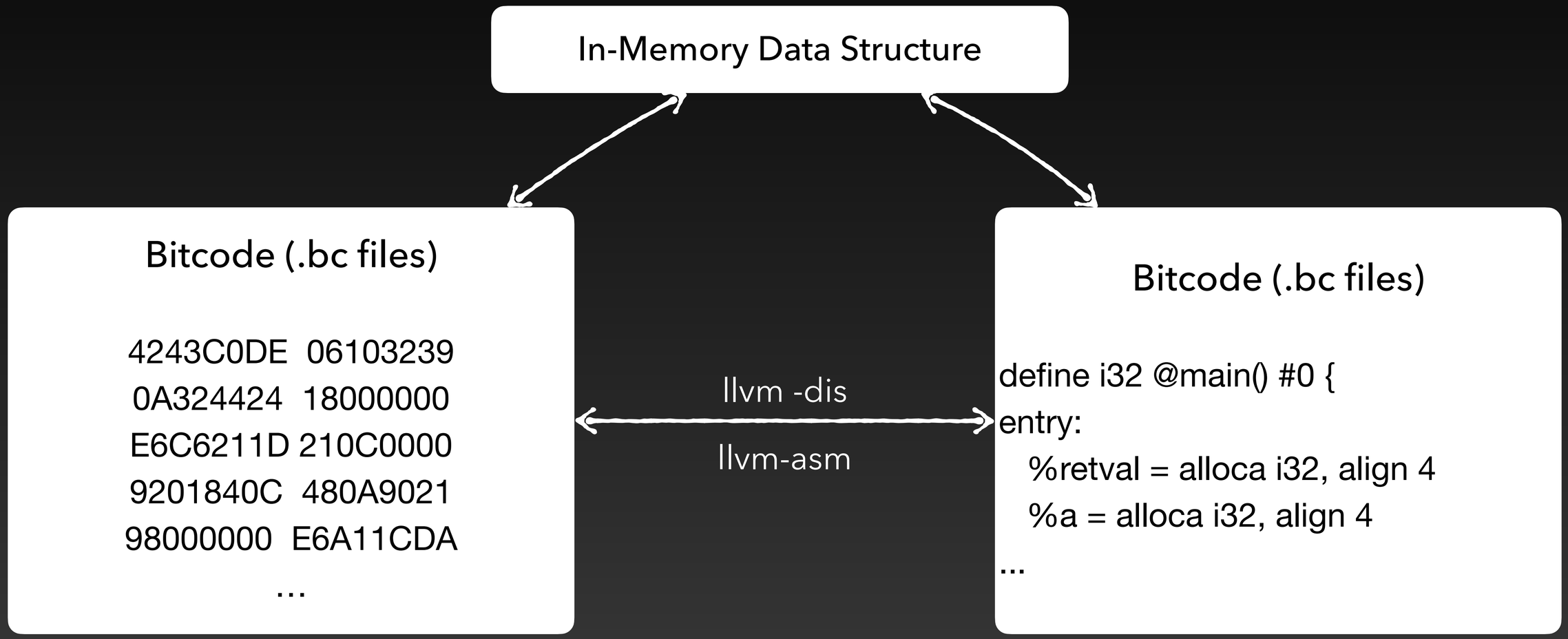
LLVM编译系统概况

Three Phases Design



- LLVM 优化器是一系列的“passes”
  - LLVM会做一系列的分析（Analysis）和优化（Optimization），一个接着一个
  - 分析的pass不会改变代码（只会告诉你哪里可以优化），优化的pass改变代码
- LLVM的中间码形式是一套虚拟的指令集（Virtual Instruction Set）
  - LLVM的passes作用于LLVM 中间码, 其独立于源代码语言和目标平台

LLVM IR (Intermediate Representation)



Bitcode files and LLVM IR text files are lossless serialization formats!

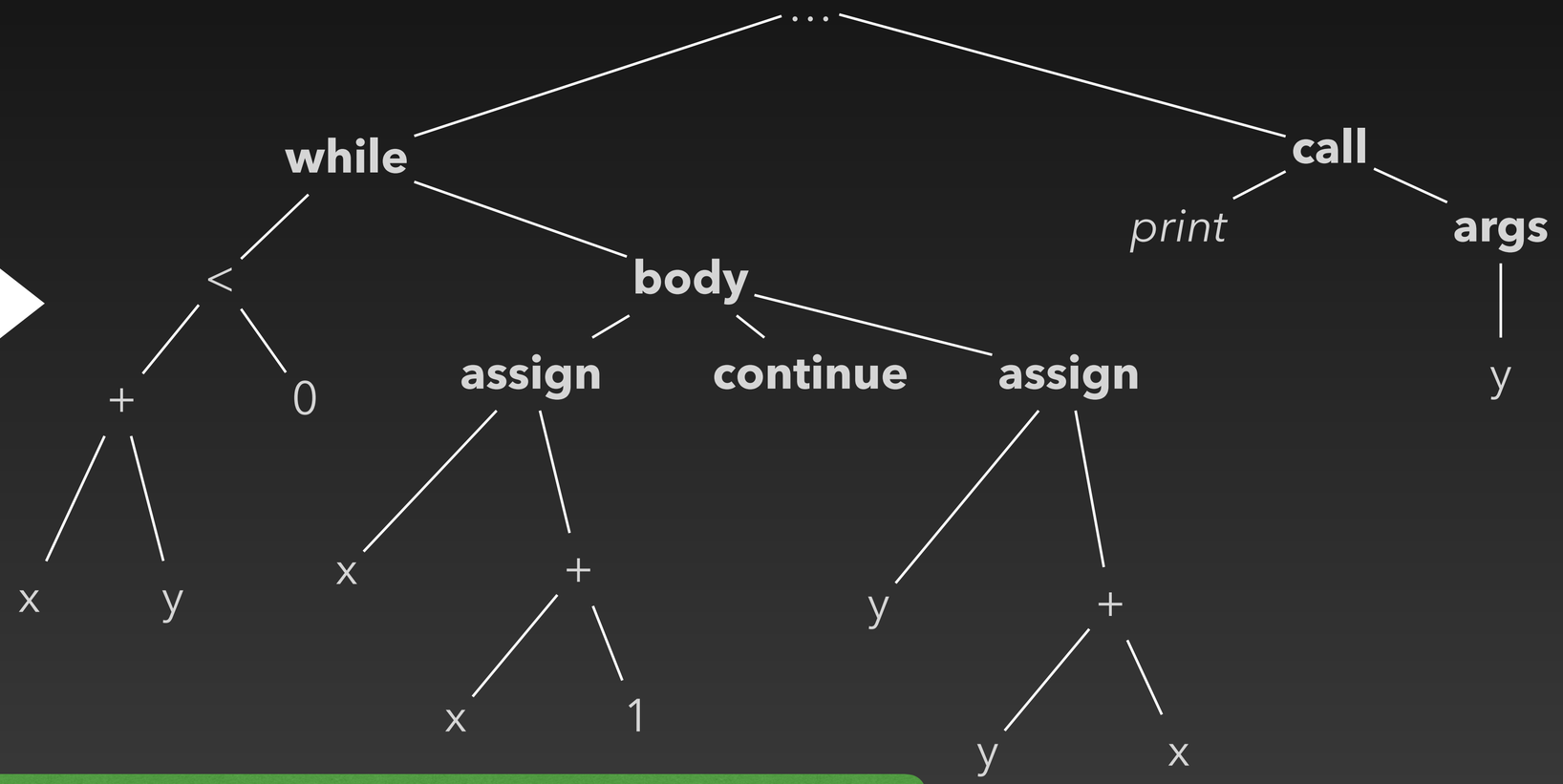
We can pause optimization and come back later.

为什么需要中间码

例子谭添老师的编译原理

- 直接源码 –>AST–> 机器码?

```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```



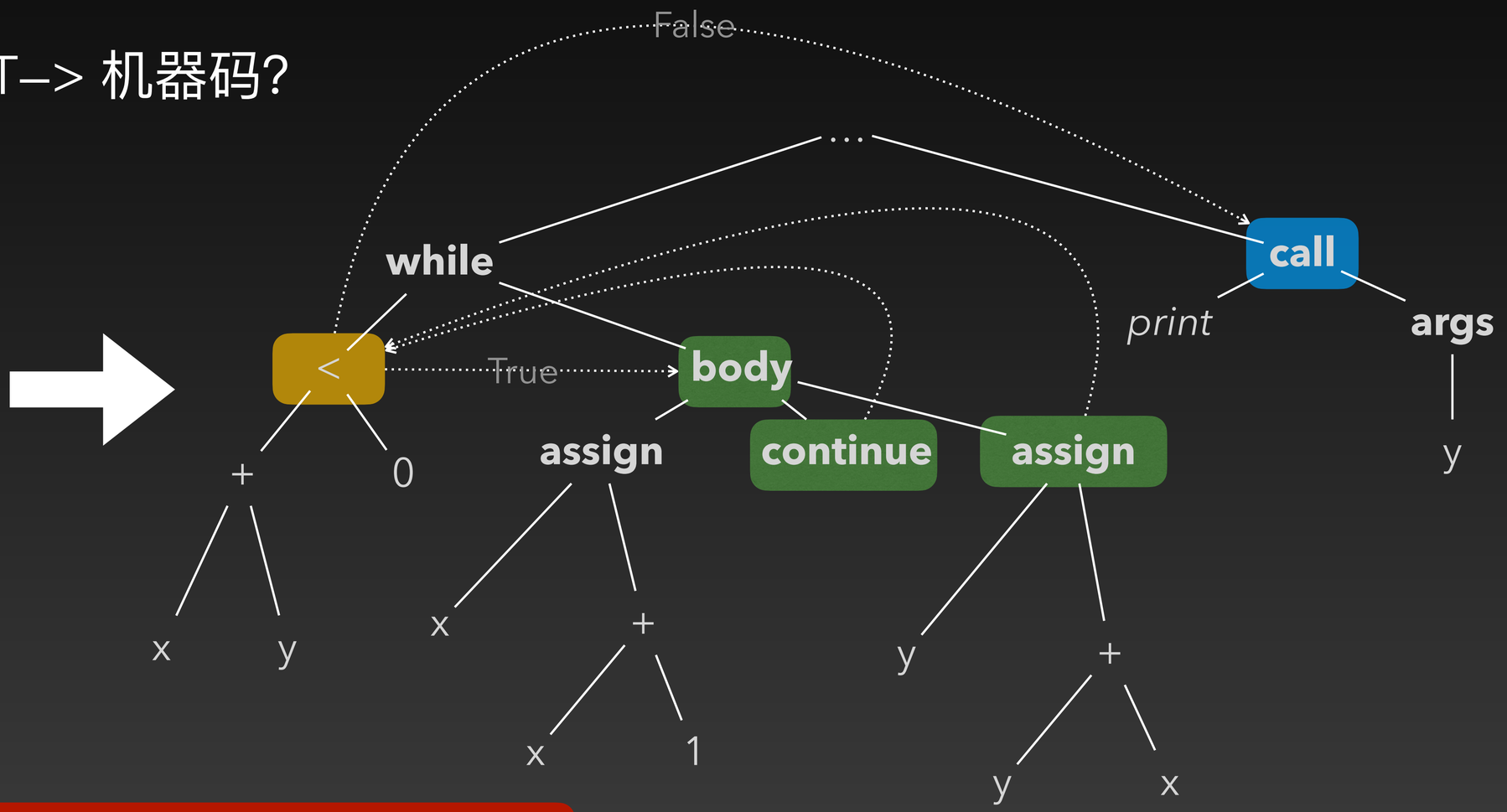
优点：high-level的程序表示（贴近源语言），与源语言语法高度相关，适合做类型检查

为什么需要中间码

例子谭添老师的编译原理

- 直接源码 –>AST–> 机器码?

```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```

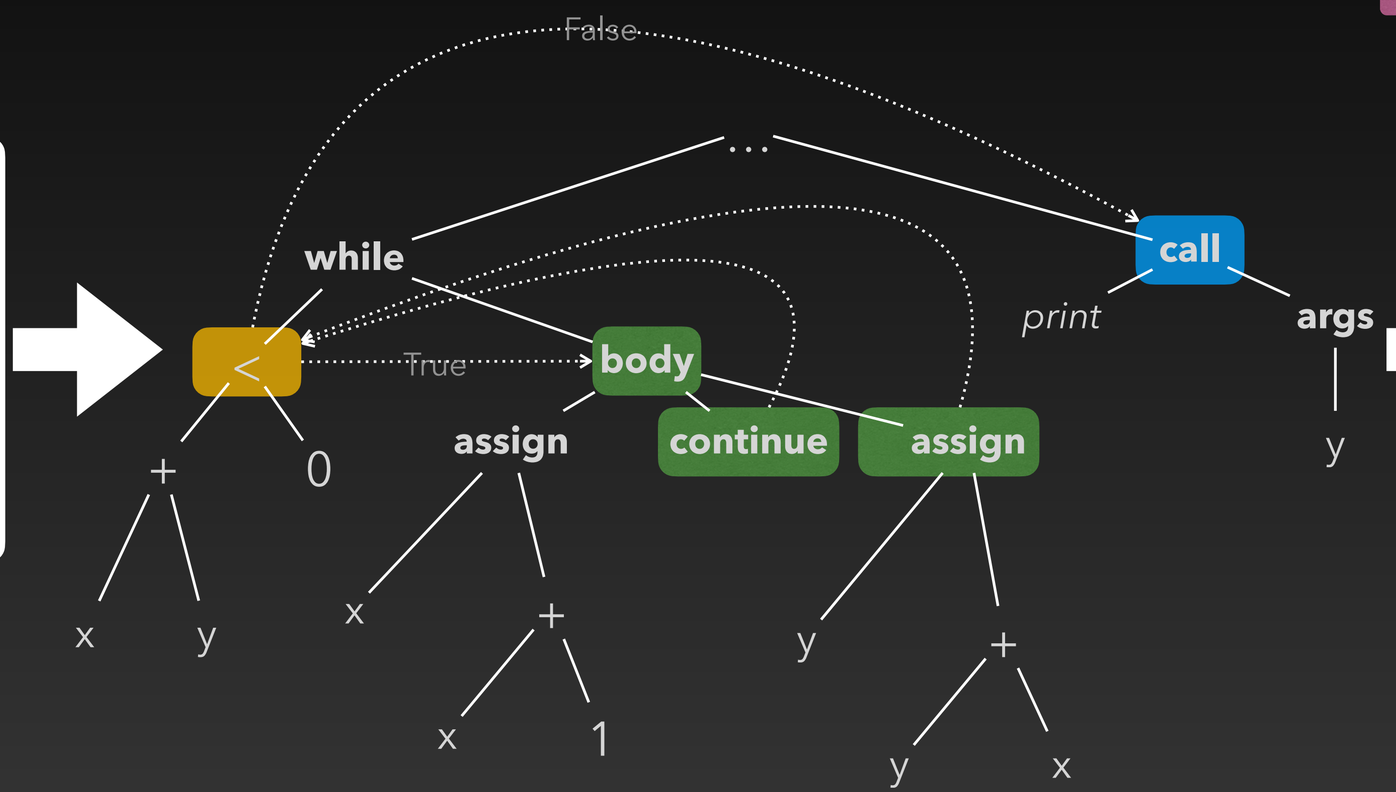


缺点：不紧凑、缺少控制流信息、不利于分析和优化

为什么需要中间码

例子谭添老师的编译原理

```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```



三地址码：接近大多数目标机器的指令，但提供有限的更高的语言特性(比如：支持复合数据、函数)

```
Label L1
    t = x + y
    if t < 0 goto L2
    else goto L3
Label L2
    x = x + 1
    goto L1
    y = y + x
    goto L1;
label L3
    print(y)
```

紧凑、清晰的控制流，易于分析和优化， 与具体语言无关（关联多个前段），与具体目标架构无关（比直接汇编好，关联多个后端）

什么是好的中间码

- Easy translation target（容易从上层翻译获得）
- Easy to translate (容易翻译到下层)
- 接口精简
  - 更少的构造方式意味着后续更简单的phases/optimizations
  - 比如: 源语言可能有**while**, **for**, **foreach**, **do-while**, **do-until loops**, …
    - IR可能只包含**while loops**
    - 翻译的过程会去除for、foreach这样的语法糖

⟦for (pre; cond; post) {body}⟧

=

⟦pre; while(cond) {body; post}⟧

The function ⟦ ⟧ maps objects in it to their meanings. Here, it denotes the “translation” or “compilation”.

LLVM Instruction Set Overview

LLVM 指令集概要

- 指令较为Low-level 并且具有平台无关的语义
  - 类似RISC的三地址码
  - 无限虚拟寄存器
    - 可以支持SSA (Static Single Assignment)
  - 简单，低级的控制流构建方式
  - 带有类型指针的Load/Store指令

```
for (i = 0; i < N; i++) {
    Sum(&A[i], &P);
}
```

```
loop:                                   ; preds = %bb0, %loop
%i.1 = phi i32 [0, %bb0],  [%i.2, %loop]
%AiAddr = getelementptr float* %A, i32 %i.1
call void @Sum(float %AiAddr, %pair* %P)
%i.2 = add i32 %i.1, 1
%exitcond = imp eq i32 %i.1, %N
br i1 %exitcond, label %outloop, label %loop
```

LLVM 指令集概要

- High-level的信息在代码中展露
  - 清晰的数据流信息（通过SSA可以获得）
  - 清晰的控制流图
  - 清晰的独立于语言之外的类型信息
  - 清晰的有类型指针的运算
  - 保留数组下标和结构体的索引等特性

```
for (i = 0; i < N; i++) {
    Sum(&A[i], &P);
}
```

```
loop:                                ; preds = %bb0, %loop
%i.1 = phi i32 [0, %bb0],  [%i.2, %loop]
%AiAddr = getelementptr float* %A, i32 %i.1
call void @Sum(float %AiAddr, %pair* %P)
%i.2 = add i32 %i.1, 1
%exitcond = imp eq i32 %i.1, %N
br i1 %exitcond, label %outloop, label %loop
```

函数调用的语法也保留

源语言中的类型在LLVM中降级了

- 源语言的类型被降级了
  - 丰富的类型系统 –> 简单类型
  - 隐式的、抽象的类型 –> 显式、具体
- 例子：
  - 引用变为指针： T& -> T*
  - 复数： complex float -> {float, float}
  - 结构体的位域(Bitfields)： struct X{int Y: 4; int Z:2; } -> {i32}

源语言中的类型在LLVM中降级了

- LLVM完整的类型系统包含：
  - 原始类型：label, void, float, integer, …
    - 任意的bit长度的整型(i1, i32, i64, i1942652)
  - 派生的：指针，数组，结构体，函数 (没有unions，通过bit cast进行转换)
  - 没有高级的类型
- 类型系统允许类型任意的强制转换

LLVM IR中的函数例子

```
int main(){
  int a = 5;
  int b = 3;
  return a - b;
}
```


```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

Explicit stack allocation

Explicit Loads and Stores

Explicit Types

LLVM IR中的函数例子

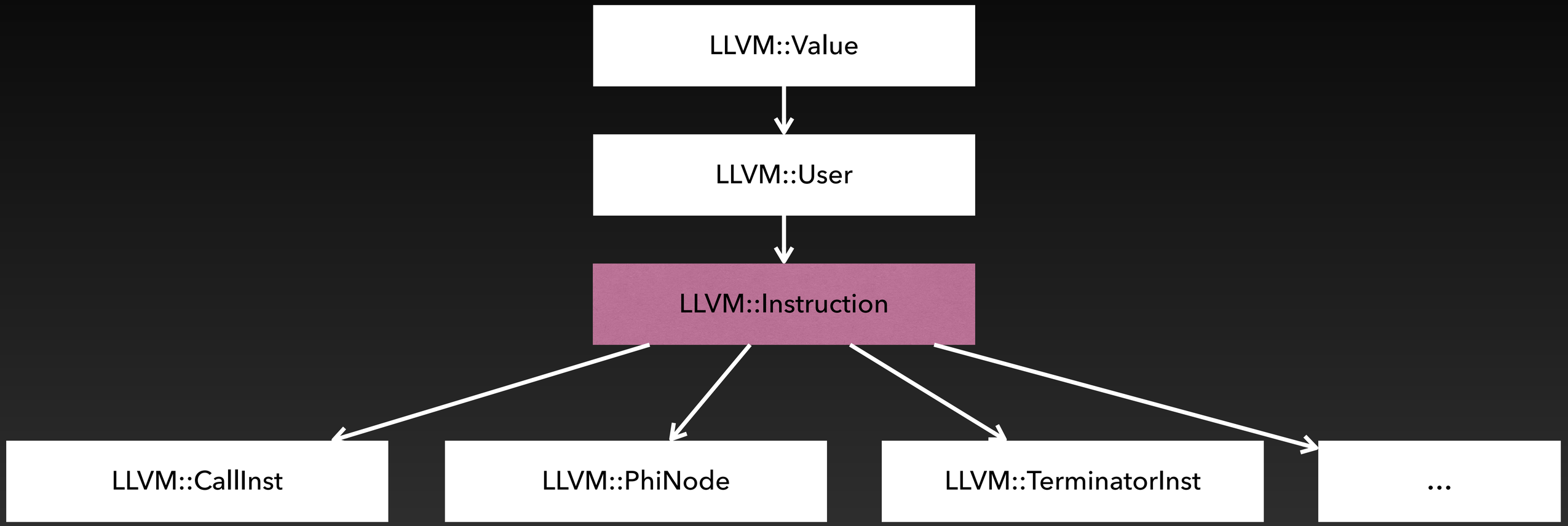
```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```


```
define i32 @main() #0 {
entry:
  %sub = sub nsw i32 5, 3
  ret i32 %sub
}
```

Not always possible:
Sometimes stack operations are too complex


LLVM中间码的处理
LLVM提供了一组内部的API可以对LLVM IR进行分析和处理

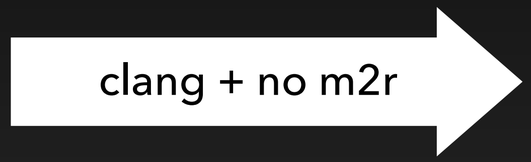
LLVM Instruction Class Hierarchy



- LLVM内部的指令类（用来描述每条IR指令）

LLVM Instructions <--> Values

```
int main(){
    int x;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}
```




```
define i32 @main() #0 {
entry:
    %retval = alloca i32, align 4
    %x = alloca i32, align 4
    %y = alloca i32, align 4
    %z = alloca i32, align 4
    store i32 0, i32* %retval
    store i32 1, i32* %x, align 4
    store i32 2, i32* %y, align 4
    store i32 3, i32* %z, align 4
    %0 = load i32* %y, align 4
    %1 = load i32* %z, align 4
    %add = add nsw i32 %0, %1
    store i32 %add, i32* %x, align 4
    ...
```

LLVM Instructions <--> Values

```
int main(){
    int x;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}
```

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
    %add = add nsw i32 2, 3
    %add1 = add nsw i32 %add, 3
    %add2 = add nsw i32 %add, %add1
    ret i32 0
}
```

Instruction I: %add1 = add nsw i32 %add, 3

You can't "get" %add1 from Instruction I.
Instruction serves as the Value %add1

Operand 1

Operand 2

LLVM Instructions <--> Values

```
int main(){
    int x;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}
```

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
    %add = add nsw i32 2, 3
    %add1 = add nsw i32 %add, 3
    %add2 = add nsw i32 %add, %add1
    ret i32 0
}
```

Instruction I: %add1 = add nsw i32 %add, 3

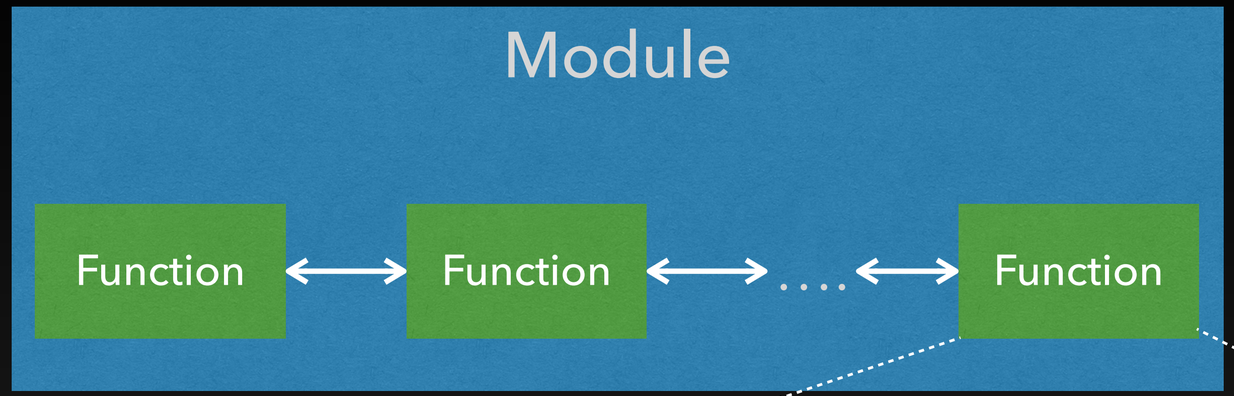
outs() << *(I.getOperand(0)); ⟶ “%add = add nsw i32 2, 3”

outs() << *(I.getOperand(0)->getOperand(0)); ⟶ “2”

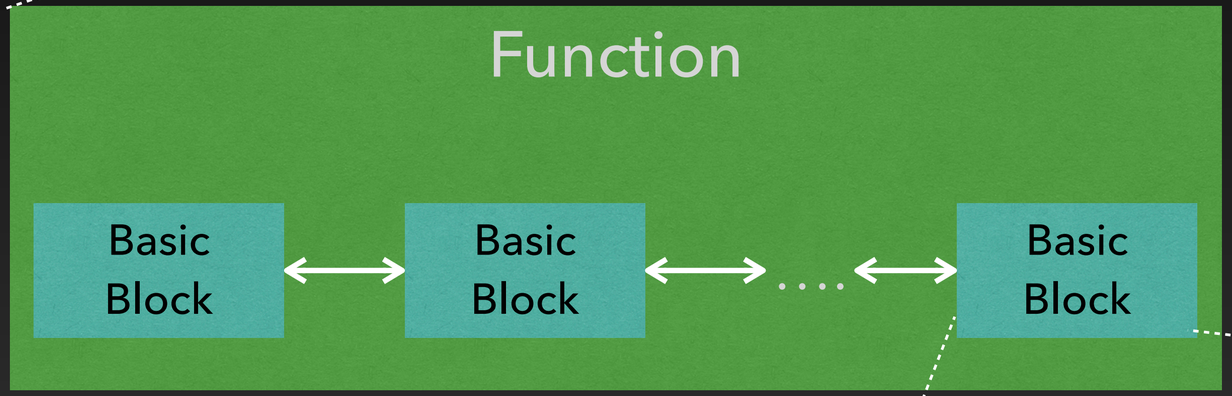
Only makes sense for an SSA Compiler

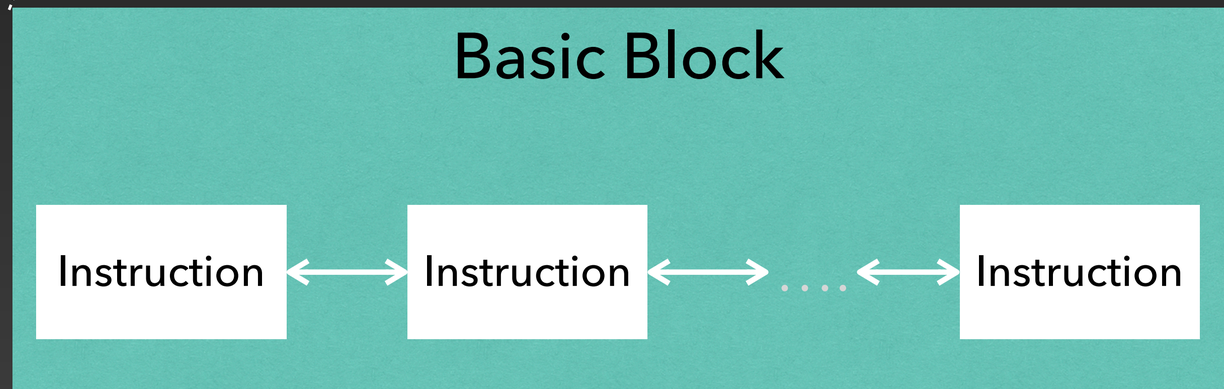
LLVM程序结构

- **模块(Module)** 包含**函数(Functions)**和**全局变量(GlobalVariables)**
- **函数(Function)** 包含**基本块(BasicBlocks)**和
  - 函数就是和C语言里的函数类似
- **基本块(BasicBlock)**包含了一组**指令(Instructions)**
  - 每个基本块都终止于一个控制流指令
- **指令(Instruction)**就是一个**操作符(opcode)**加上一组**操作数(operands)**
  - 每个操作数都有类型
  - 返回的指令也是有类型的



LLVM IR内部的数据结构往往是双向链表组织起来的，可以通过迭代进行遍历





LLVM也支持访问者模式（Visitor Pattern）对这些数据结构进行遍历和处理

Navigating the LLVM IR - Iterators

- Module::iterator
  - 模块是大的分析目标，该迭代器会遍历其中的所有函数
- Function::iterator
  - 遍历函数中的所有基本块
- BasicBlock::iterator
  - 遍历基本块中的每个指令
- Value::use_iterator
  - 遍历**使用**该指令Value的所有其他指令
- User::op_iterator
  - 遍历指令的每一个操作数

Navigating the LLVM IR - Iterators

- 遍历一个函数中的每个指令：

一个C++程序，输入是一个LLVM IR的文件

```
for (Function::iterator FI = func->begin(); FI != func->end(); ++FI) {
    for (BasicBlock::iterator BBI = FI->begin(); BBI != FI->end(); ++BBI) {
        outs() << “Instruction: ” << *BBI << “\n”;
    }
}
```

Navigating the LLVM IR - Iterators

- 也可以直接使用 InstIterator (provided by "llvm/IR/InstIterator.h“):

```
#include “llvm/IR/InstIterator.h”

for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    outs() << *I << “\n”;
}
```

Navigating the LLVM IR - Iterators

- 找到一个基本块的前驱/后继

```
#include "llvm/Support/CFG.h" BasicBlock *BB = ...;

for (pred_iterator PI = pred_begin(BB); PI != pred_end(BB); ++PI) {
    BasicBlock *pred = *PI;
    // ...
}
```

Navigating the LLVM IR - Iterators

- Very common code pattern: casting and Iterators

```
for (Function::iterator FI = func->begin(); FI != func->end(); ++FI) {
    for (BasicBlock::iterator BBI = FI->begin(); BBI != FI->end(); ++BBI) {
        Instruction * I = BBI;
        if (CallInst * CI = dyn_cast<CallInst>(I)) {
            outs() << "I'm a Call Instruction!\n";
        }
        if (UnaryInstruction * UI = dyn_cast<UnaryInstruction>(I)) {
            outs() << "I'm a Unary Instruction!\n";
        }
        if (CastInstruction * CI = dyn_cast<CastInstruction>(I)) {
            outs() << "I'm a Cast Instruction!\n";
        }
        ...
    }
}
```

cast 类型转换，如果是目标类型，转换成功，否则full

Navigating the LLVM IR – Visitor Pattern

```
class MyVisitor : public InstVisitor<MyVisitor> {
    void visitCallInst(CallInst &CI) {
        outs() << "I'm a Call Instruction!\n";
    }
    void visitUnaryInstruction(UnaryInstruction &UI) {
        outs() << "I'm a Unary Instruction!\n";
    }
    void visitCastInst(CastInst &CI) {
        outs() << "I'm a Cast Instruction!\n";
    }
    void visitBinaryOperator(BinaryOperator &I) {
        switch (I.getOpcode()) {
        case Instruction::Mul:
            outs() << "I'm a multiplication Instruction!\n";
        }
    }
}
```

访问者模式让我们不必显示的遍历
一个给定的instruction只会触发一个vist方法!

改变IR：写passes

- 删除instructions：
  - **eraseFromParent()**
    - Remove from basic block, drop all references, delete
  - **removeFromParent()**
    - Remove from basic block
    - Use if you will re-attach the instruction
    - Does not drop references (or clear the use list), so if you don't re-attach it Bad Things will happen
  - **moveBefore/insertBefore/insertAfter** are also available
  - **replaceInstWithValue** and **replaceInstWithInst** are also useful to have

Writing Passes – Adding New Instructions

```
define i32 @main() #0 {
entry:
      %add = add nsw i32 2, 2
      %add1 = add nsw i32 %add, 2
      %mul = mul nsw i32 %add, %add1
      %sub = sub nsw i32 %add1, %mul
      %add2 = add nsw i32 %mul, 5
      ret i32 %sub
}
```

```
define i32 @main() #0 {
entry:
      %add = add nsw i32 2, 2
      %add1 = add nsw i32 %add, 2
      %mul = mul nsw i32 %add, %add1
      %0 = add i32 %add,  0
      %sub = sub nsw i32 %add1, %mul
      %add2 = add nsw i32 %mul, 5
      ret i32 %sub
}
```

```
Instruction *I = “%mul = mul nsw i32 %add, %add1”;
Instruction *newInst = BinaryOperator::Create(Instruction::Add, I.getOperand(0),
ConstantInt::get(I.getOperand(0)->getType(), 0));
I->getParent()->getInstList().insert(I, newInst)
```

LLVM的Pass(趟)管理

- 编译器的流程可以划分为一系列的趟（Pass）
  - 每个Pass做一类分析（Analysis）或代码变换（Transformation）
  - 每个Pass需要做的具体工作取决于上一个Pass的处理结果
- 一些有用的Passes类型
  - **BasicBlockPass**: 这个类型的Pass对基本块进行迭代处理
  - **CallGraphSCCPass:** 自底向上对Call graph （以SCC形式）进行迭代
  - **FunctionPass**: 对函数（没有特定顺序）进行迭代处理
  - …

LLVM的一些工具

- llvm-dis: Convert from .bc (IR binary) to .ll (human-readable IR text)
- llvm-as: Convert from .ll (human-readable IR text) to .bc (IR binary)
- opt: LLVM optimizer
- llc: LLVM static compiler
- lli: LLVM bitcode interpreter
- llvm-link: LLVM bitcode linker
- bugpoint - automatic test case reduction tool
- llvm-extract - extract a function from an LLVM module
- llvm-bcanalyzer - LLVM bitcode analyzer

一些有用的LLVM学习文档

- LLVM Coding Standards: http://llvm.org/docs/CodingStandards.html
- LLVM Programmer's Manual: http://llvm.org/docs/ProgrammersManual.html
- LLVM Language Reference Manual : http://llvm.org/docs/LangRef.html
- Writing an LLVM Pass: http://llvm.org/docs/WritingAnLLVMPass.html
- LLVM's Analysis and Transform Passes: http://llvm.org/docs/Passes.html
- LLVM Internal Documentation: http://llvm.org/docs/doxygen/html/
- 一个好的LLVM中文简介推荐：https://evian-zhang.github.io/llvm-ir-tutorial/


Q&A