

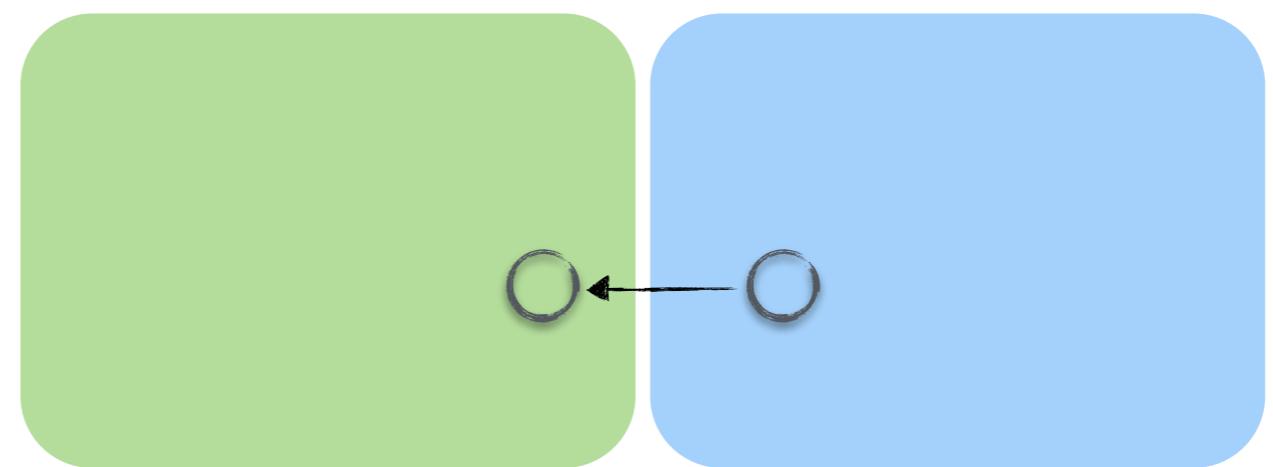
# Analysing Native Code

Applied Static Analysis 2016

**Ben Hermann**  
**@benhermann**

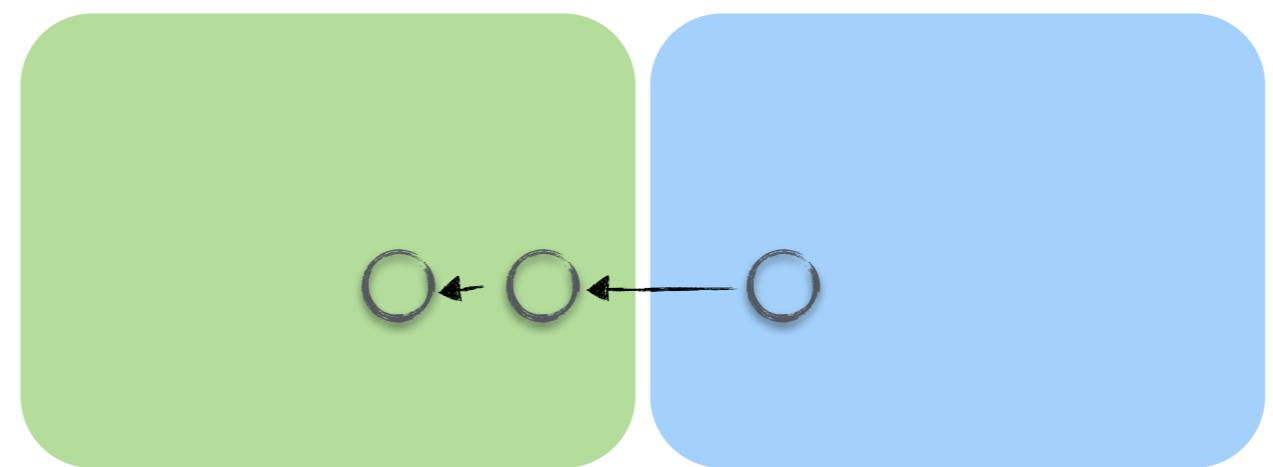
Dr. Michael Eichberg, Johannes Lerch, Sebastian Proksch, Karim Ali Ph.D.





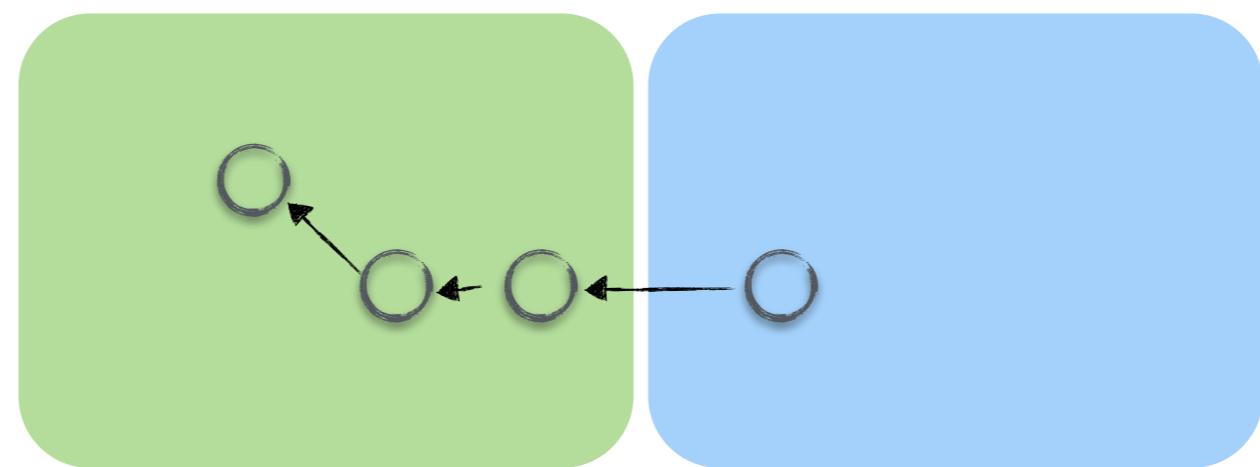
JCL Code

User Code



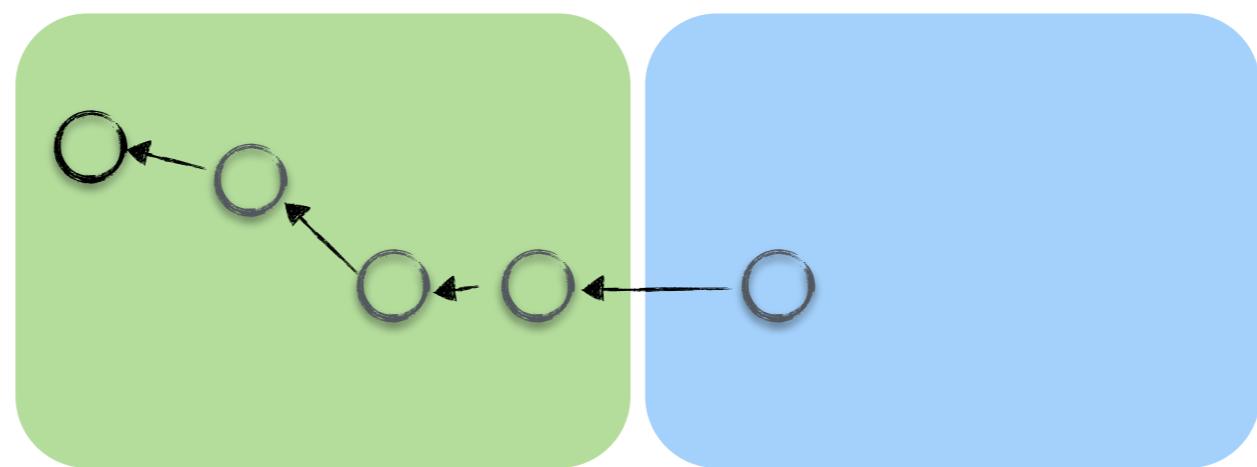
JCL Code

User Code



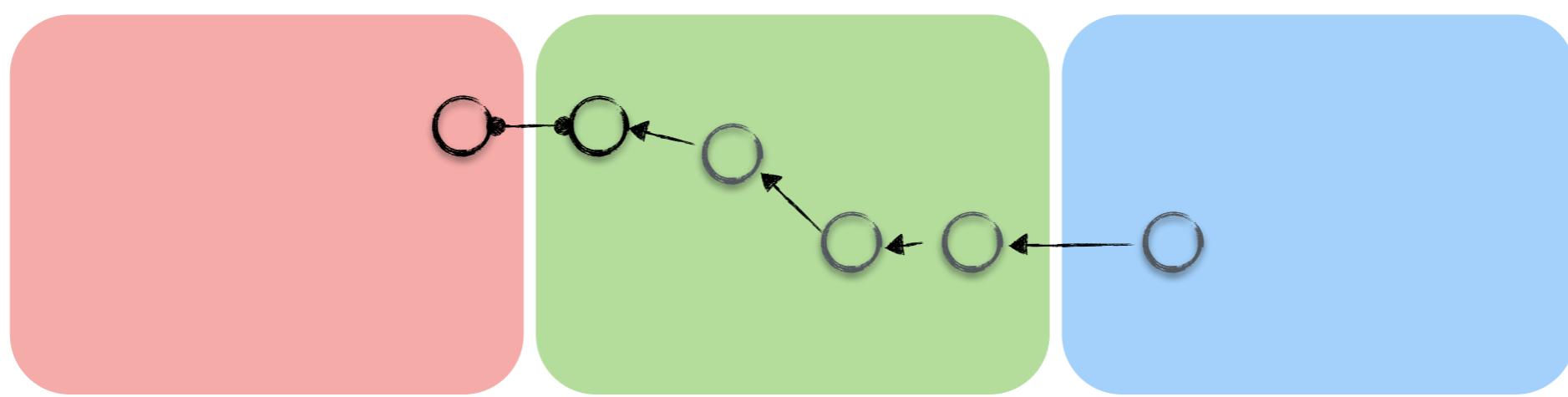
JCL Code

User Code



JCL Code

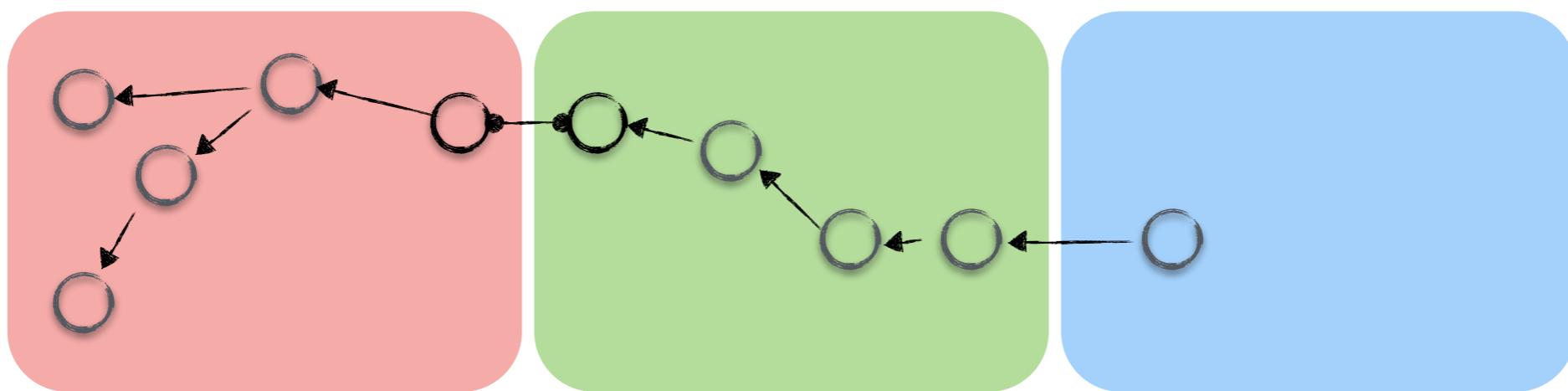
User Code



JCL  
Native Code

JCL Code

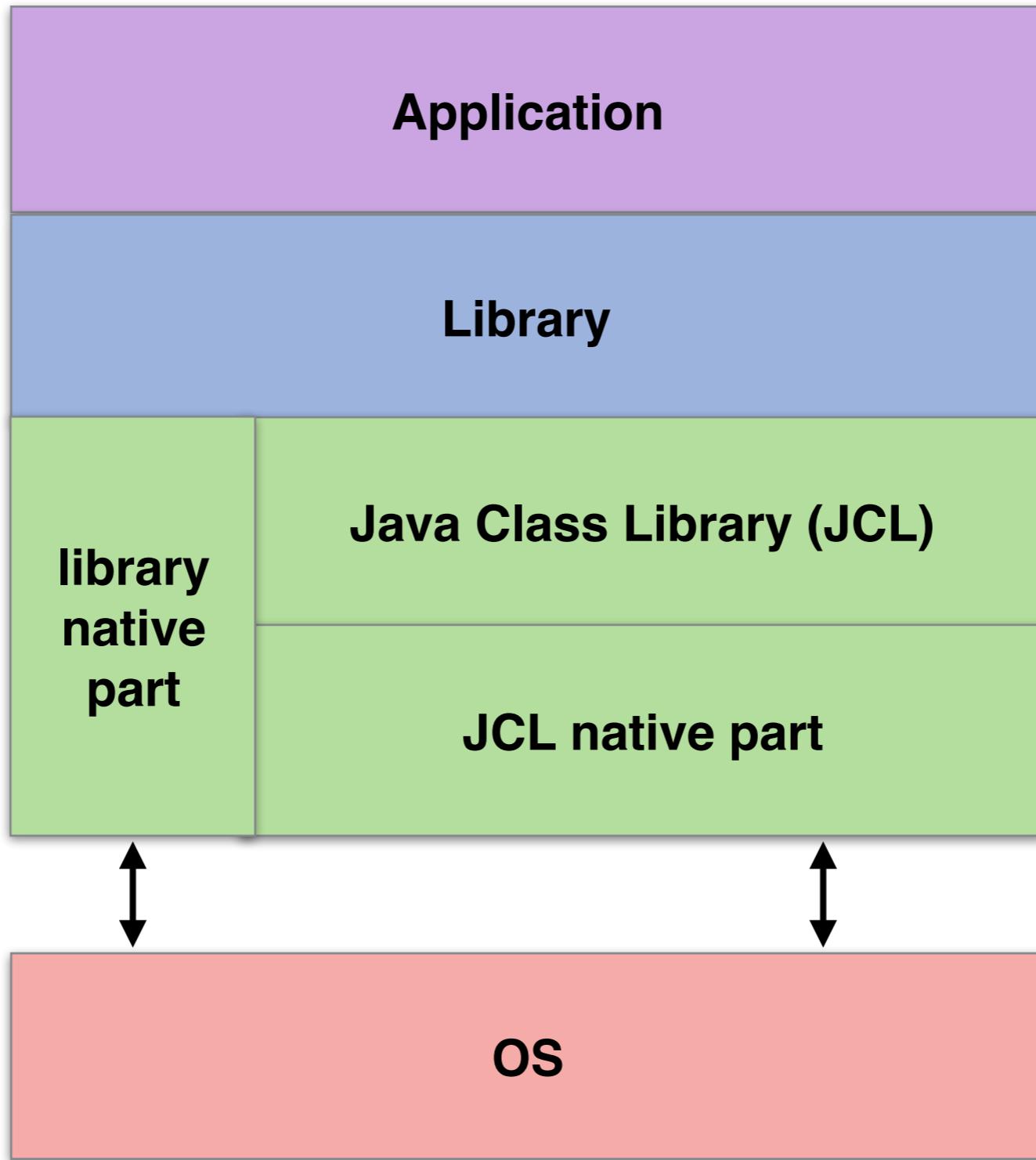
User Code



JCL  
Native Code

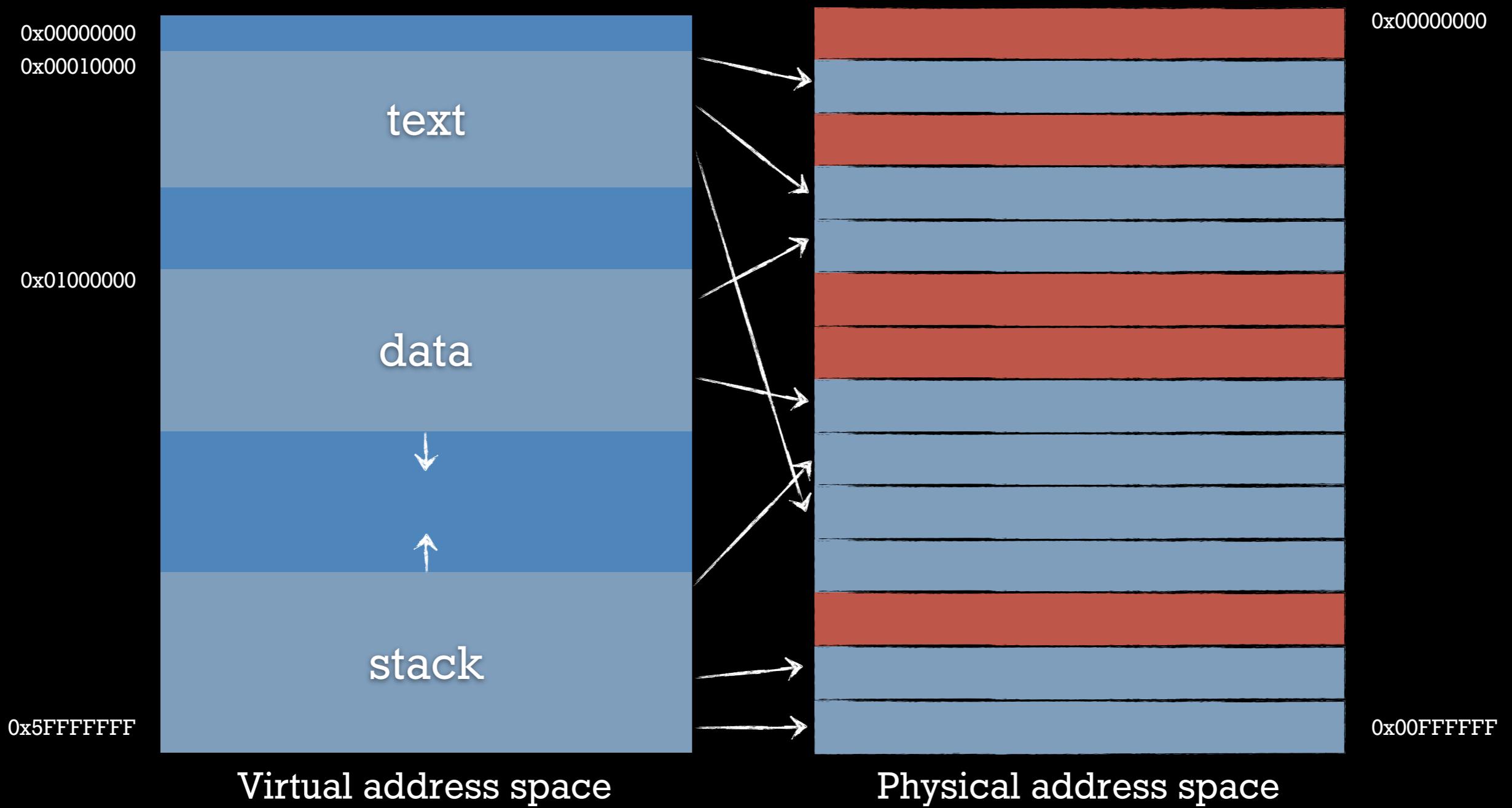
JCL Code

User Code



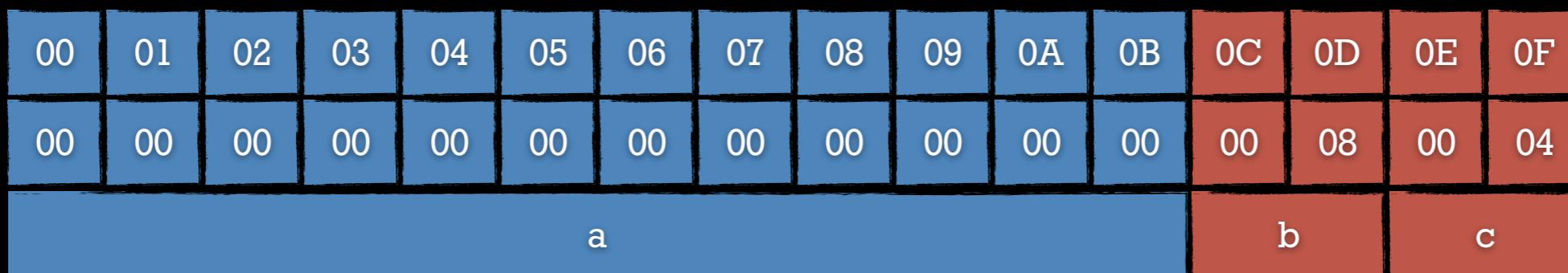


# Process isolation & virtual memory



# Buffer overflows

```
char a[12];
unsigned int b = 8;
unsigned int c = 4;
```



# Buffer overflows

```
char a[12];
unsigned int b = 8;
unsigned int c = 4;
```

```
strcpy(a, "test");
```

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	00	00	00	00	00	00	00	00	00	00	08	00	04
a												b	c		

# Buffer overflows

```
char a[12];
unsigned int b = 8;
unsigned int c = 4;
```

```
strcpy(a, "test");
```

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
74	65	73	74	00	00	00	00	00	00	00	00	00	08	00	04
a										b		c			

# Buffer overflows

```
char a[12];
unsigned int b = 8;
unsigned int c = 4;
```

```
strcpy(a, "test");
```

```
strcpy(a, "muchtoolongfora");
```

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
74	65	73	74	00	00	00	00	00	00	00	00	00	08	00	04
a												b	c		

# Buffer overflows

```
char a[12];  
unsigned int b = 8;  
unsigned int c = 4;
```

```
strcpy(a, "test");
```

```
strcpy(a, "muchtoolongfora");
```



**Please all stand up for a moment**

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

I've never used a console

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

I've never used a console

I don't know what C and C++ are

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

I've never used a console

I don't know what C and C++ are

I have never read any C or C++ code

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

I've never used a console

I don't know what C and C++ are

I have never read any C or C++ code

I have never written any C or C++ code

**Please all stand up for a moment**

**Sit down, when the following is true for you:**

I've never used a console

I don't know what C and C++ are

I have never read any C or C++ code

I have never written any C or C++ code

I hate writing code in C or C++

# Why is this different?

- Direct memory access
- Pointer arithmetic
- All kinds of unsafe programming
- C code might behave differently on different machines

# Why should I care?

- Because your Java program calls into native.
- You use native code all the time ... and you want to in order to have any effect on the outside world other than using energy and heating up the planet.
- Also other people do that for you
  - > 70% of the JCL methods do that
  - Many libraries use `sun.misc.Unsafe`

# sun.misc.What?!

```
package sun.misc
public final class Unsafe

public native long allocateMemory(long bytes)

public void copyMemory(long srcAddress,
                      long destAddress,
                      long bytes)

public native int getInt(long address)
```

# Is this used?

- Oh yes it is...
- 3% of the top 1,000 maven artifacts directly
- 47% of them use it transitively

```
// crashing your JVM in one line  
unsafe.freeMemory(1);
```

Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. **Use at your own risk: the Java unsafe API in the wild.** In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)

So... yes you should care  
about the native layer  
when analyzing Java code

# Why are the slides black?

# Why are the slides black?

- Because your Unix console is too

# Why are the slides black?

- Because your Unix console is too
- To show you that you are dealing with a different beast now

# Why are the slides black?

- Because your Unix console is too
- To show you that you are dealing with a different beast now
- Looks much better on the projector

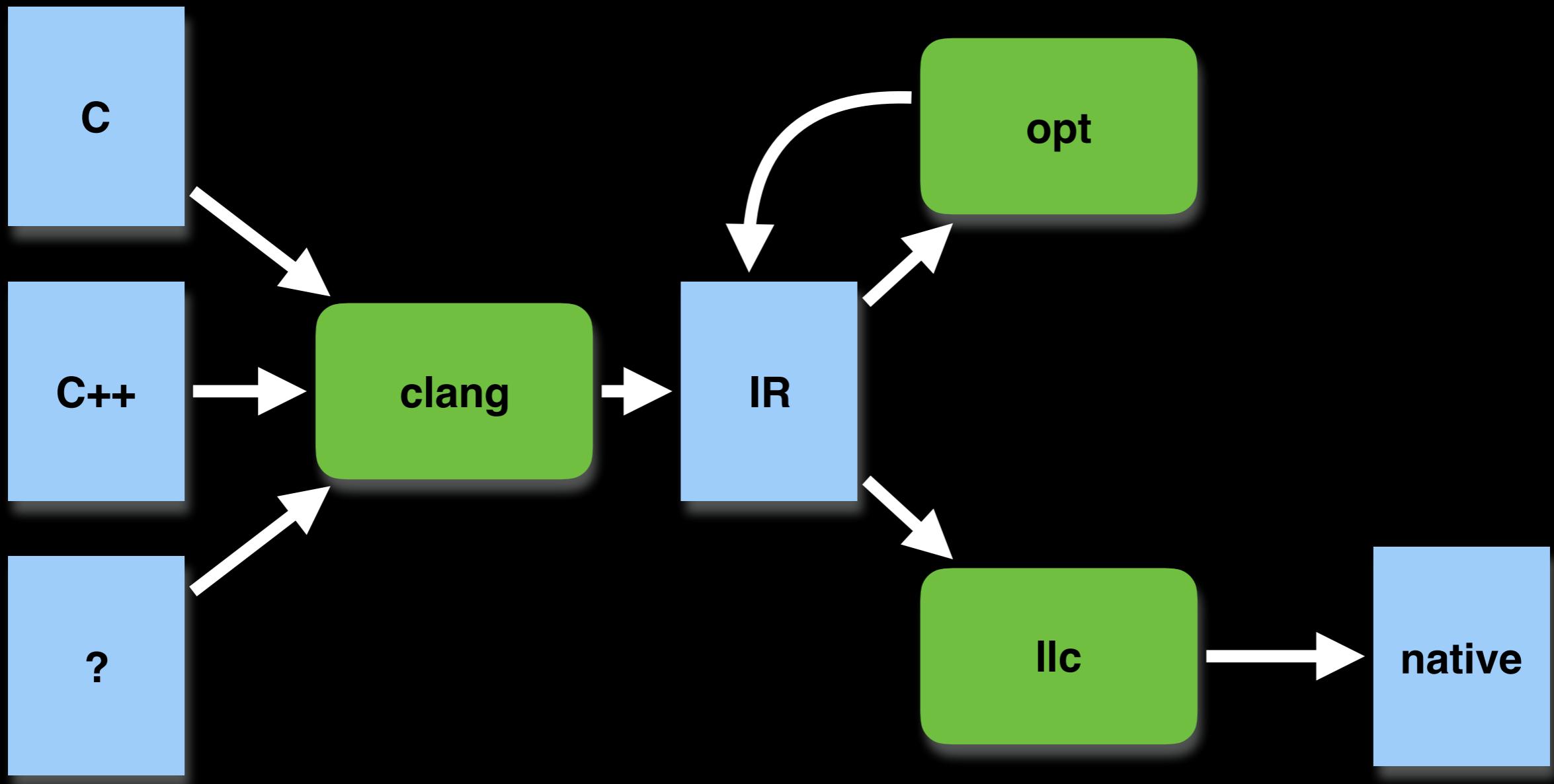
# Why are the slides black?

- Because your Unix console is too
- To show you that you are dealing with a different beast now
- Looks much better on the projector
- **But ... there will also be a printable version of the slides in the repository**

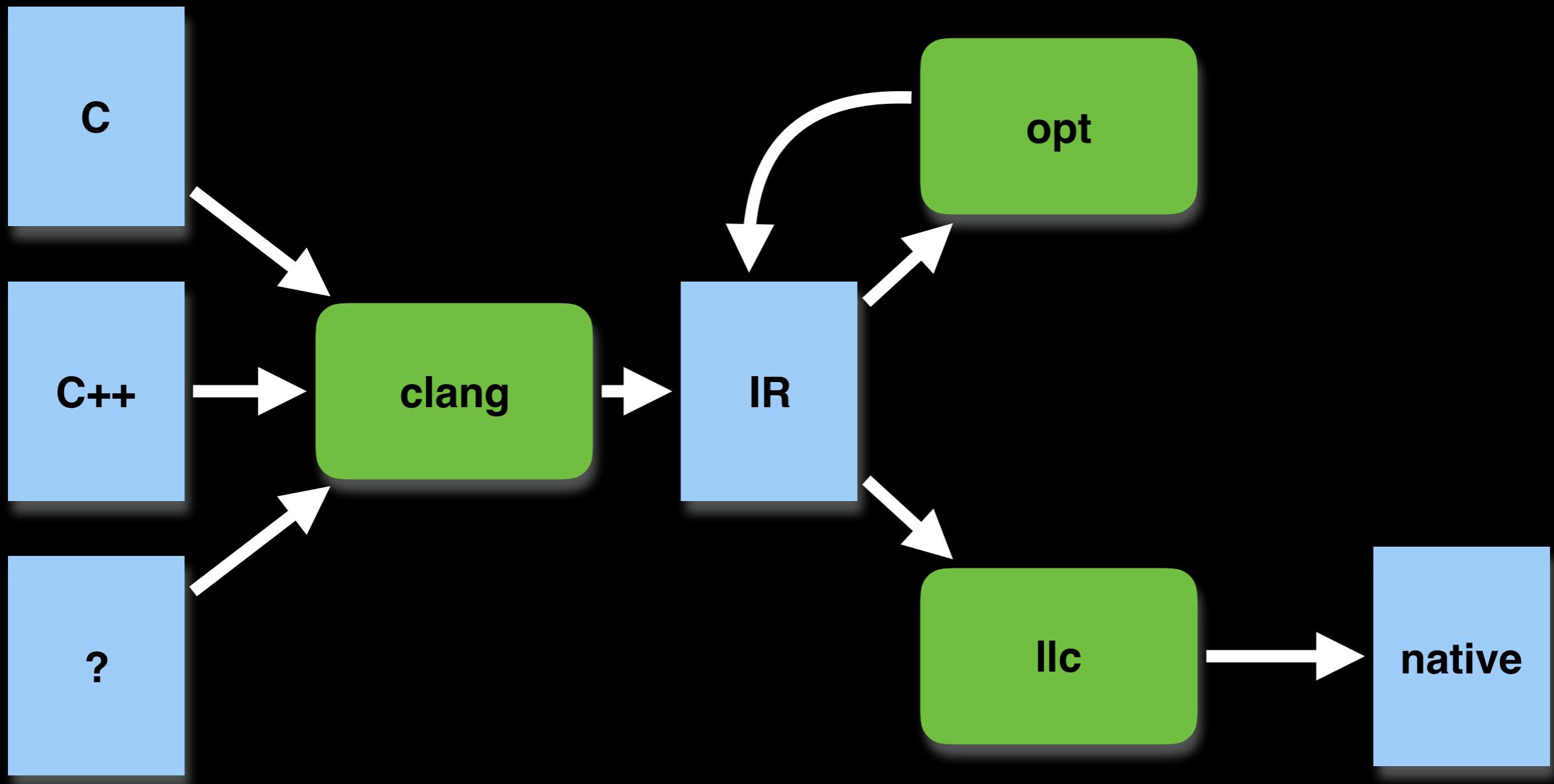
# LLVM

- Compiler Infrastructure
- Lightweight
- Modular
- Reusable
- SSA-based

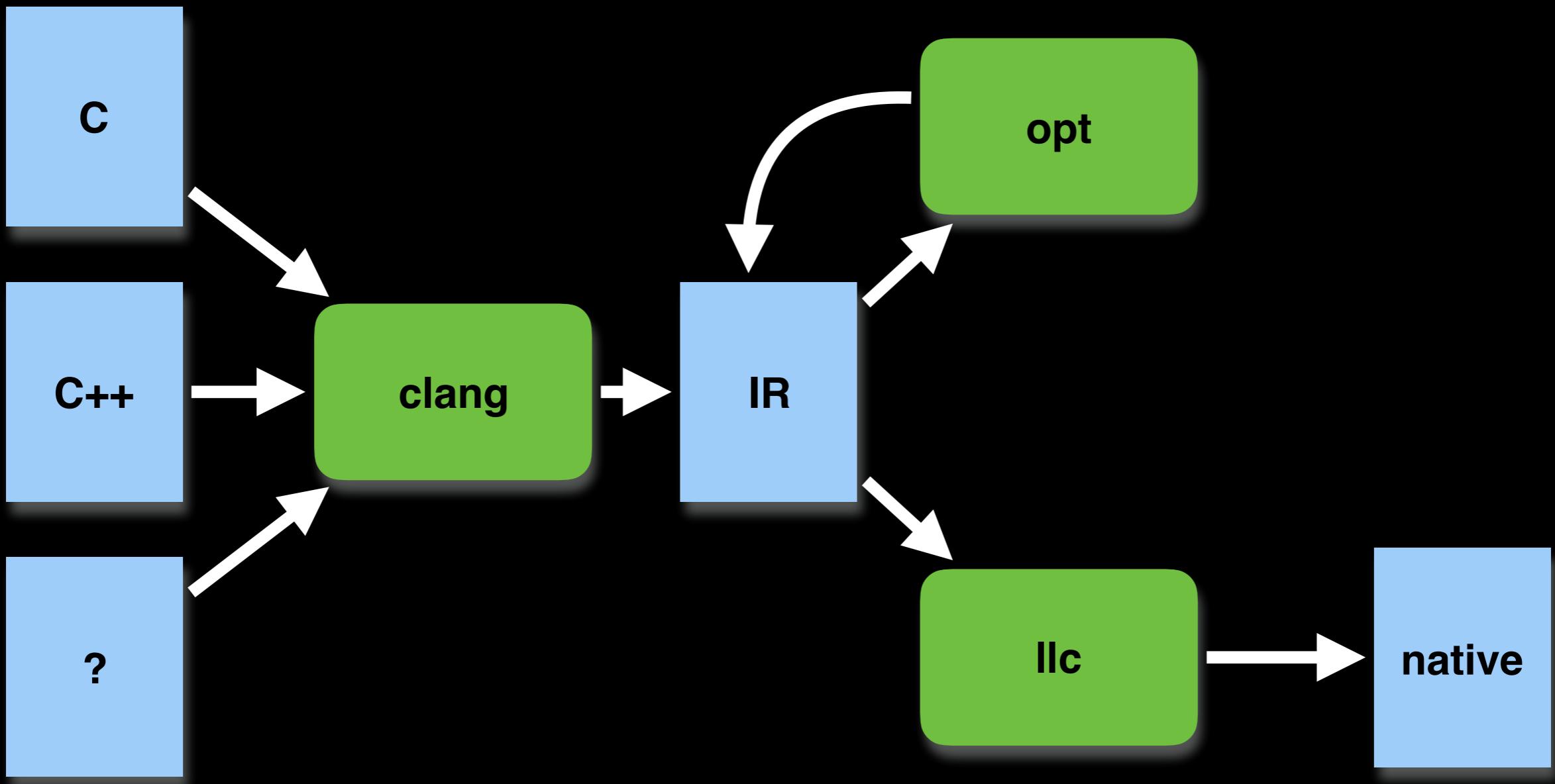
# LLVM Components



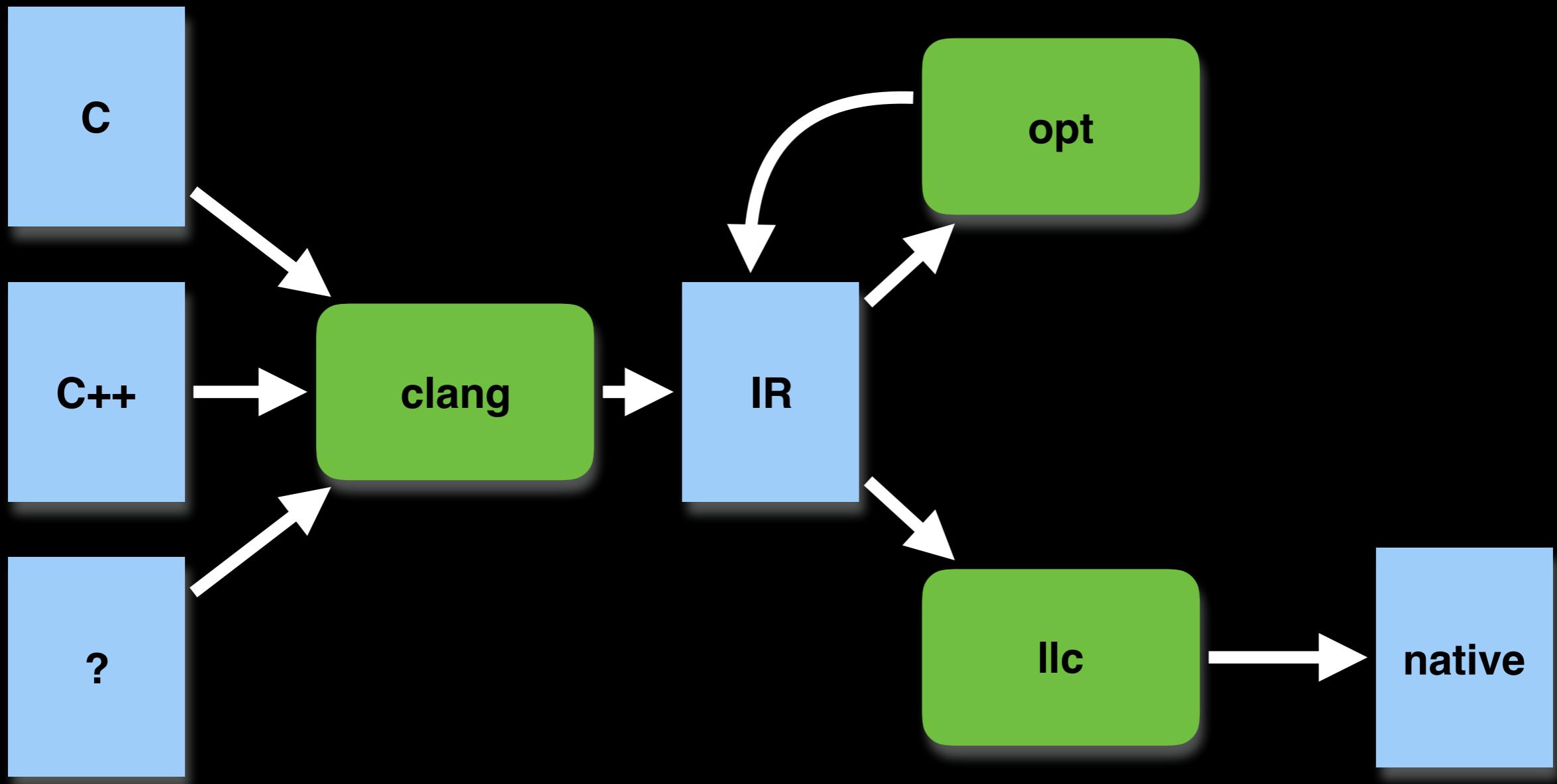
# LLVM Components



# LLVM Components



# LLVM Components



# Preparing for this Lecture

- Lose any fear towards a UNIX console NOW
- Install LLVM on your machine

```
wget http://llvm.org/releases/3.8.0/llvm-3.8.0.src.tar.xz  
tar xvfz llvm-3.8.0.src.tar.xz  
cd llvm-3.8.0.src  
cp Makefile.config.in Makefile.config  
mkdir build  
cd build  
cmake -g "Unix Makefiles" ../  
make
```

- Get a coffee.... Or maybe five.

# Preparing for this Lecture

- Lose any fear towards a UNIX console NOW
- Install LLVM on your machine

```
wget http://llvm.org/releases/3.8.0/llvm-3.8.0.src.tar.xz  
tar xvfz llvm-3.8.0.src.tar.xz  
cd llvm-3.8.0.src  
cp Makefile.config.in Makefile.config  
mkdir build  
cd build  
cmake -g "Unix Makefiles" ../  
make
```

It should also work with Ninja,  
Visual Studio, or XCode  
— I use `make` here —

- Get a coffee.... Or maybe five.

exercise 6.1a

# Preparing for this Lecture

- Now we've got LLVM, but we will also need clang

```
wget http://llvm.org/releases/3.8.0/cfe-3.8.0.src.tar.xz  
tar xfz cfe-3.8.0.src.tar.xz  
cd cfe-3.8.0.src  
cmake -g "Unix Makefiles" ../llvm-3.8.0.src  
make
```
- Get a coffee.... Or tea... there's always tea

# Or use our Docker Container

```
docker run --rm -ti bhermann/llvm
```



- Time for coffee depends on your download speed
- Doing this right now will probably not work given the classroom's bandwidth

# Working with the container

Stay updated

```
docker pull bhermann/llvm
```

Having something persistent

```
docker run -ti -v /your/path:/home/llvm/workspace bhermann/llvm
```



# Factorial Example

```
int factorial (int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

PC	Instruction
0	iconst_1
1	istore_1
2	goto →12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc (lv=0,val=-1)
12	iload_0
13	ifgt →5
16	iload_1
17	ireturn

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
    %1 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %n, i32* %1, align 4
    store i32 1, i32* %r, align 4
    br label %2

; <label>:2 ; preds = %5, %0
    %3 = load i32* %1, align 4
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
    %6 = load i32* %1, align 4
    %7 = load i32* %r, align 4
    %8 = mul nsw i32 %7, %6
    store i32 %8, i32* %r, align 4
```

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0
%1 = alloca i32, align 4
%r = alloca i32, align 4
store i32 %n, i32* %1, align 4
store i32 1, i32* %r, align 4
br label %2

; <label>:2
%3 = load i32* %1, align 4
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %11 ; preds = %5, %0

; <label>:5
%6 = load i32* %1, align 4
%7 = load i32* %r, align 4
%8 = mul nsw i32 %7, %6 ; preds = %2
store i32 %8, i32* %r, align 4
```

## Data Layout (dash separated)

little-endian

ELF mangling

64-bit integers

80 to 128-bit floats

native CPU integers are 8, 16, 32, and 64 -bit

Stack alignment is 128-bits

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
    %1 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %n, i32* %1, align 4
    store i32 1, i32* %r, align 4
    br label %2

; <label>:2 ; preds = %5, %0
    %3 = load i32* %1, align 4
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
    %6 = load i32* %1, align 4
    %7 = load i32* %r, align 4
    %8 = mul nsw i32 %7, %6
    store i32 %8, i32* %r, align 4
```

# LLVM Representation

```
; ModuleID = 'factorial.c'  
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"  
target triple = "x86_64-pc-linux-gnu"
```

```
; Function Attrs: nounwind uwtable  
define i32 @factorial(i32 %n) #0  
%1 = alloca i32, align 4  
%r = alloca i32, align 4  
store i32 %n, i32* %1, align 4  
store i32 1, i32* %r, align 4  
br label %2
```

```
; <label>:2 ; preds = %5, %0  
%3 = load i32* %1, align 4  
%4 = icmp sgt i32 %3, 0  
br i1 %4, label %5, label %11
```

```
; <label>:5 ; preds = %2  
%6 = load i32* %1, align 4  
%7 = load i32* %r, align 4  
%8 = mul nsw i32 %7, %6  
store i32 %8, i32* %r, align 4
```

## Target Triple (dash separated)

Architecture (e.g., x86\_64, ARM, PowerPC)

Vendor (e.g., pc, apple)

Operating System (e.g., linux, macosx10.7.0)

Environment (e.g., gnu)

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
    %1 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %n, i32* %1, align 4
    store i32 1, i32* %r, align 4
    br label %2

; <label>:2 ; preds = %5, %0
    %3 = load i32* %1, align 4
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
    %6 = load i32* %1, align 4
    %7 = load i32* %r, align 4
    %8 = mul nsw i32 %7, %6
    store i32 %8, i32* %r, align 4
```

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
    %1 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %n, i32* %1, align 4
    store i32 1, i32* %r, align 4
    br label %2

; <label>:2
    %3 = load i32* %1, align 4
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %11

; <label>:5
    %6 = load i32* %1, align 4
    %7 = load i32* %r, align 4
    %8 = mul nsw i32 %7, %6
    store i32 %8, i32* %r, align 4
    ; preds = %2
```

## Function Definition

Return type (here: i32 → 32-bit Integer)  
Function name (@-prefixed → global identifier)  
Argument list (%-prefixed → local identifier)  
Attribute reference

# LLVM Representation

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
    %1 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %n, i32* %1, align 4
    store i32 1, i32* %r, align 4
    br label %2

; <label>:2 ; preds = %5, %0
    %3 = load i32* %1, align 4
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
    %6 = load i32* %1, align 4
    %7 = load i32* %r, align 4
    %8 = mul nsw i32 %7, %6
    store i32 %8, i32* %r, align 4
```

```

%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
%6 = load i32* %1, align 4
%7 = load i32* %r, align 4
%8 = mul nsw i32 %7, %6
store i32 %8, i32* %r, align 4
%9 = load i32* %1, align 4
%10 = add nsw i32 %9, -1
store i32 %10, i32* %1, align 4
br label %2

; <label>:11 ; preds = %2
%12 = load i32* %r, align 4
ret i32 %12
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointe
r-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-na
ns-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "
use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{"Ubuntu clang version 3.6.0-2ubuntu1~trusty1 (tags/RELEASE_360/final) (b
ased on LLVM 3.6.0)"}

```

# Down to Machine Code

```
.text
.file "factorial.c"
.globl factorial
.align 16, 0x90
.type factorial,@function
factorial:                      # @factorial
    .cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    movl %edi, -4(%rbp)
    movl $1, -8(%rbp)
.LBB0_1:                          # =>This Inner Loop Header: Depth=1
    cmpl $0, -4(%rbp)
    jle .LBB0_3
# BB#2:                           # in Loop: Header LBB0_1 Depth 1
```

```

    movq    %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
    cmpl    $0, -4(%rbp)
    jle .LBB0_3
# BB#2:                                     #   in Loop: Header=BB0_1 Depth=1
    movl    -4(%rbp), %eax
    imull  -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    $4294967295, %eax      # imm = 0xFFFFFFFF
    movl    %eax, -4(%rbp)
    jmp .LBB0_1
.LBB0_3:
    movl    -8(%rbp), %eax
    popq    %rbp
    retq
.Ltmp3:
    .size   factorial, .Ltmp3-factorial
    .cfi_endproc

```

```

.ident "Ubuntu clang version 3.6.0-2ubuntu1~trusty1 (tags/RELEASE_360/final)
(based on LLVM 3.6.0)"
.section ".note.GNU-stack","",@progbits

```

# Exercises & Examples

- In our course Git repository, `llvm` directory
- `examples` contains all examples discussed here
- `exercises` contains all exercises
- `workspace` can be used as your personal area
- And if you have the Docker container you already have them in your home directory
- Please perform all commands following and inspect the output closely

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

Only run preprocess  
and compilation steps  
(outputs human  
readable format)

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```



Use LLVM IR

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```



Input file

```
clang -cc1 -ast-dump factorial.c
```

exercise 6.2

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

Output to stdout

```
clang -cc1 -ast-dump factorial.c
```

exercise 6.2

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

Use only compiler  
front-end

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

Dump the complete  
AST on the console

```
clang -cc1 -ast-dump factorial.c
```

# Basic Tooling

- Familiarize yourself with the tooling

```
clang -S -emit-llvm factorial.c -o -
```

```
clang -cc1 -ast-dump factorial.c
```

# Abstract Syntax Tree

```
TranslationUnitDecl 0x2f3d6f0 <><> <><>
|-TypedefDecl 0x2f3dbf0 <><> <><> implicit __int128_t '__int128'
|-TypedefDecl 0x2f3dc50 <><> <><> implicit __uint128_t 'unsigned __int128'
|-TypedefDecl 0x2f3dfa0 <><> <><> implicit __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x2f3e0c0 <factorial.c:1:1, line:8:1> line:1:5 factorial 'int (int)'
 | -ParmVarDecl 0x2f3e000 <col:16, col:20> col:20 used n 'int'
`-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
 |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
   `-VarDecl 0x2f3e180 <col:2, col:10> col:6 used r 'int' cinit
     `-IntegerLiteral 0x2f3e1d8 <col:10> 'int' 1
`-WhileStmt 0x2f3e3a8 <line:3:2, line:6:2>
 |-<<<NULL>>>
`-BinaryOperator 0x2f3e270 <line:3:9, col:13> 'int' '>'
  |-ImplicitCastExpr 0x2f3e258 <col:9> 'int' <LValueToRValue>
    `-DeclRefExpr 0x2f3e210 <col:9> 'int' lvalue ParmVar 0x2f3e000 'n' 'int'
    `-IntegerLiteral 0x2f3e238 <col:13> 'int' 0
`-CompoundStmt 0x2f3e380 <col:16, line:6:2>
  |-CompoundAssignOperator 0x2f3e300 <line:4:3, col:8> 'int' '*=' ComputeLHSTy='int'
ComputeResultTy='int'
  |-DeclRefExpr 0x2f3e298 <col:3> 'int' lvalue Var 0x2f3e180 'r' 'int'
  `-ImplicitCastExpr 0x2f3e2e8 <col:8> 'int' <LValueToRValue>
    `-DeclRefExpr 0x2f3e2c0 <col:8> 'int' lvalue ParmVar 0x2f3e000 'n' 'int'
  `-UnaryOperator 0x2f3e360 <line:5:3, col:4> 'int' postfix '--'
    `-DeclRefExpr 0x2f3e338 <col:3> 'int' lvalue ParmVar 0x2f3e000 'n' 'int'
`-ReturnStmt 0x2f3e410 <line:7:2, col:9>
  `-ImplicitCastExpr 0x2f3e3f8 <col:9> 'int' <LValueToRValue>
    `-DeclRefExpr 0x2f3e3d0 <col:9> 'int' lvalue Var 0x2f3e180 'r' 'int'
```

# Abstract Syntax Tree

```
TranslationUnitDecl 0x2f3d6f0 <> <>
|-TypedefDecl 0x2f3dbf0 <> <> implicit __int128_t '__int128'
|-TypedefDecl 0x2f3dc50 <> <> implicit __uint128_t 'unsigned __int128'
|-TypedefDecl 0x2f3dfa0 <> <> implicit __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x2f3e0c0 <factorial.c:1:1, line:8:1> line:1:5 factorial 'int (int)'
  |-ParmVarDecl 0x2f3e000 <col:16, col:20> col:20 used n 'int'
`-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
  |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
    `-
  |-IfStmt 0x2f3e200 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-ForStmt 0x2f3e210 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-WhileStmt 0x2f3e220 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-DoStmt 0x2f3e230 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-SwitchStmt 0x2f3e240 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-CaseStmt 0x2f3e250 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-DefaultStmt 0x2f3e260 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-ReturnStmt 0x2f3e270 <col:23, line:8:1>
    |-CompoundStmt 0x2f7dfa0 <col:23, line:8:1>
      |-DeclStmt 0x2f3e1f8 <line:2:2, col:11>
        `-
  |-ImplicitCastExpr 0x2f3e2e8 <col:8> 'int' <LValueToRValue>
    |-DeclRefExpr 0x2f3e2c0 <col:8> 'int' lvalue ParmVar 0x2f3e000 'n' 'int'
  |-UnaryOperator 0x2f3e360 <col:8> 'int' postfix '--'
    |-DeclRefExpr 0x2f3e338 <col:8> 'int' lvalue ParmVar 0x2f3e000 'n' 'int'
  |-ReturnStmt 0x2f3e410 <col:8>
    |-ImplicitCastExpr 0x2f3e3f8 <col:9> 'int' <LValueToRValue>
      |-DeclRefExpr 0x2f3e3d0 <col:9> 'int' lvalue Var 0x2f3e180 'r' 'int'
`-ComputeResult 0x2f3e420 <col:8>
```

We could also analyze code on this level, but we won't be doing this in this lecture

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

Compile to LLVM IR

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

Optimize Bitcode

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```



Disassemble to human  
readable form

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

# Basic Tooling

- Optimizing

```
clang -c -emit-llvm factorial.c  
opt -O1 -o factorial-o1.bc factorial.bc  
llvm-dis factorial.bc  
llvm-dis factorial-o1.bc
```

- Compare `factorial.ll` and `factorial-o1.ll`
- Optimized version is shorter and contains phi instructions
- Continue for `-O2` and `-O3`

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

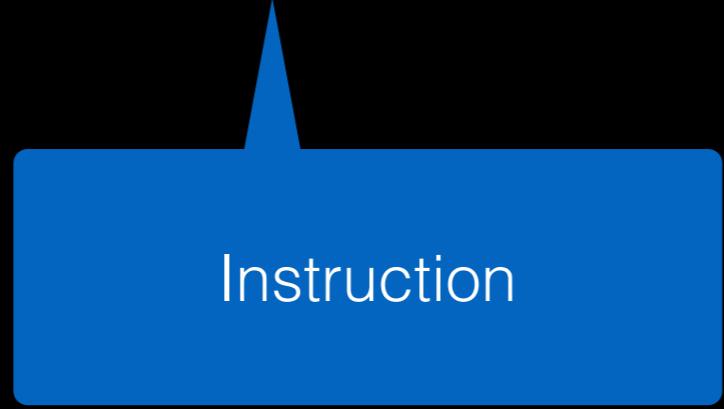
Target register

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```



Instruction

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```



No signed wrap

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```



Type

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

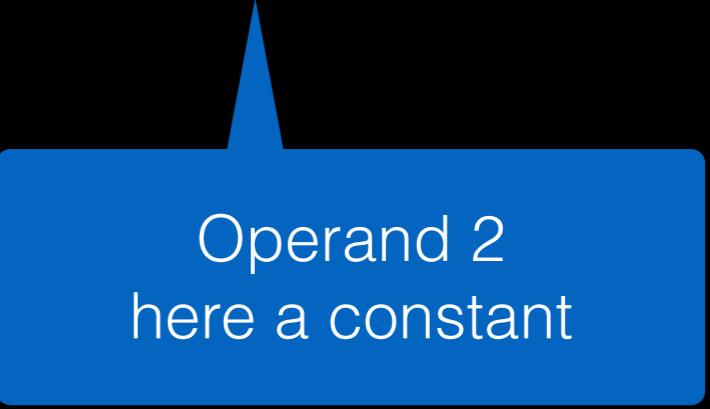
Operand 1  
here source register

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```



Operand 2  
here a constant

# Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

# LLVM IR

- RISC-like instruction set, but
  - Strongly typed — every instruction has typed arguments
  - Explicit control flow
  - Explicit data flow (Static Single Assignment form)

**Why should we include  
type information in  
assembly languages?**

# Let's compare

- IA-64 xor  
 $xor\ r1 = r2, r3$
- What guarantees do we have for this operation?

# Let's compare

- IA-64 xor  
 $xor\ r1 = r2, r3$
- What guarantees do we have for this operation?
- LLVM IR xor  
 $\%r = xor\ i32\ \%v1, \%v2$
- What guarantees do we have for this operation?

# Let's compare

- IA-64 xor

`xor r1 = r2, r3`

- What guarantees do we have for this operation?

- LLVM IR xor

`%r = xor i32 %v1, %v2`

- What guarantees do we have for this operation?

Operands are 32-bit  
integers

# Let's compare

- IA-64 xor  
 $xor\ r1 = r2, r3$
- What guarantees do we have for this operation?
- LLVM IR xor  
 $\%r = xor\ i32\ \%v1, \%v2$
- What guarantees do we have for this operation?

# Let's compare

- IA-64 xor  
 $\text{xor } r1 = r2, r3$
- What guarantees do we have for this operation?

Result in a 32-bit integer

$\ll \text{v1} \text{ v2} \text{ i32} \text{ xor}$

$\%r = \text{xor } \text{i32 } \%v1, \%v2$

- What guarantees do we have for this operation?

# Let's compare

- IA-64 xor  
 $xor\ r1 = r2, r3$
- What guarantees do we have for this operation?
- LLVM IR xor  
 $\%r = xor\ i32\ \%v1, \%v2$
- What guarantees do we have for this operation?

# Typed Assembly Languages

- Do you know more typed assembly languages?

# Typed Assembly Languages

- Do you know more typed assembly languages?

**e.g., Java Bytecode**

# Typed Assembly Languages

- Do you know more typed assembly languages?

e.g., **Java Bytecode**

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. **From system f to typed assembly language**. ACM Trans. Program. Lang. Syst., 21(3):527– 568, May 1999

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. **TALx86: A realistic typed assembly language**. In In Second Workshop on Compiler Support for System Software, pages 25–35, 1999

# Writing Analysis Passes

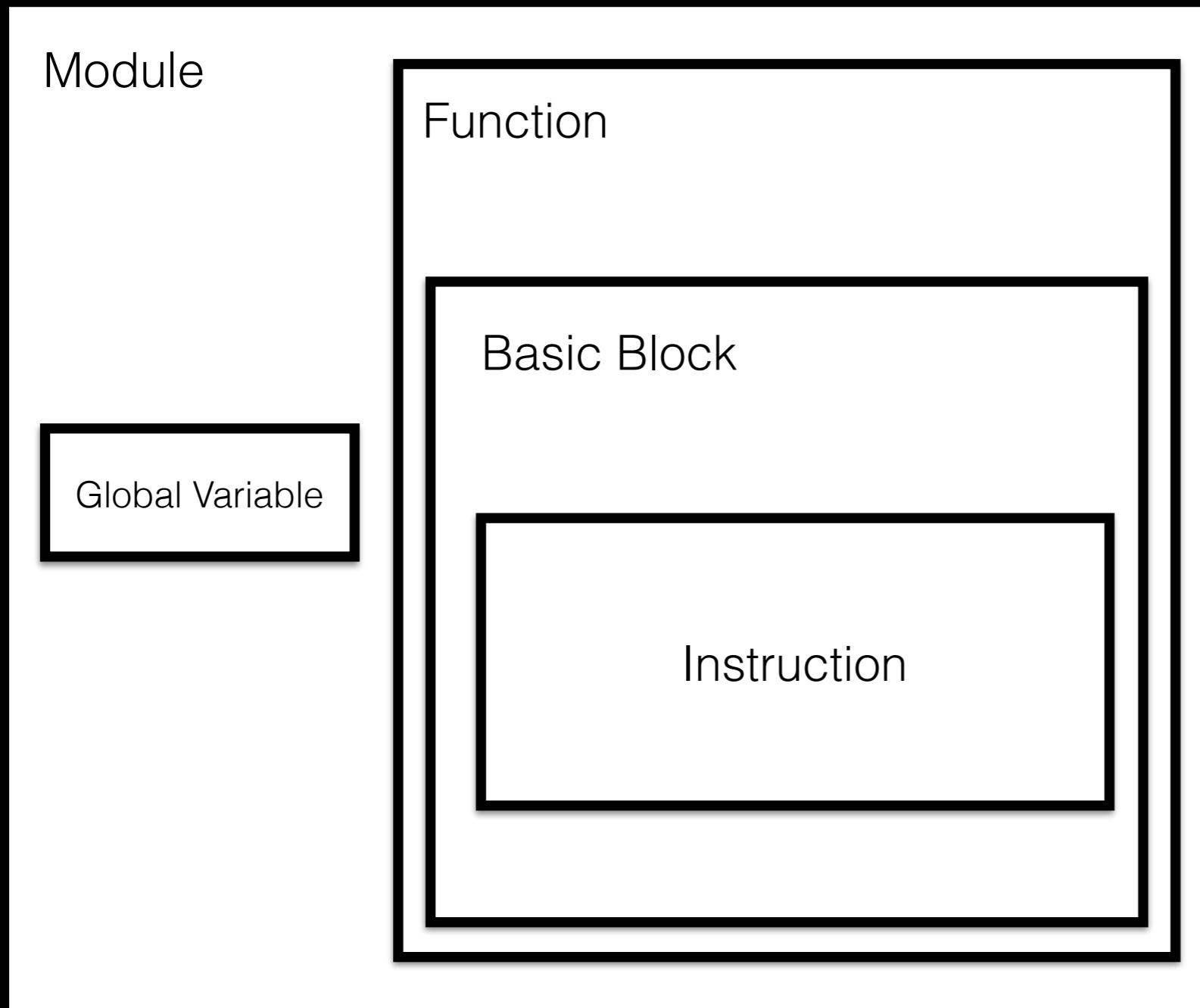
# LLVM is its own ecosystem

- LLVM comes with many classes that are useful when writing analyses and transforms
- Error handling
- Data structures
- Analysis/Transform API
- Always a good read:  
<http://llvm.org/docs/ProgrammersManual.html>

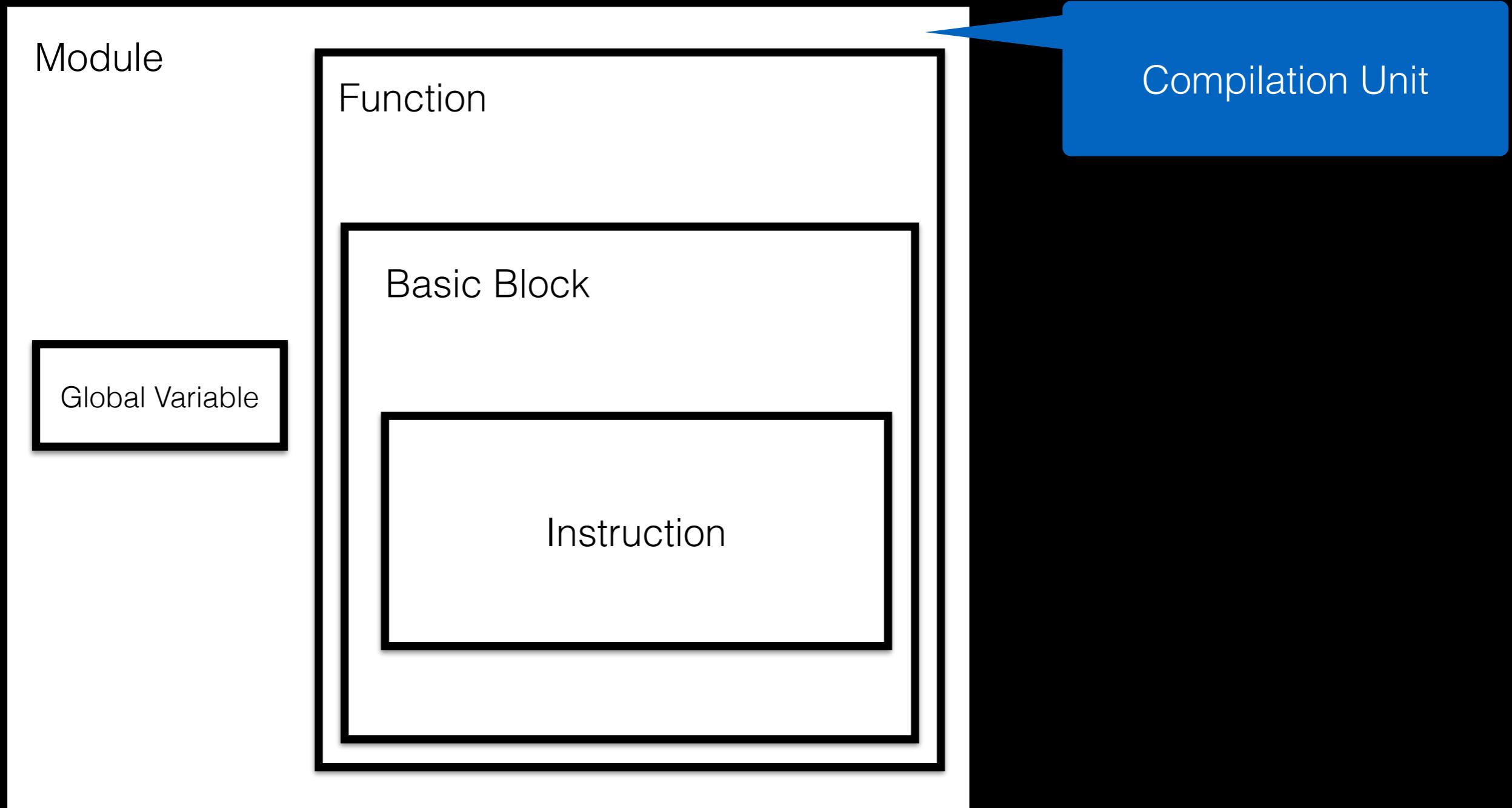
# Passes in LLVM

- During runtime LLVM applies a sequence of analyses and transformations on the target program to achieve a desired result
- These analyses and transformations are called passes
- Machine independent passes are invoked using `opt`
- Machine dependent passes are invoked using `llc`

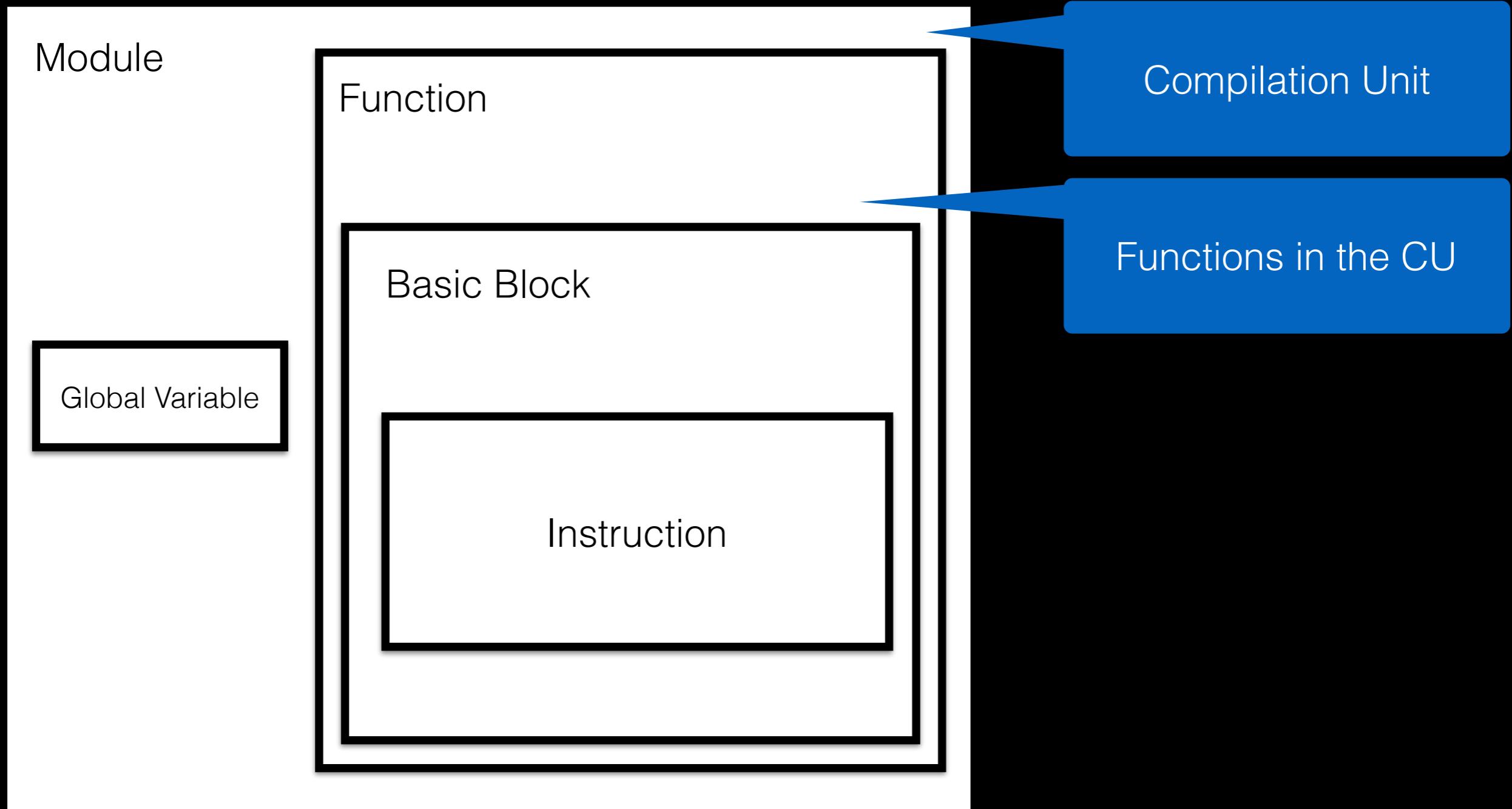
# Scopes in the Language



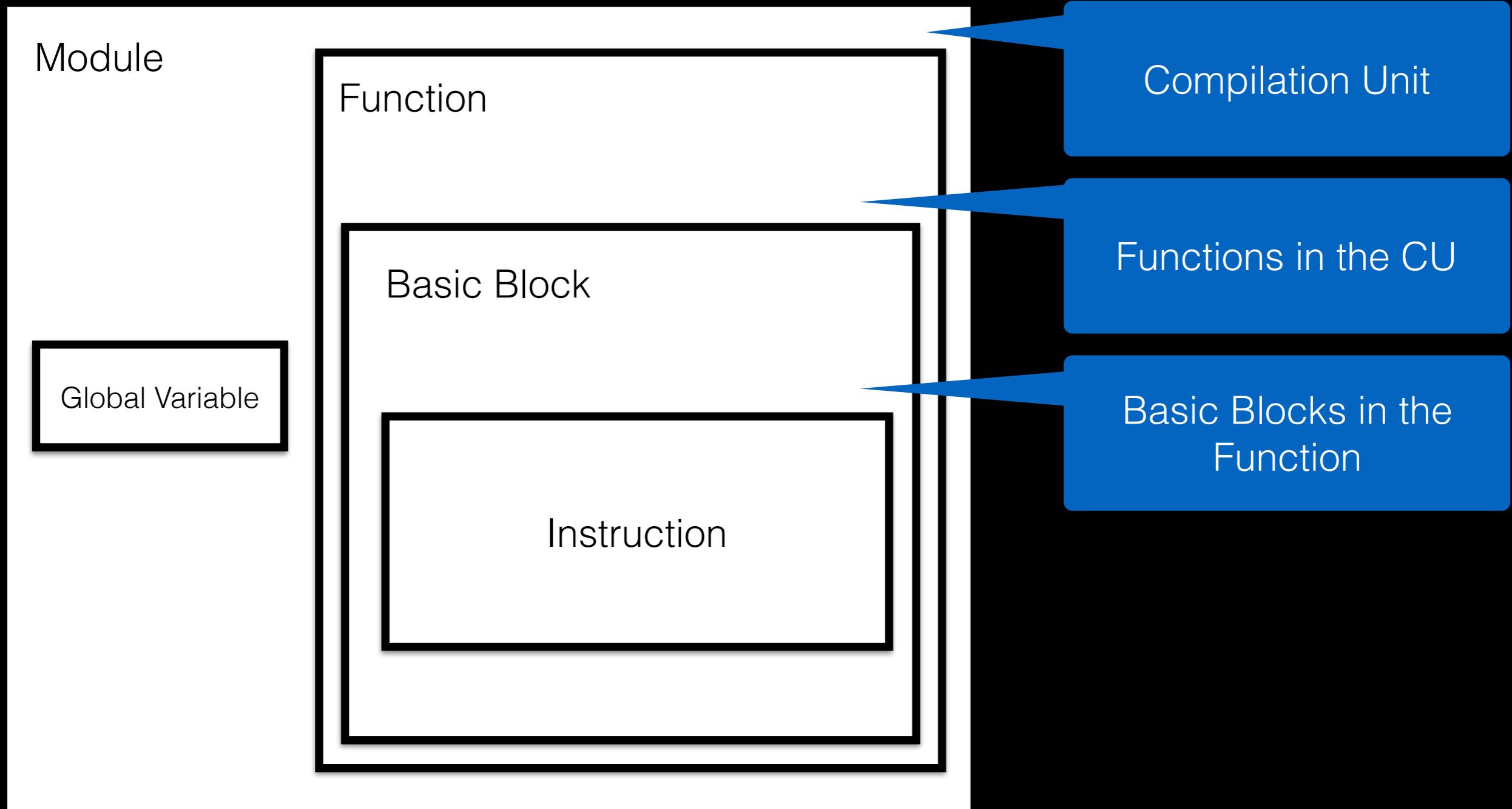
# Scopes in the Language



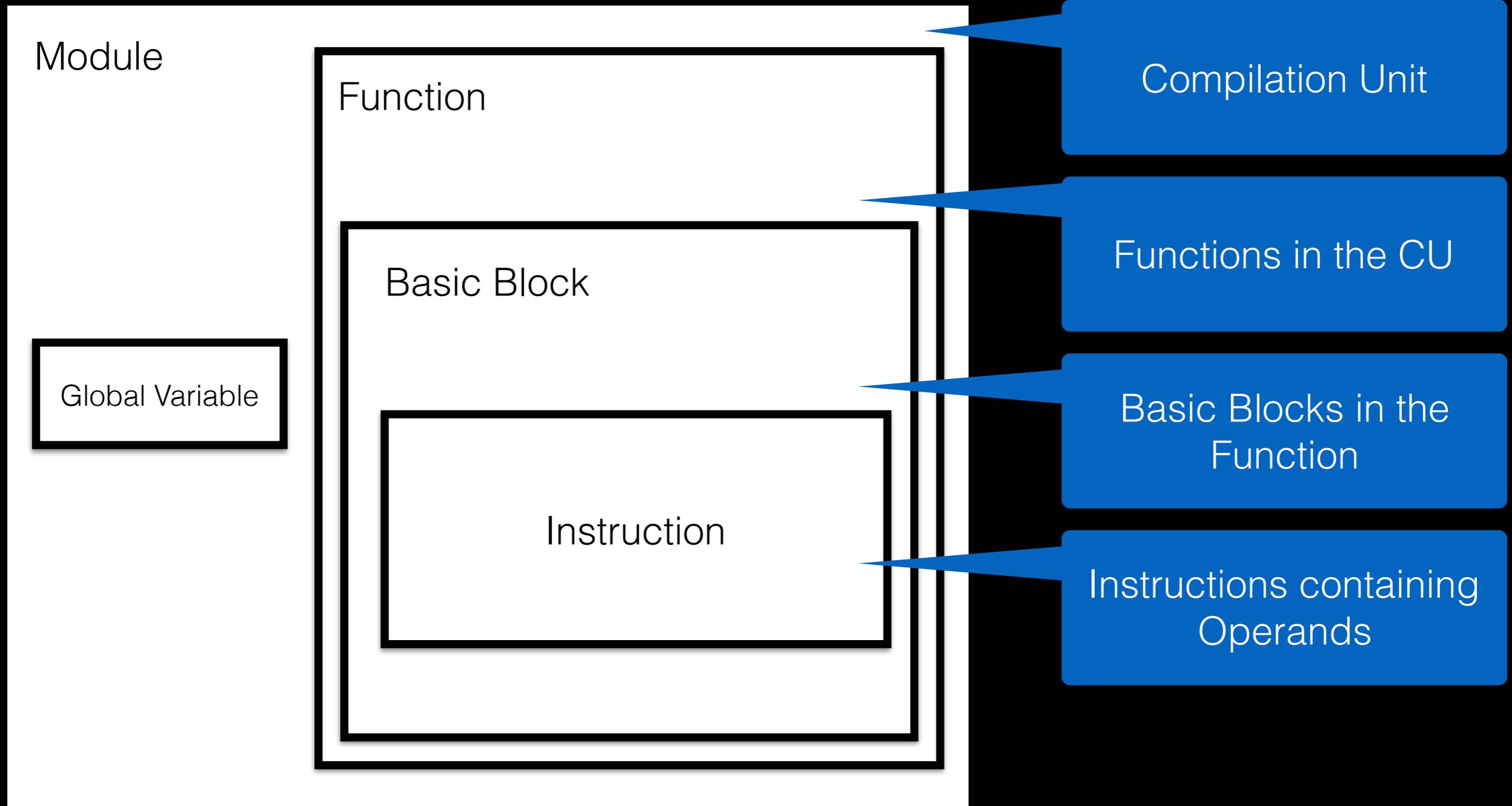
# Scopes in the Language



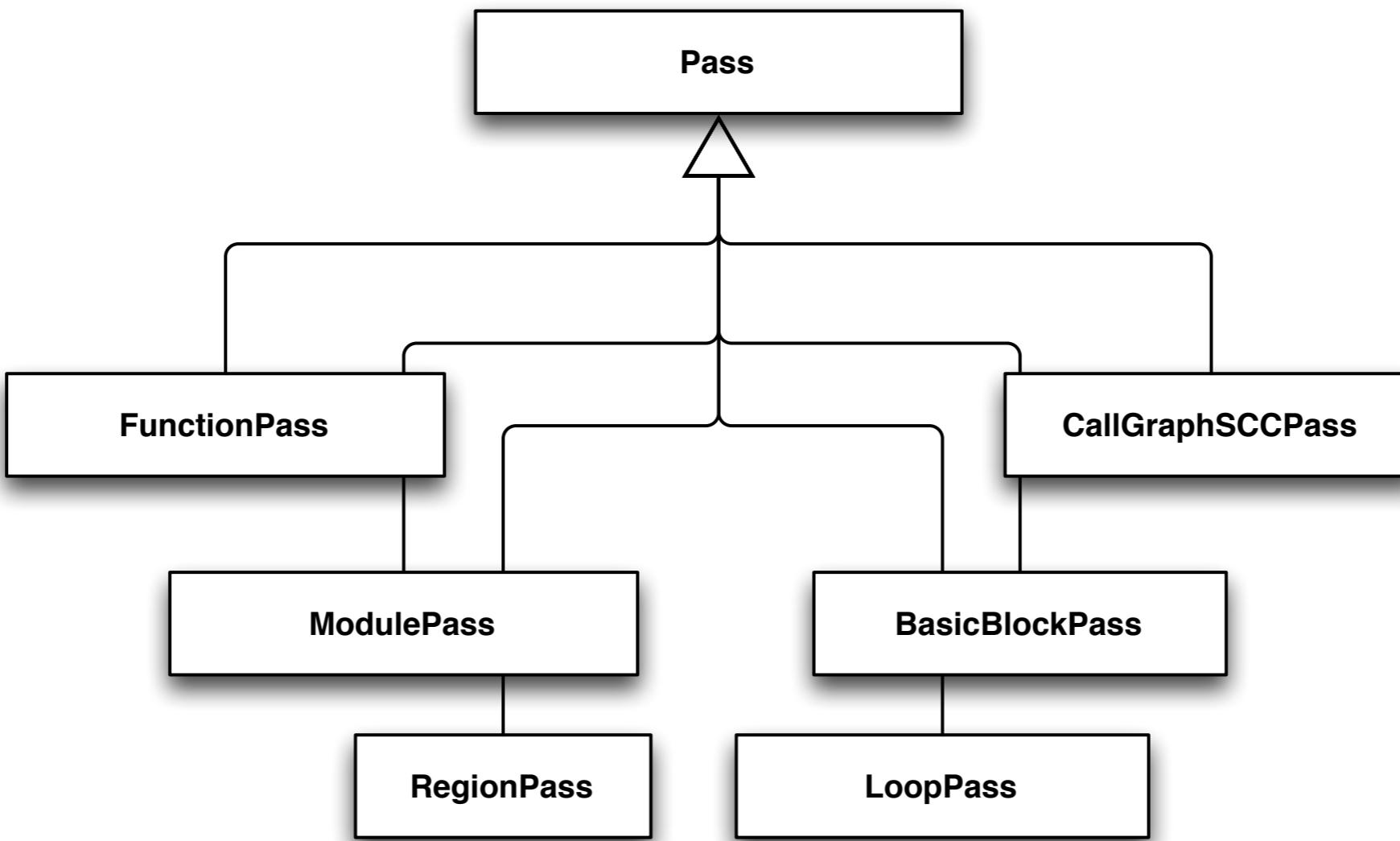
# Scopes in the Language



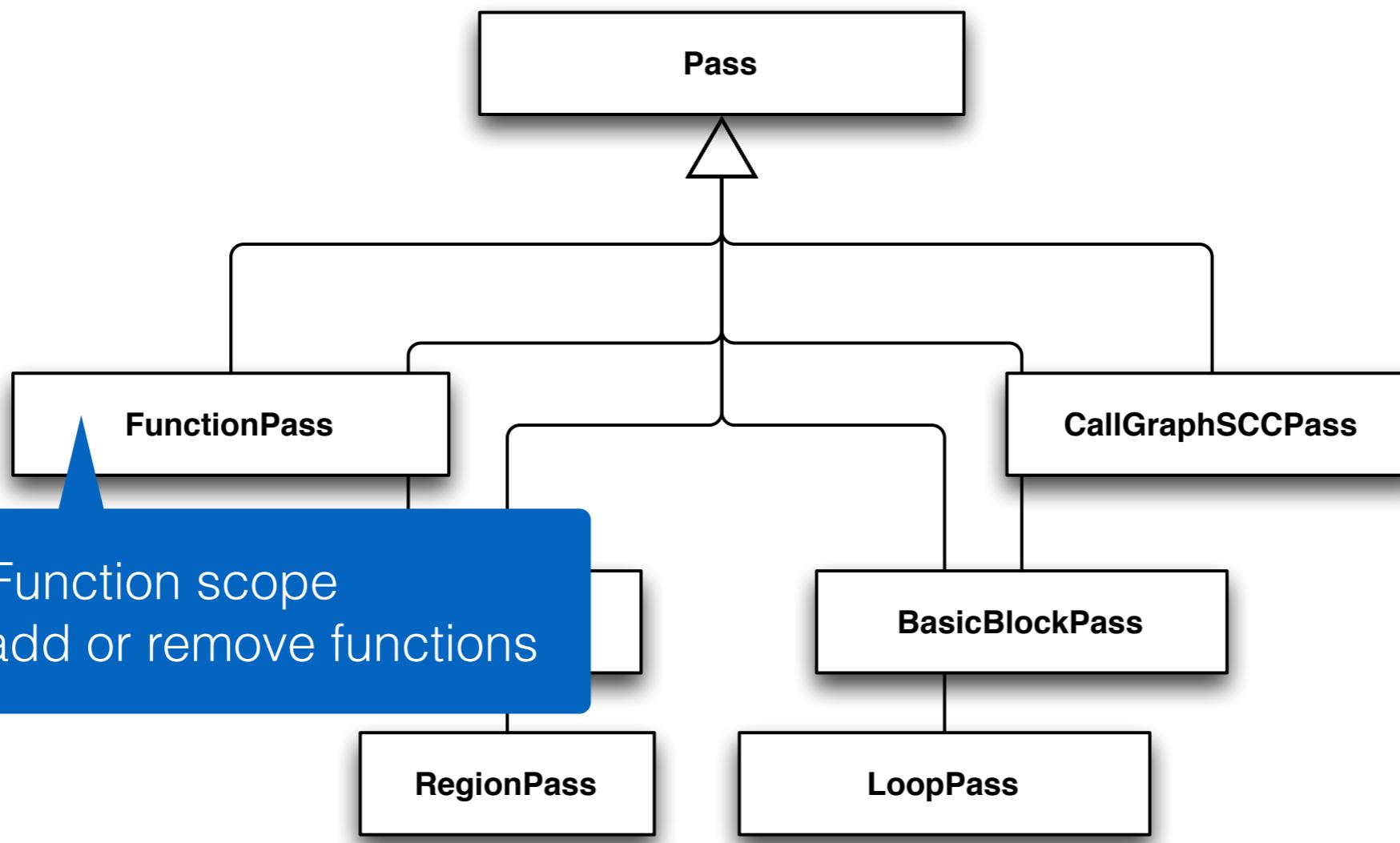
# Scopes in the Language



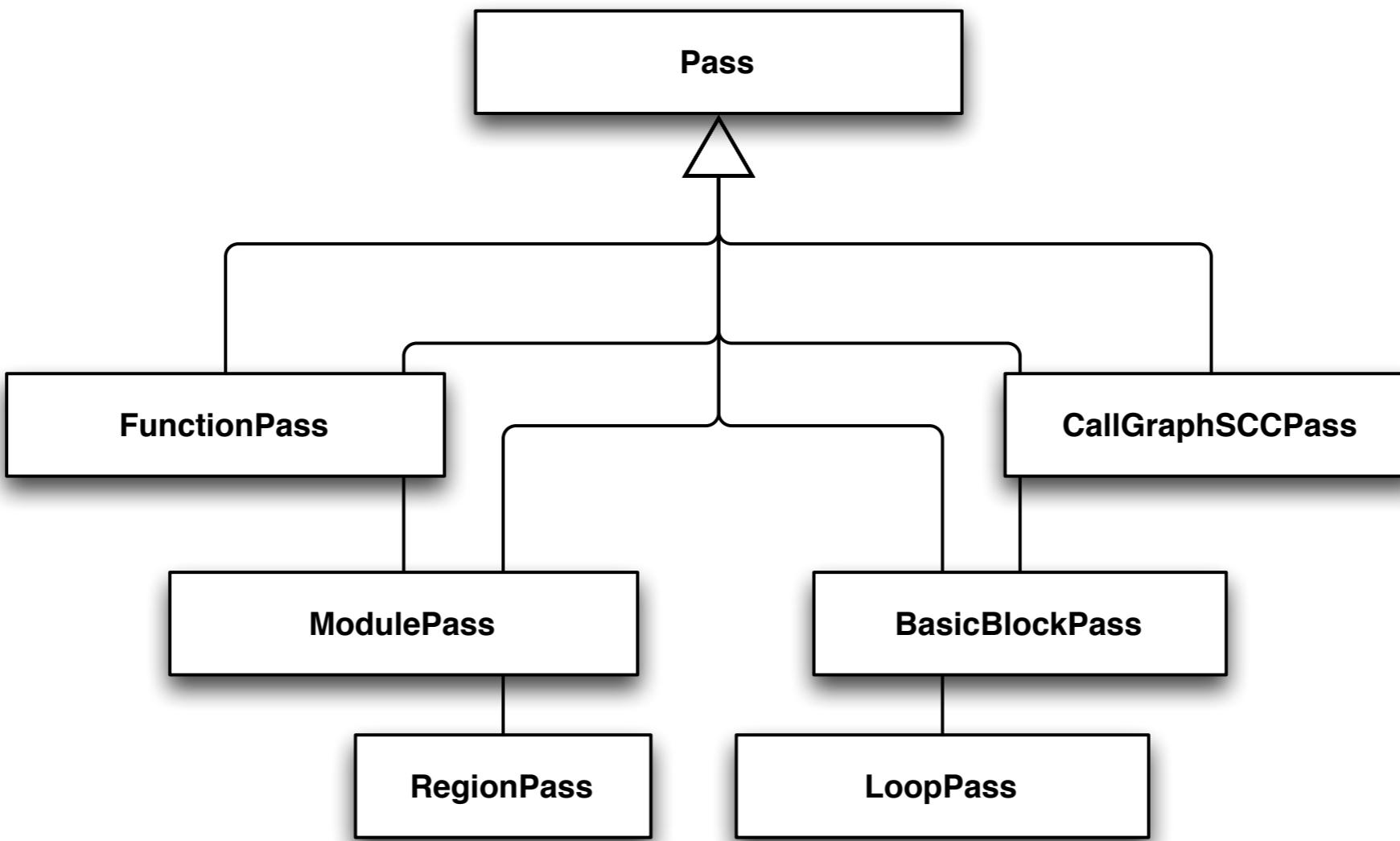
# Scopes for Analyses



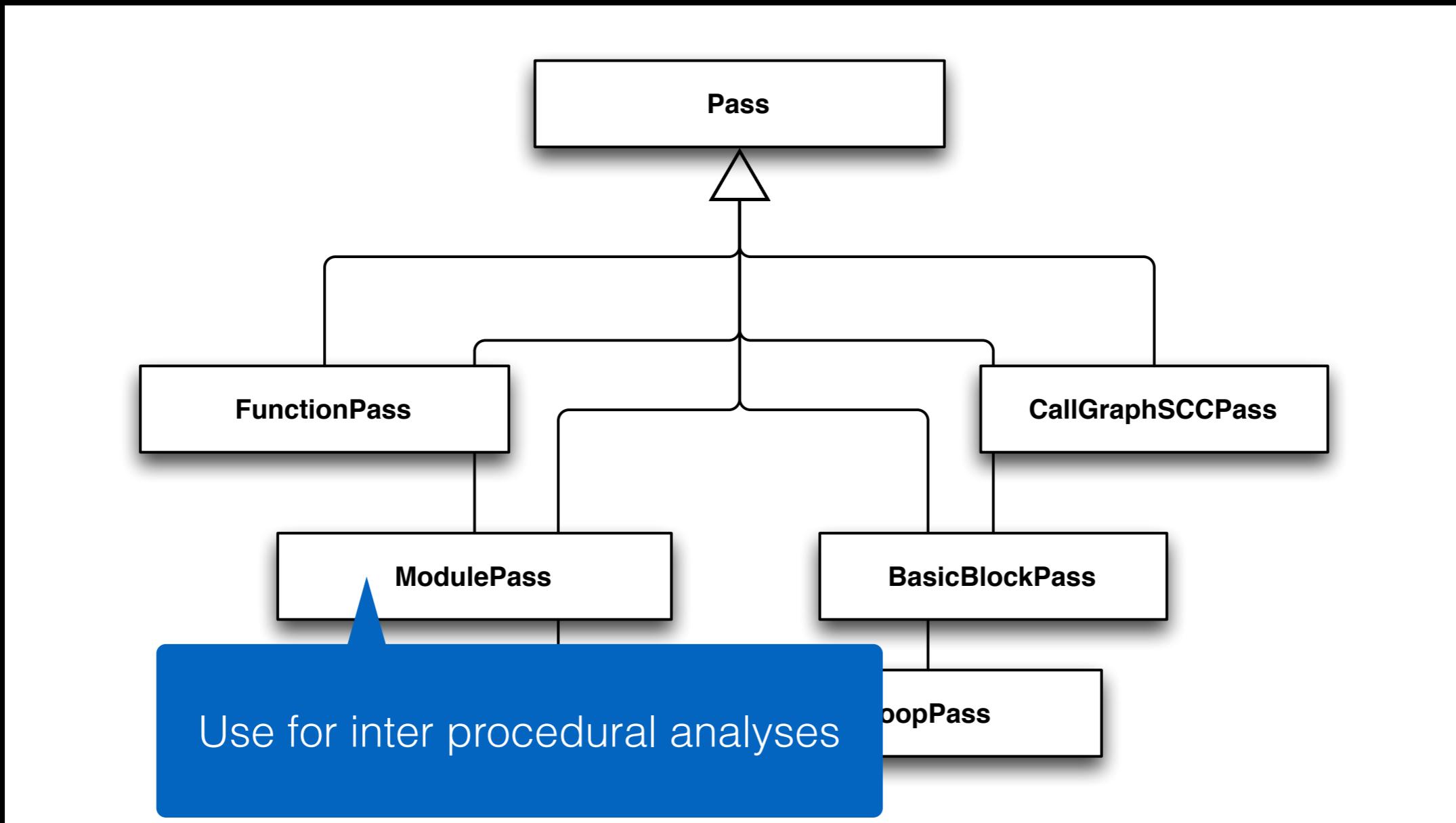
# Scopes for Analyses



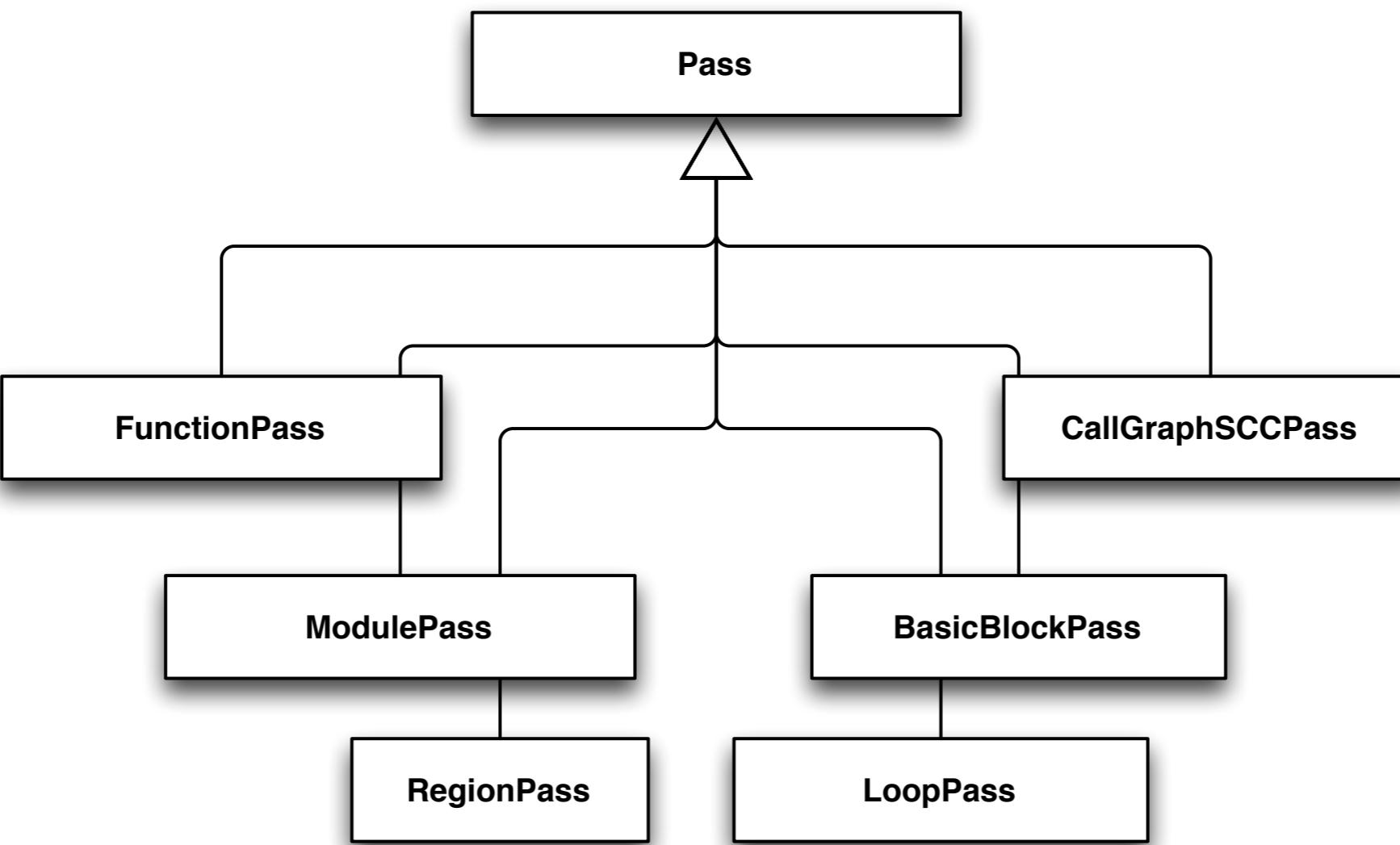
# Scopes for Analyses



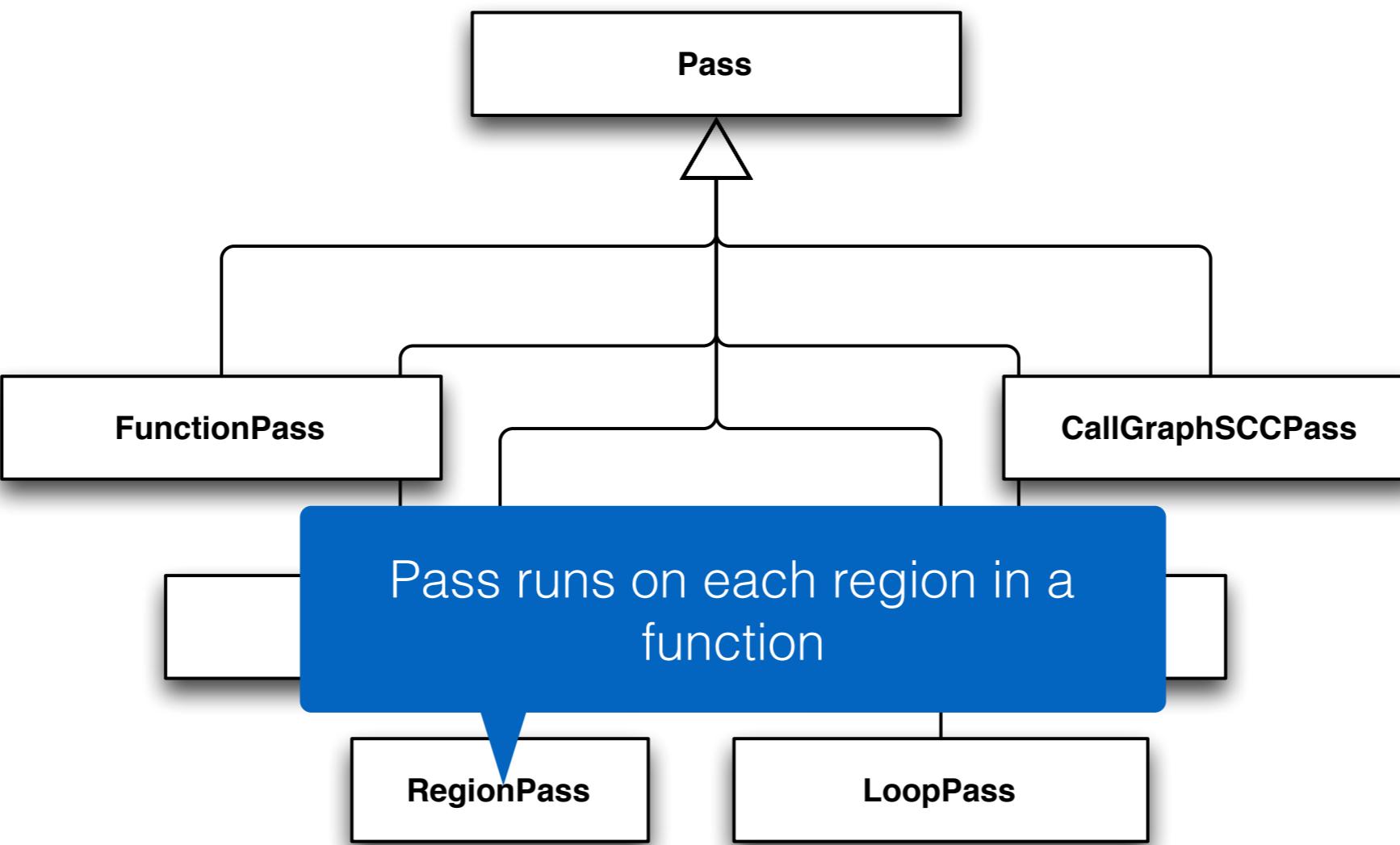
# Scopes for Analyses



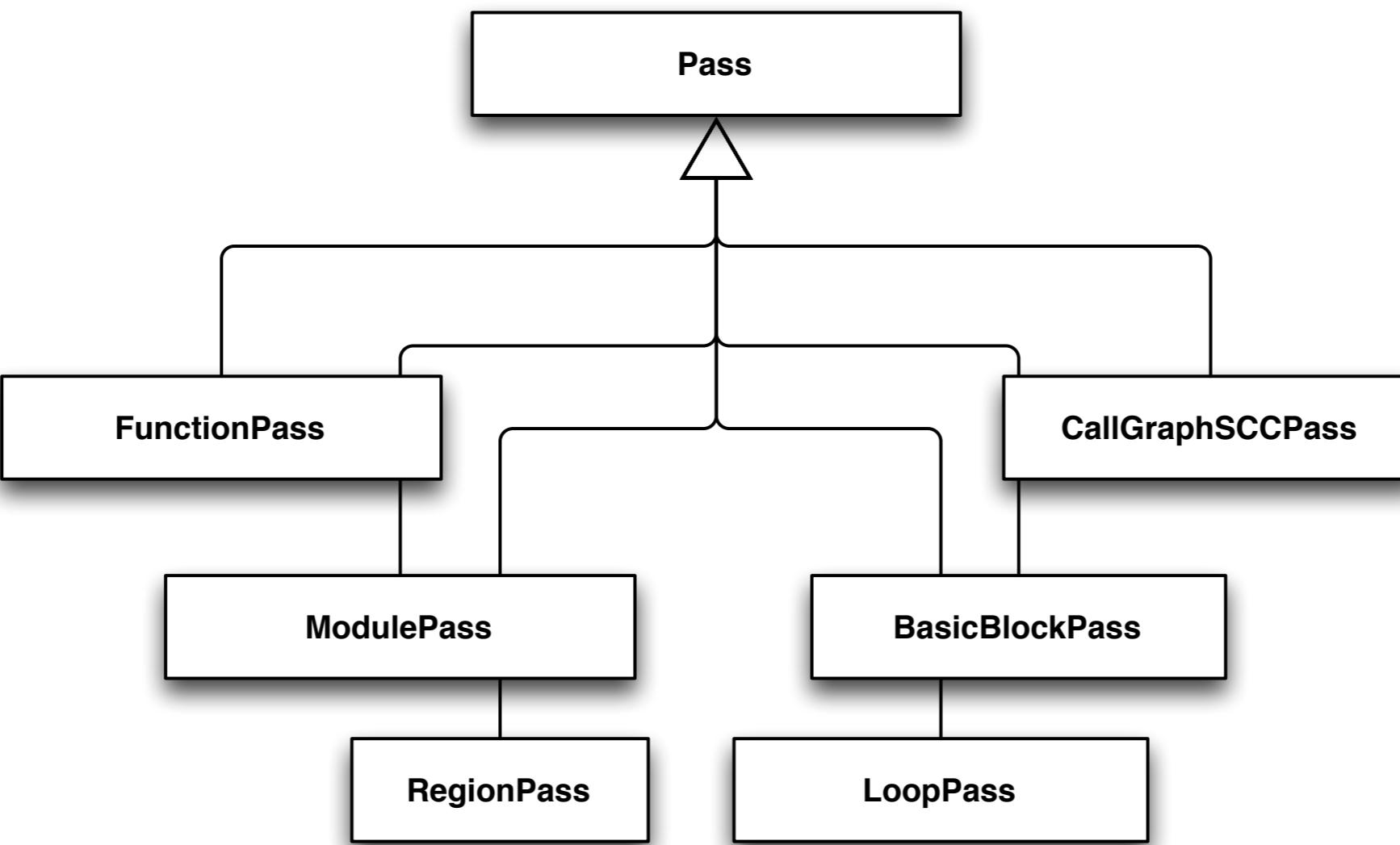
# Scopes for Analyses



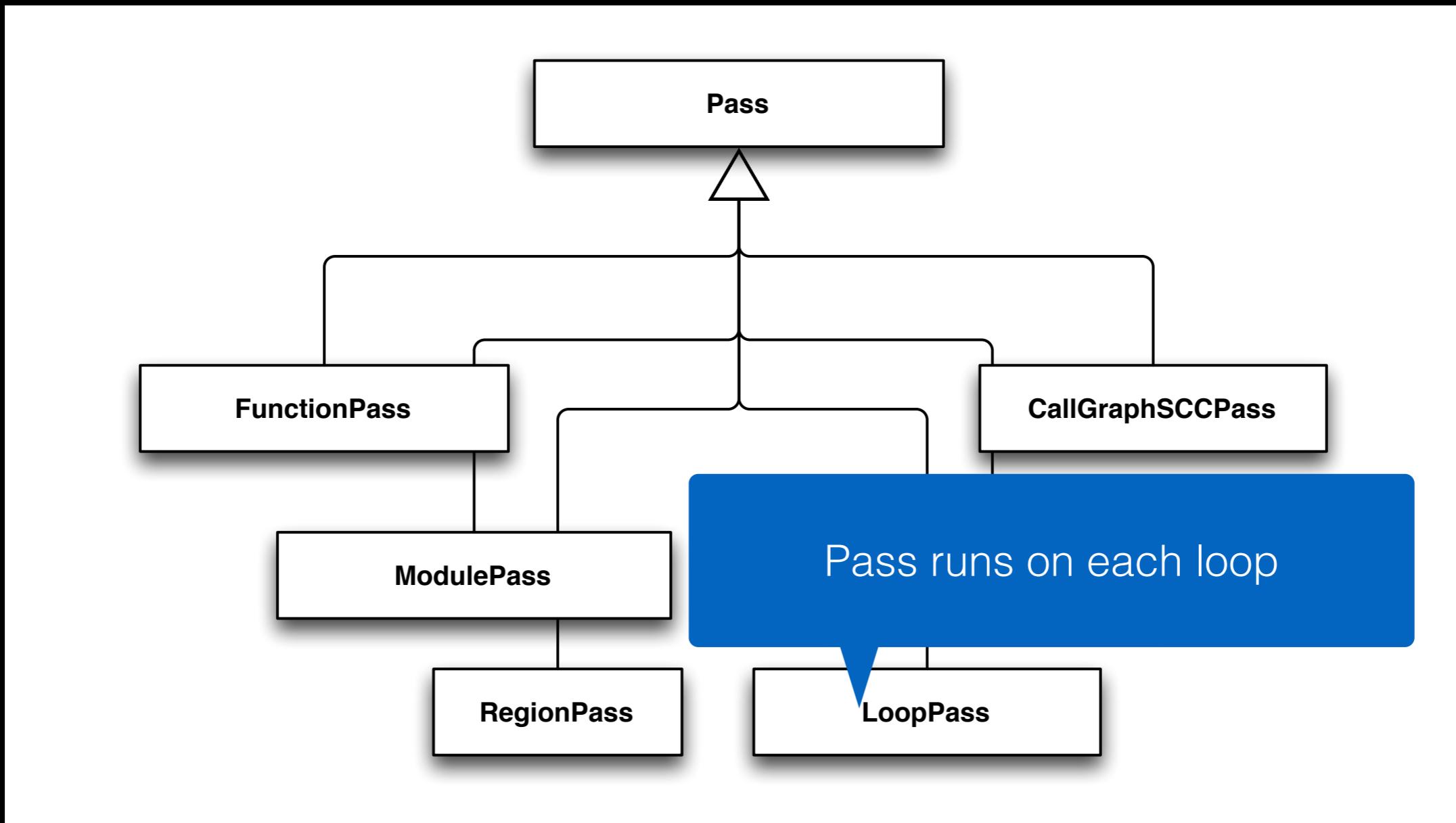
# Scopes for Analyses



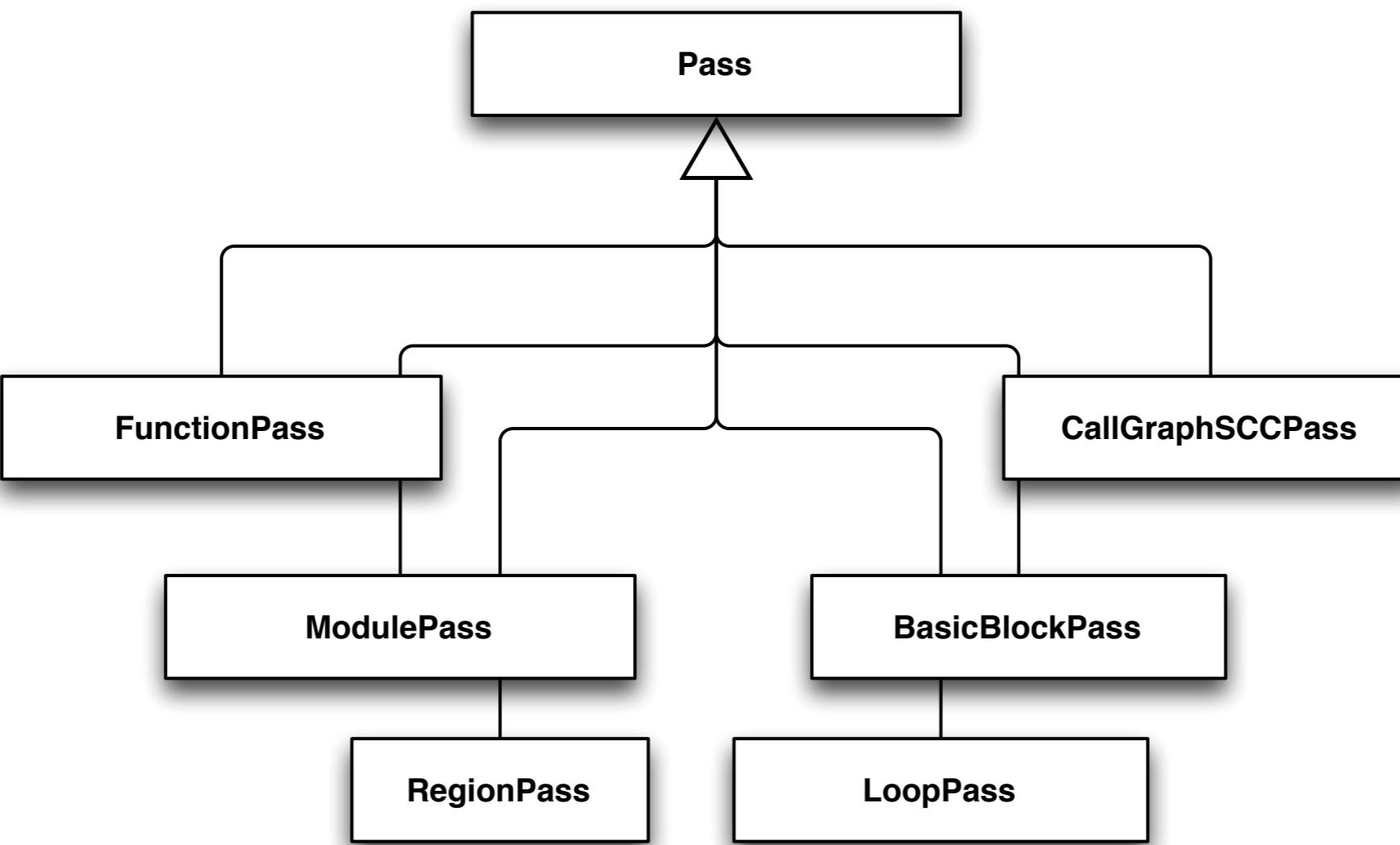
# Scopes for Analyses



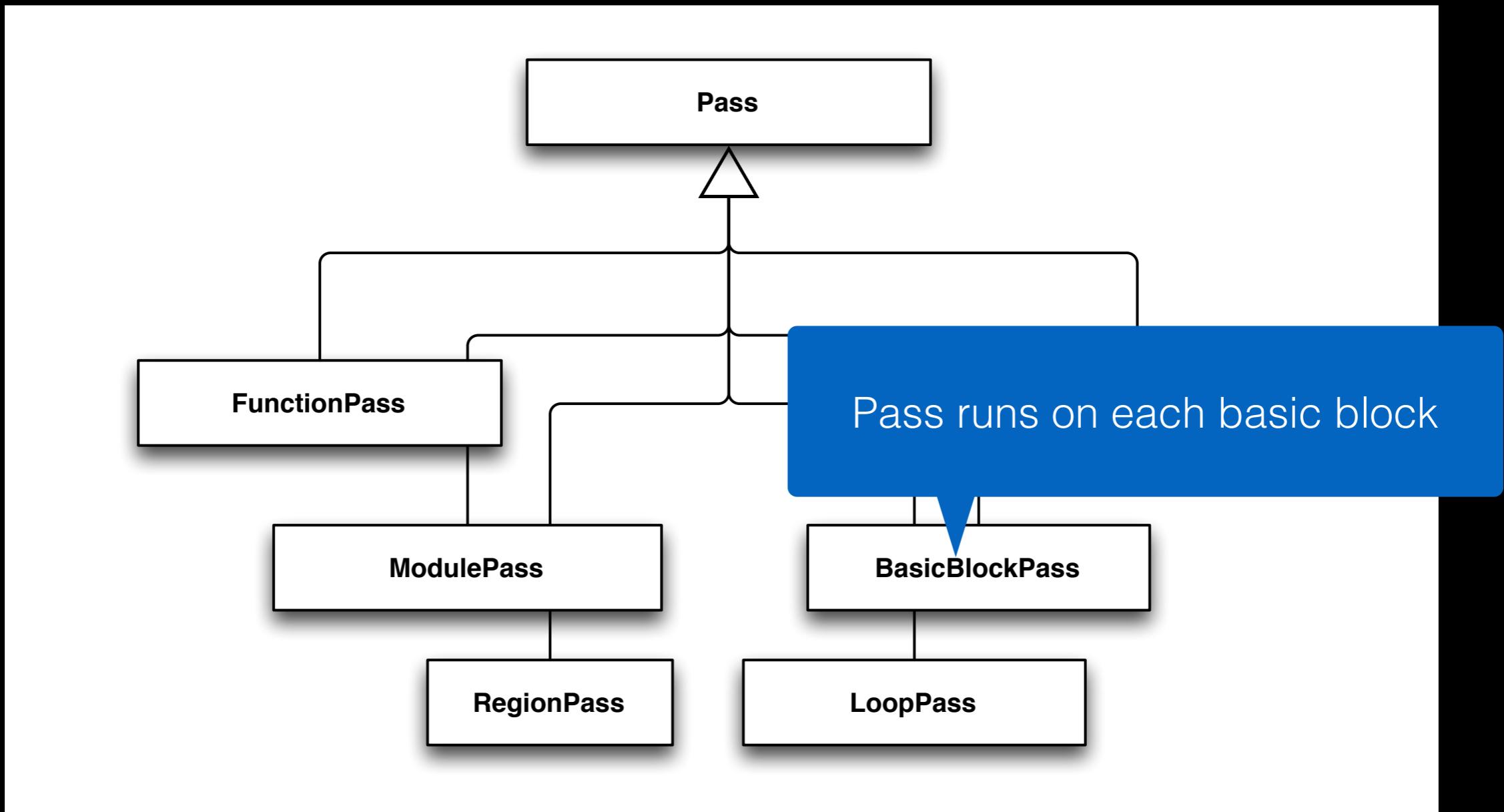
# Scopes for Analyses



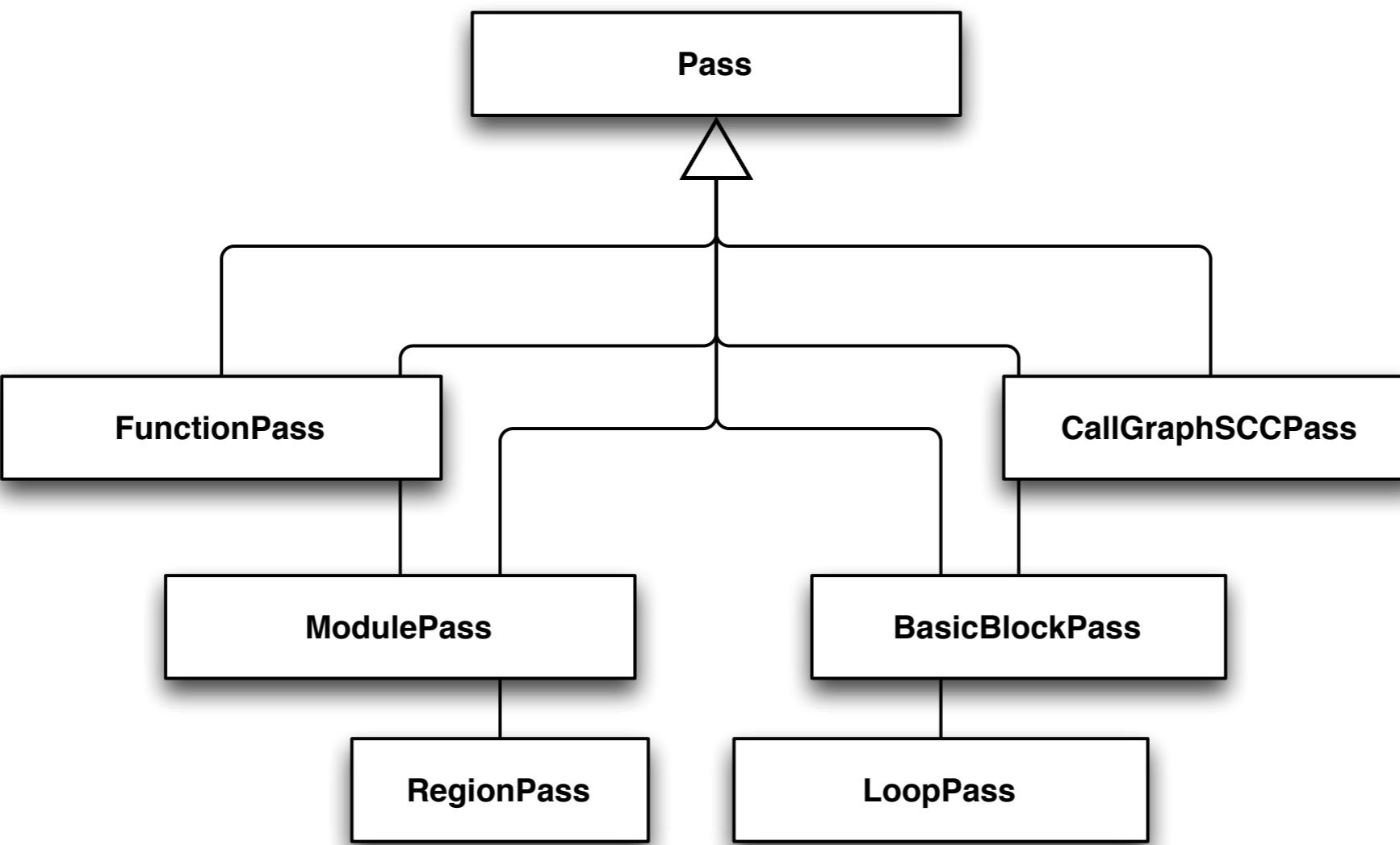
# Scopes for Analyses



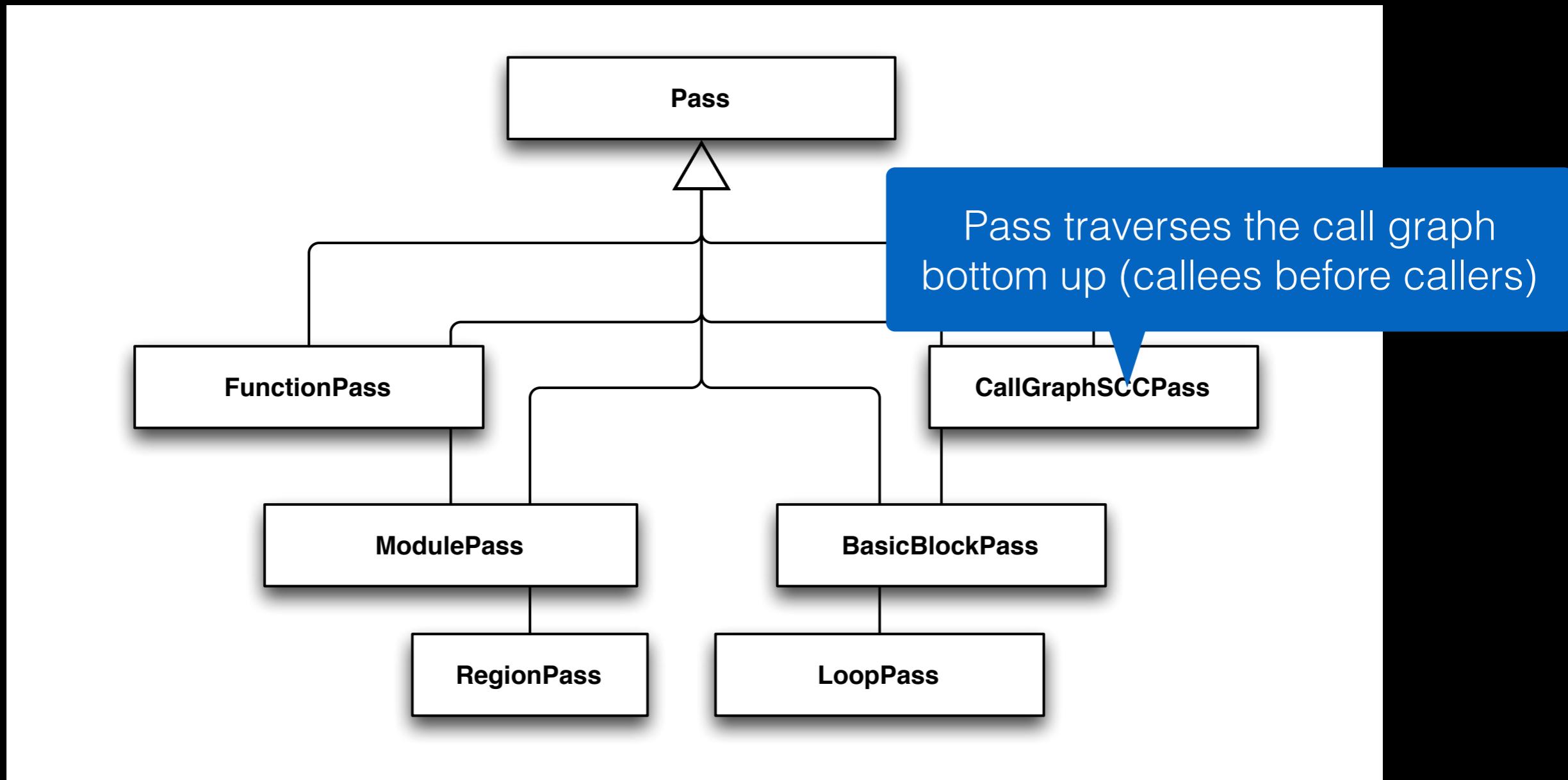
# Scopes for Analyses



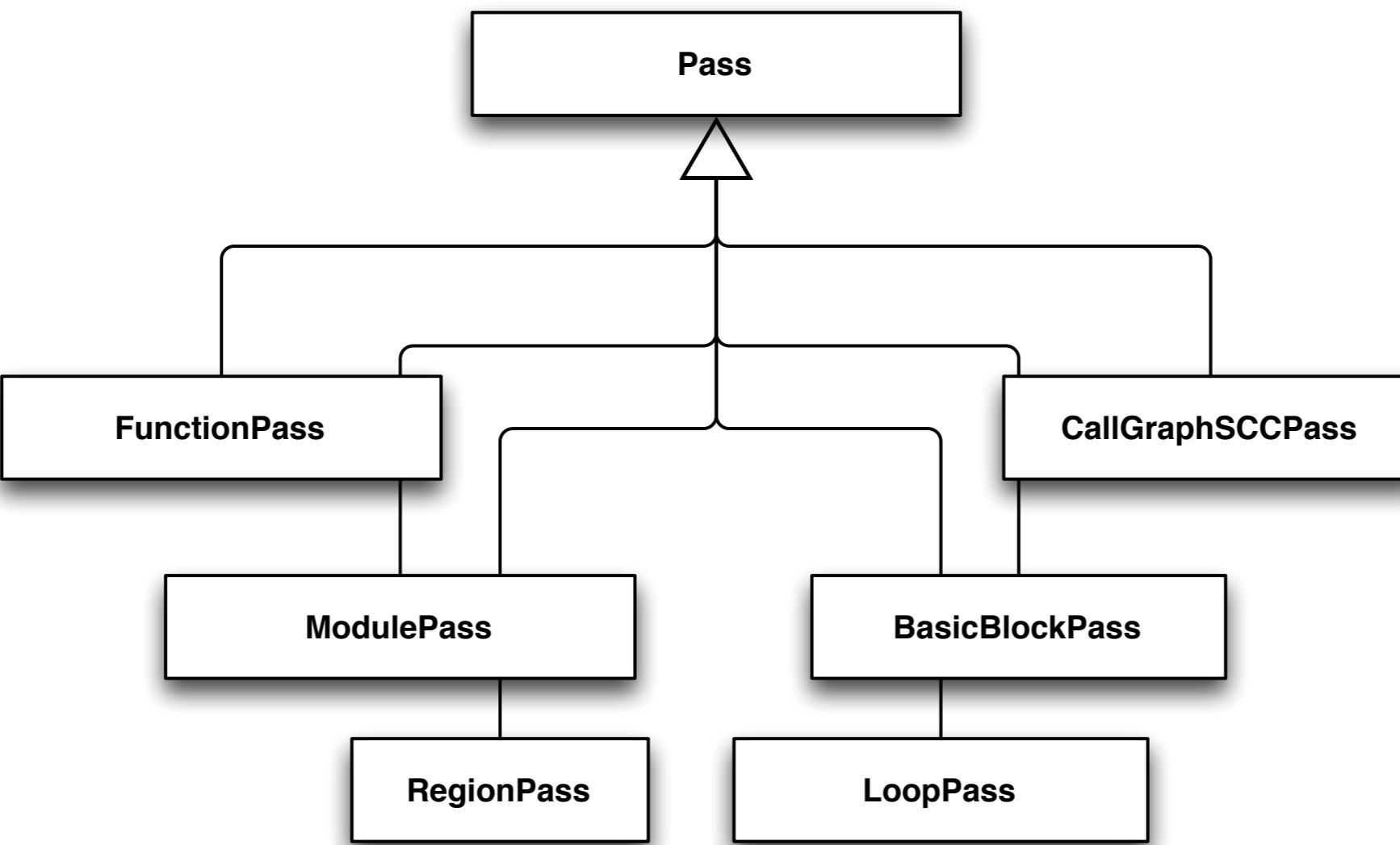
# Scopes for Analyses



# Scopes for Analyses



# Scopes for Analyses



# Setting up your environment

- Compiling the exercises

```
cd exercises  
mkdir build  
cd build  
cmake -g "Unix Makefiles" -DLLVM_DIR=${LLVMSRC_DIR} ..  
make
```

- Running the first custom analysis pass

```
cd ~/examples  
clang -c -emit-llvm test.c  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so -help  
  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so  
-FirstPass < test.bc > /dev/null
```

# Setting up your environment

- Compiling the exercises

```
cd exercises  
mkdir build  
cd build  
cmake -g "Unix Makefiles" -DLLVM_DIR=${LLVMSRC_DIR} ..  
make
```

Change this if not using  
our docker container

- Running the first custom analysis pass

```
cd ~/examples  
clang -c -emit-llvm test.c  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so -help  
  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so  
-FirstPass < test.bc > /dev/null
```

# Setting up your environment

- Compiling the exercises

```
cd exercises  
mkdir build  
cd build  
cmake -g "Unix Makefiles" -DLLVM_DIR=${LLVMSRC_DIR} ..  
make
```

Change this if not using  
our docker container

- Running the exercises

That's an alias in the docker container

— use the opt you build against —

```
cd examples  
clang -c -emit-llvm test.c  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so -help
```

```
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so  
-FirstPass < test.bc > /dev/null
```

# Setting up your environment

- Compiling the exercises

```
cd exercises  
mkdir build  
cd build  
cmake -g "Unix Makefiles" -DLLVM_DIR=${LLVMSRC_DIR} ..  
make
```

Change this if not using  
our docker container

- Running the first custom analysis pass

```
cd ~/examples  
clang -c -emit-llvm test.c  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so -help  
  
opt-3.8 -load ../../exercises/build/FirstPass/libFirstPass.so  
-FirstPass < test.bc > /dev/null
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

Include what you need

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

Use the LLVM namespace

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
```

We implement a FunctionPass

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```



The implementation

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

The registration block

# What makes an analysis pass?

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FirstPass : public FunctionPass {
        static char ID;
        FirstPass() : FunctionPass(ID) {}

        bool runOnFunction(Function &F) override {
            errs() << "Function recognized: ";
            errs().write_escaped(F.getName());
            errs() << '\n';
            return false;
        }
    };
}

char FirstPass::ID = 0;
static RegisterPass<FirstPass> X("FirstPass", "Function printer pass", false,
false);
```

# How does that compile?

```
~/exercises/  
    CMakeLists.txt  
FirstPass/  
    CMakeLists.txt  
    FirstPass.cpp
```

# How does that compile?

```
~/exercises/  
    CMakeLists.txt  
FirstPass/  
    CMakeLists.txt  
        FirstPass.cpp
```

Binding to LLVM and list of  
subdirectories to consider

# How does that compile?

~/exercises/

    CMakeLists.txt

    FirstPass/

        CMakeLists.txt

        FirstPass.cpp

Binding to LLVM and list of  
subdirectories to consider

```
cmake_minimum_required(VERSION 2.8)
find_package(LLVM REQUIRED CONFIG)
```

```
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")
```

```
include_directories(${LLVM_INCLUDE_DIRS})
message(STATUS "Using includes in: ${LLVM_INCLUDE_DIRS}")
add_definitions(${LLVM_DEFINITIONS})
message(STATUS "Using definitions in in: ${LLVM_DEFINITIONS}")
add_definitions("-std=c++11")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti -std=c++11")

add_subdirectory(FirstPass)
```

# How does that compile?

~/exercises/

CMakeLists.txt

FirstPass/

CMakeLists.txt

FirstPass.cpp

Binding to LLVM and list of  
subdirectories to consider

```
cmake_minimum_required(VERSION 2.8)
find_package(LLVM REQUIRED CONFIG)
```

```
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")
```

```
include_directories(${LLVM_INCLUDE_DIRS})
message(STATUS "Using includes in: ${LLVM_INCLUDE_DIRS}")
add_definitions(${LLVM_DEFINITIONS})
message(STATUS "Using definitions in in: ${LLVM_DEFINITIONS}")
add_definitions("-std=c++11")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti -std=c++11")
```

```
add_subdirectory(FirstPass)
```

Add your analyses here

# How does that compile?

```
~/exercises/  
    CMakeLists.txt  
FirstPass/  
    CMakeLists.txt  
    FirstPass.cpp
```

# How does that compile?

~/exercises/

  CMakeLists.txt

  FirstPass/

    CMakeLists.txt

    FirstPass.cpp

Compilation information for the  
specific analysis

# How does that compile?

~/exercises/

    CMakeLists.txt

    FirstPass/

        CMakeLists.txt

        FirstPass.cpp

Compilation information for the  
specific analysis

```
cmake_minimum_required(VERSION 2.8)
add_library(FirstPass MODULE FirstPass.cpp)
```

# How does that compile?

```
~/exercises/  
    CMakeLists.txt  
FirstPass/  
    CMakeLists.txt  
    FirstPass.cpp
```

# How does that compile?

~/exercises/

    CMakeLists.txt

    FirstPass/

        CMakeLists.txt

        FirstPass.cpp

Analysis implementation

# How does that compile?

```
~/exercises/  
    CMakeLists.txt  
FirstPass/  
    CMakeLists.txt  
    FirstPass.cpp
```

# Exercise: Count Instructions

- Goal: Count instructions as a histogram per instruction type
- Scope: Complete code file
- Example: 14 `store` instructions, 14 `load` instructions, 5 `jmp` instructions, 4 `br` instructions

# Instruction Histogram

exercises/InstructionCount

# Instruction Histogram

`exercises/InstructionCount`

- We develop a `ModulePass`

```
struct InstructionCount : public ModulePass
```

# Instruction Histogram

## exercises/InstructionCount

- We develop a `ModulePass`  
`struct InstructionCount : public ModulePass`
- For the histogram we use a LLVM data structure  
`DenseMap<unsigned int, uint64_t> counts;`

# Instruction Histogram

## exercises/InstructionCount

- We develop a `ModulePass`

```
struct InstructionCount : public ModulePass
```

- For the histogram we use a LLVM data structure

```
DenseMap<unsigned int, uint64_t> counts;
```

- We iterate over functions, basic blocks, and instructions

```
for(Function &fun : M) {  
    for(BasicBlock &bb : fun) {  
        for(Instruction &i : bb) {
```

# Instruction Histogram

exercises/InstructionCount

# Instruction Histogram

exercises/InstructionCount

- Let's look at the opcode of the instruction

```
unsigned int currentOpcode = i.getOpcode();
```

# Instruction Histogram

## exercises/InstructionCount

- Let's look at the opcode of the instruction

```
unsigned int currentOpcode = i.getOpcode();
```

- We increment for each opcode

```
auto instCount = counts.find(currentOpcode);
if (counts.end() == instCount) {
    instCount = counts.insert(std::make_pair(currentOpcode,
0)).first;
}
++instCount->second;
```

# Instruction Histogram

## exercises/InstructionCount

- Let's look at the opcode of the instruction

```
unsigned int currentOpcode = i.getOpcode();
```

- We increment for each opcode

```
auto instCount = counts.find(currentOpcode);
if (counts.end() == instCount) {
    instCount = counts.insert(std::make_pair(currentOpcode,
0)).first;
}
++instCount->second;
```

- After the loops, we print out the histogram information

Let's use it.

# Data-flow Analysis

- Goal: Reconstruct how data flows through a program
- Application
  - Find programming mistakes
  - Find security flaws (confidentiality, integrity)
  - Optimization

# Concepts

- For each basic block compute:

$$out_b = trans_b(in_b)$$

$$in_b = join_{p \in pred_b}(out_p)$$

- Beware of

- entry points
- cycles (i.e., loops)

# Concepts

Transfer function

- For each basic block compute:

$$out_b = \overbrace{trans_b}^{\text{Transfer function}}(in_b)$$

$$in_b = join_{p \in pred_b}(out_p)$$

- Beware of

- entry points
- cycles (i.e., loops)

# Concepts

Transfer function

- For each basic block compute:

$$out_b = trans_b(in_b)$$

$$in_b = \underset{pred_b}{join}_{p \in pred_b}(out_p)$$

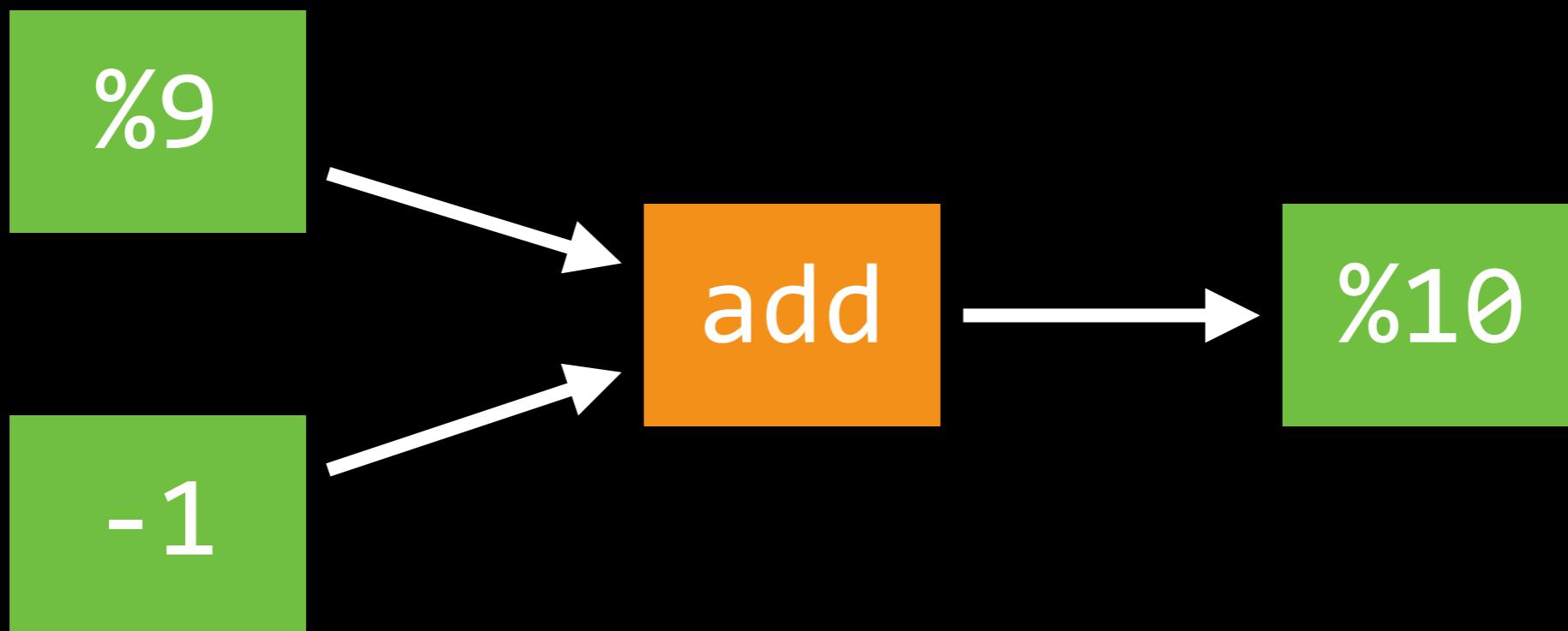
Join over all preceding  
blocks

- Beware of

- entry points
- cycles (i.e., loops)

# Transfer functions

```
%10 = add nsw i32 %9, -1
```



# Sensitivity

- A data-flow analysis is
  - **flow-sensitive**, if it honors the order of the statements
  - **path-sensitive**, if it honors the conditions applied on a path
  - **context-sensitive**, if an interprocedural analysis honors calling context when processing calls

# Exercise: Parameter Flows

- Write an analysis that tracks the flow of parameter values of functions
- Report the flow of the values by printing out the instructions it flows over
- Also print out a string of identifiers that it influences
- Think easy and naive - we will refine that next time

# Exercise: Parameter Flows

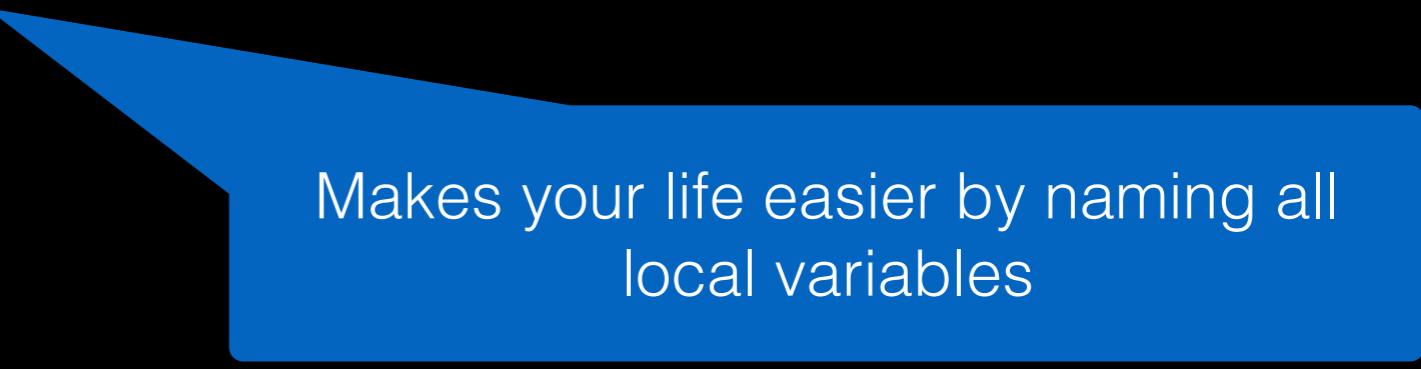
## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```



Makes your life easier by naming all local variables

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

```
define i32 @fib(i32 %n) #0 {  
bb:  
    %tmp = alloca i32, align 4  
    %tmp1 = alloca i32, align 4  
    store i32 %n, i32* %tmp1, align 4  
    %tmp2 = load i32, i32* %tmp1, align 4  
    ...
```

Makes your life easier by naming all local variables

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

```
define i32 @fib(i32 %n) #0 {  
bb:  
    %tmp = alloca i32, align 4  
    %tmp1 = alloca i32, align 4  
    store i32 %n, i32* %tmp1, align 4  
    %tmp2 = load i32, i32* %tmp1, align 4  
    ...
```



Makes your life easier by naming all local variables

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

```
define i32 @fib(i32 %n) #0 {  
bb:  
  %tmp = alloca i32, align 4  
  %tmp1 = alloca i32, align 4  
  store i32 %n, i32* %tmp1, align 4  
  %tmp2 = load i32, i32* %tmp1, align 4  
  ...
```

Makes your life easier by naming all local variables

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

```
define i32 @fib(i32 %n) #0 {  
bb:  
  %tmp = alloca i32, align 4  
  %tmp1 = alloca i32, align 4  
  store i32 %n, i32* %tmp1, align 4  
  %tmp2 = load i32, i32* %tmp1, align 4  
  ...
```

Makes your life easier by naming all local variables

# Exercise: Parameter Flows

## Example

```
$ opt-3.8 -instnamer < ~/examples/fib.bc > fib-named.bc
```

```
define i32 @fib(i32 %n) #0 {  
bb:  
  %tmp = alloca i32, align 4  
  %tmp1 = alloca i32, align 4  
  store i32 %n, i32* %tmp1, align 4  
  %tmp2 = load i32, i32* %tmp1, align 4  
  ...  
}
```

Makes your life easier by naming all local variables

Tracing argument n of function fib  
Storing tracked value n in pointer tmp1  
Loading tracked pointer tmp1 into value tmp2  
...

# Exercises in this Block

**exercise 6.1a**

Getting your environment ready



**exercise 6.1b**

**exercise 6.2**

Understanding basic tooling



**exercise 6.3**

Understanding optimization



**exercise 6.4**

Setting up your build env.



**exercise 6.5**

Instruction Histogram Analysis

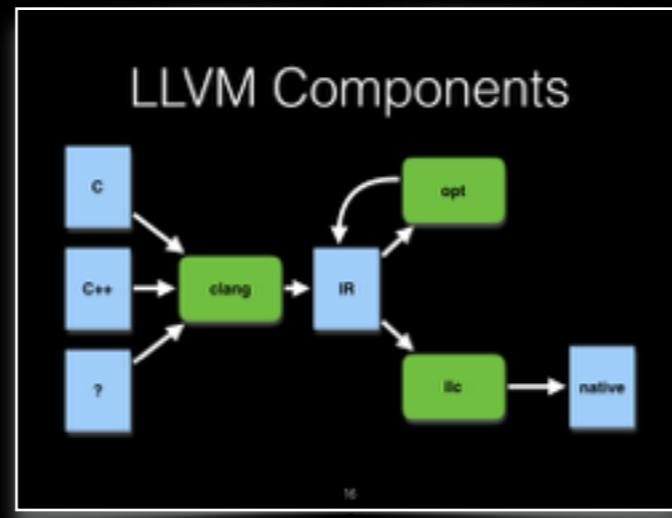


**exercise 6.6**

Parameter Data-flow Analysis



# Summary



Or use our Docker Container

```
docker run --rm -ti bhermann/llvm
```



- Time for coffee depends on your download speed
- Doing this right now will probably not work given the classroom's bandwidth

exercise 6.1b

## Basic Tooling

- Optimizing
  - clang -c -emit-llvm factorial.c
  - opt -O1 -o factorial-o1.bc factorial.bc
  - llvm-dis factorial.llc
  - llvm-dis factorial-o1.bc
- Compare **factorial.ll** and **factorial-o1.ll**
- Optimized version is shorter and contains phi instructions
- Continue for -O2 and -O3

exercise 6.3

## Exercise: Count Instructions

- Goal: Count instructions as a histogram per instruction type
- Scope: Complete code file
- Example: 14 store instructions, 14 load instructions, 5 cmp instructions, 4 br instructions

exercise 6.5

## Data-flow Analysis

- Goal: Reconstruct how data flows through a program
- Application
  - Find programming mistakes
  - Find security flaws (confidentiality, integrity)
  - Optimization

45