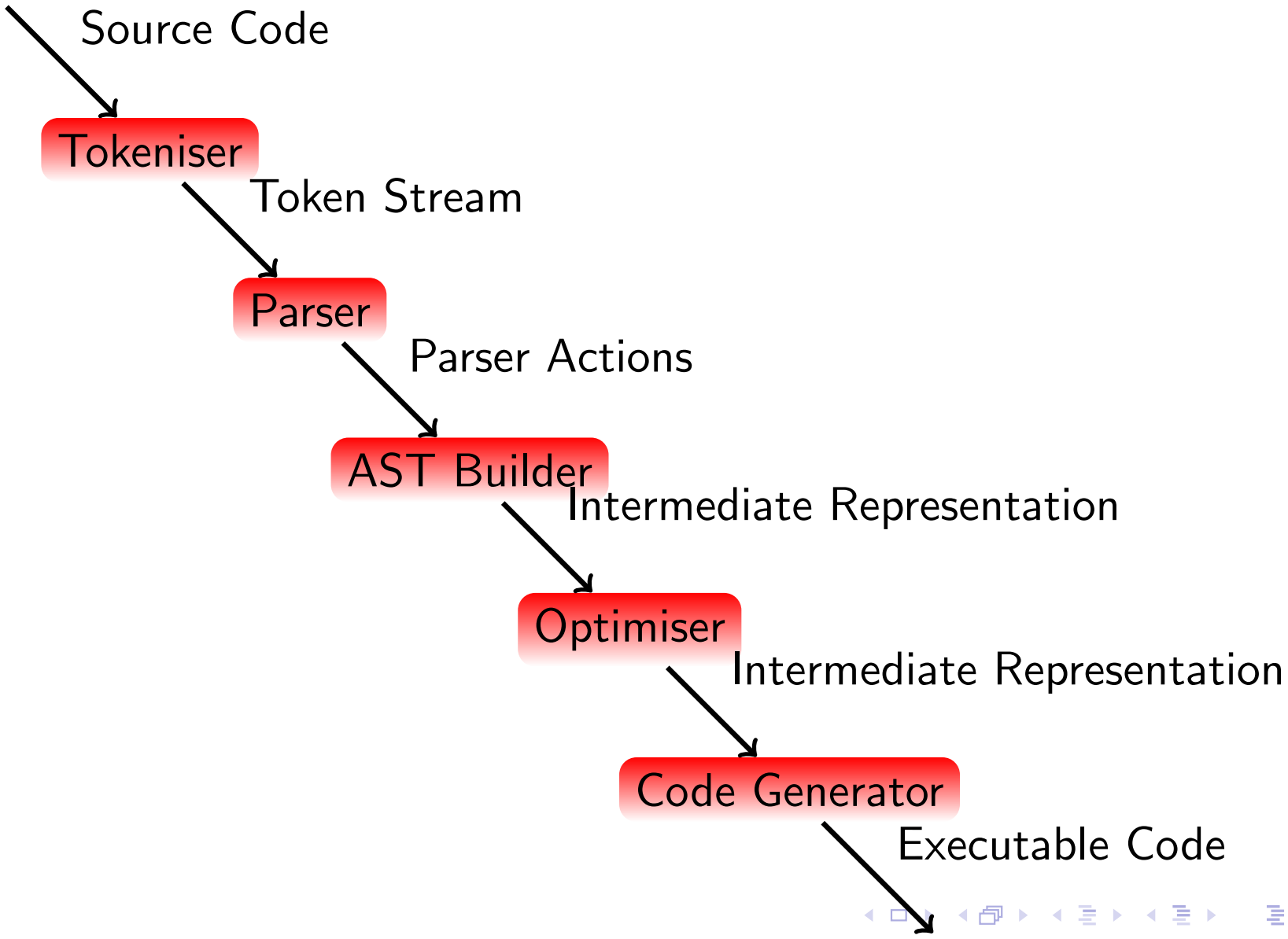
What LLVM Can Do For You

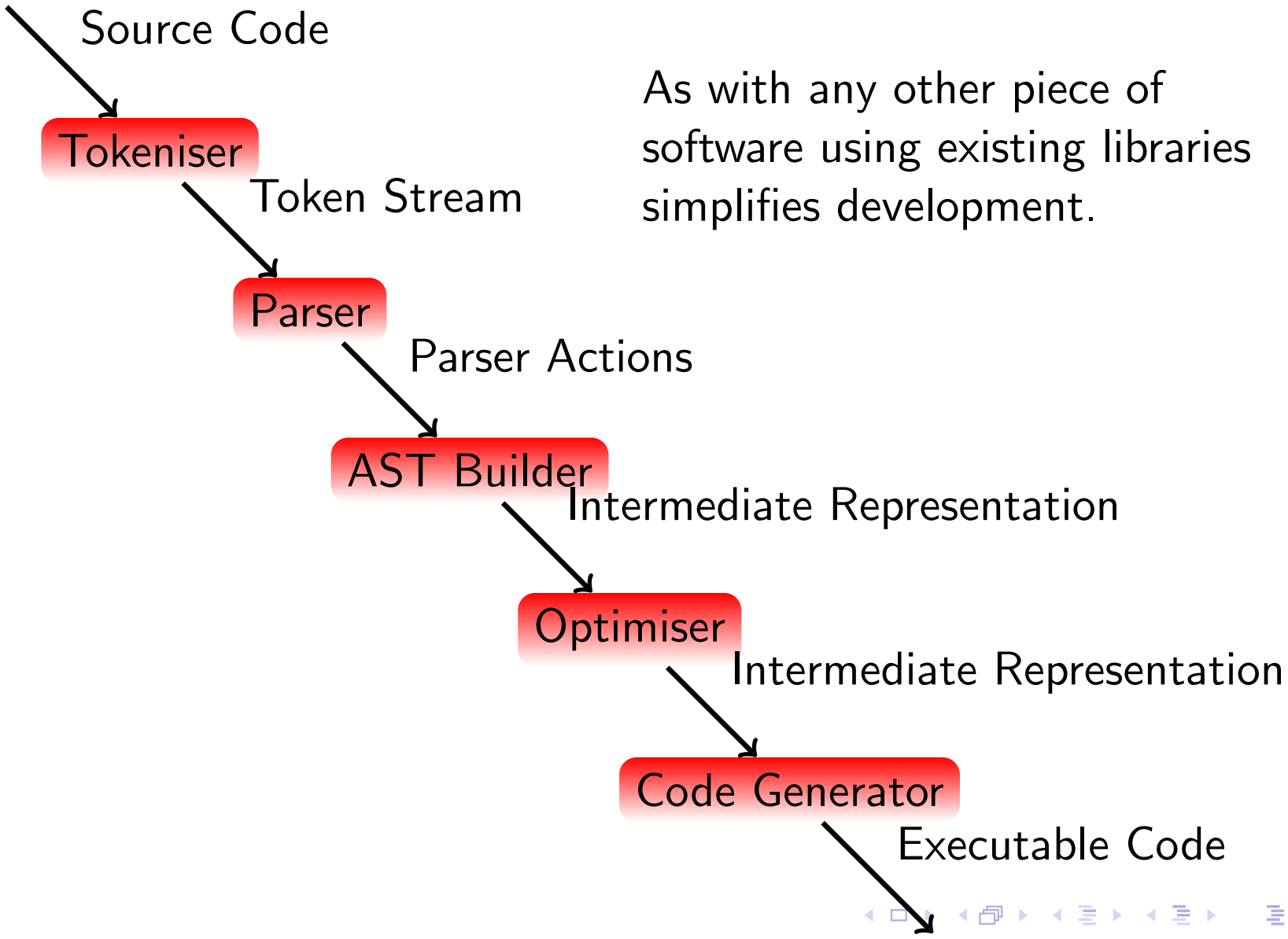
David Chisnall

April 13, 2013

Overview of a Compiler

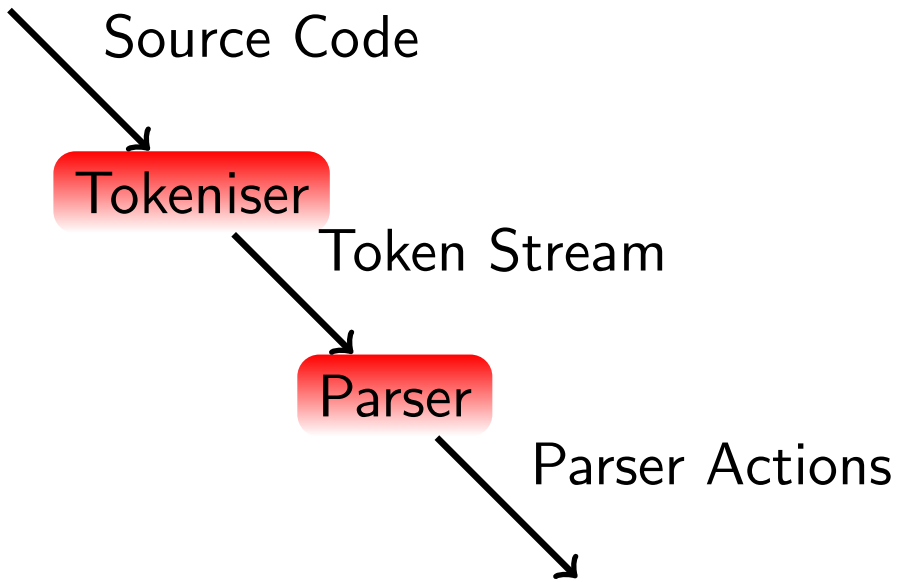


Overview of a Compiler



As with any other piece of software using existing libraries simplifies development.

Building a Front End



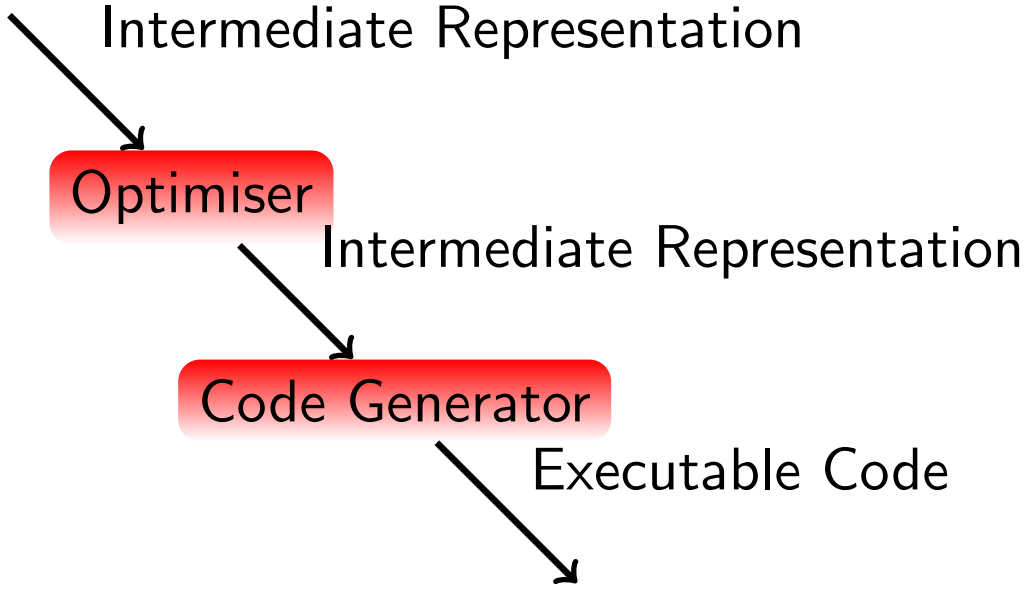
Many existing tools:

- Lex + yacc
- ANTLR
- OMeta
- ...

And the Middle?

- ASTs tend to be very language-specific
- You're (mostly) on your own there

What About the Back End?



This is where LLVM comes in.

What is LLVM?

- A set of libraries for implementing compilers
- Intermediate representation (LLVM IR) for optimisation
- Various helper libraries

Great for Compiler Writers!

- Other tools help you write the front end
- LLVM gives you the back end
- A simple compiler can be under 1000 lines of (new) code

What About Library Developers?

- LLVM optimisations are modular
- Does your library encourage some common patterns among users?
- Write an optimisation that makes them faster!

All programmers use compilers. Now all programmers can improve their compiler.

What Is LLVM IR?

- Unlimited Single-Assignment Register machine instruction set
- Three common representations:
  - Human-readable LLVM assembly (.ll files)
  - Dense 'bitcode' binary representation (.bc files)
  - C++ classes

Unlimited Register Machine?

- Real CPUs have a fixed number of registers
- LLVM IR has an infinite number
- New registers are created to hold the result of every instruction
- CodeGen's register allocator determines the mapping from LLVM registers to physical registers

Static Single Assignment

- Registers may be assigned to only once
- Most (imperative) languages allow variables to be... variable
- This requires some effort to support in LLVM IR

Multiple Assignment

```
int a = someFunction();
a++;
```

- One variable, assigned to twice.

Translating to LLVM IR

```
%a = call i32 @someFunction()
%a = add i32 %a, 1
```

```
error: multiple definition of local value named 'a'
  %a = add i32 %a, 1
     ^
```

Translating to *Correct* LLVM IR

```
%a = call i32 @someFunction()
%a2 = add i32 %a, 1
```

- How do we track the new values?

Translating to LLVM IR The Easy Way

```
; int a
%a = alloca i32, align 4
; a = someFunction
%0 = call i32 @someFunction()
store i32 %0, i32* %a
; a++
%1 = load i32* %a
%2 = add i32 %0, 1
store i32 %2, i32* %a
```

- Numbered register are allocated automatically
- Each expression in the source is translated without worrying about data flow
- Memory is not SSA in LLVM

Isn't That Slow?

- Lots of redundant memory operations
- Stores followed immediately by loads
- The mem2reg pass cleans it up for is

```
%0 = call i32 @someFunction()
%1 = add i32 %0, 1
```

Sequences of Instructions

- A sequence of instructions that execute in order is a *basic block*
- Basic blocks must end with a terminator
- Terminators are flow control instructions.
- `call` is not a terminator because execution resumes at the same place after the call

Intraprocedural Flow Control

- Assembly languages typically manage flow control via jumps / branches
- LLVM IR has conditional and unconditional branches
- Branch instructions go at the end of a basic block
- Basic blocks are branch targets
- You can't jump into the middle of a basic block

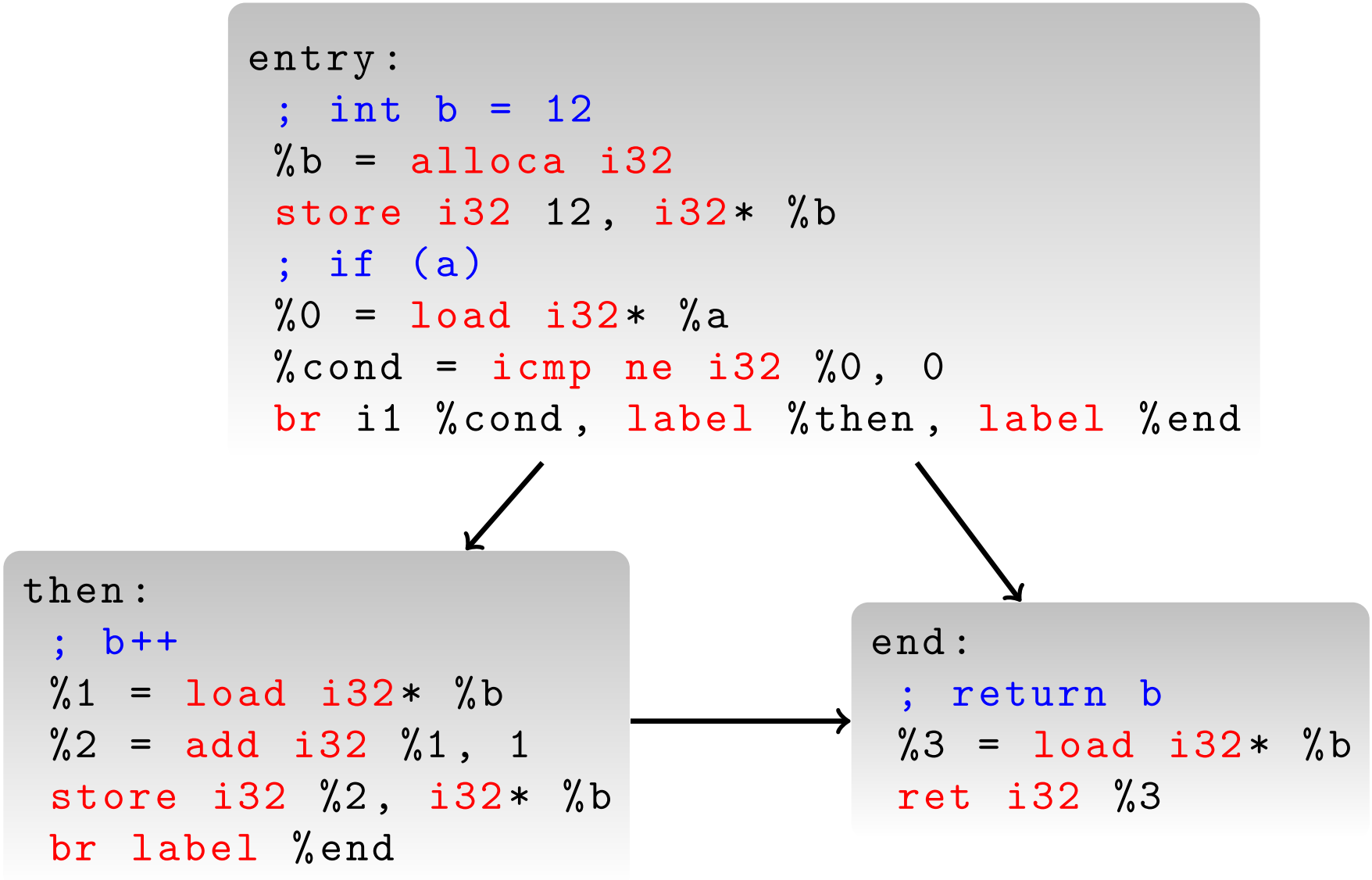
What About Conditionals?

```
int b = 12;
if (a)
    b++;
return b;
```

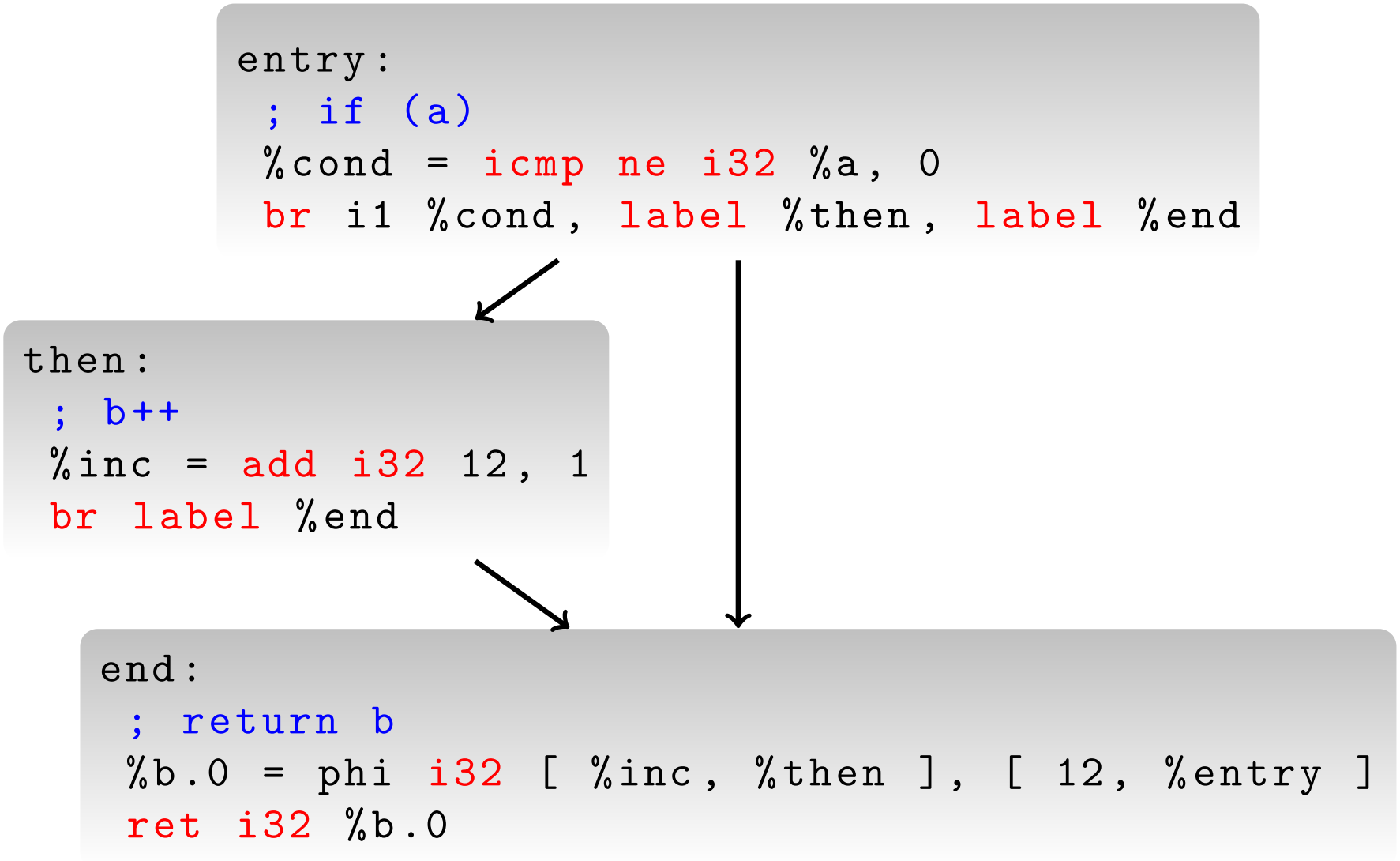
- Flow control requires one basic block for each path
- Conditional branches determine which path is taken

Phi, my lord, phi! - Lady Macbeth

- PHI nodes are special instructions used in SSA construction
- Their value is determined by the preceding basic block
- PHI nodes must come before any non-PHI instructions in a basic block


entry:
; int b = 12
%b = alloca i32
store i32 12, i32* %b
; if (a)
%0 = load i32* %a
%cond = icmp ne i32 %0, 0
br i1 %cond, label %then, label %end
then:
; b++
%1 = load i32* %b
%2 = add i32 %1, 1
store i32 %2, i32* %b
br label %end
end:
; return b
%3 = load i32* %b
ret i32 %3

In SSA Form...


entry:
; if (a)
%cond = icmp ne i32 %a, 0
br i1 %cond, label %then, label %end
then:
; b++
%inc = add i32 12, 1
br label %end
end:
; return b
%b.0 = phi i32 [%inc, %then], [12, %entry]
ret i32 %b.0

In SSA Form...

```
entry:
  ; if (a)
  %cond = icmp ne i32 %a, 0
  br i1 %cond, label %then, label %end
```

```
then:
  ; b++
  %inc = add i32 12, 1
  br label %end
```

```
end:
  ; return b
  %b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]
  ret i32 %b.0
```

The output from the mem2reg pass

And After Constant Propagation...

```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 br label %end
```

The output from the constprop pass. No add instruction.

```
end:
 ; b++
 ; return b
 %b.0 = phi i32 [ 13, %then ], [ 12, %entry ]
 ret i32 %b.0
```

And After CFG Simplification...

```
entry:
  %tobool = icmp ne i32 %a, 0
  %0 = select i1 %tobool, i32 13, i32 12
  ret i32 %0
```

- Output from the simplifycfg pass
- No flow control in the IR, just a `select` instruction

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.

Functions

- LLVM functions contain at least one basic block
- Arguments are explicitly typed

```
@hello = private constant [13 x i8] c"Hello
    world!\00"

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = getelementptr [13 x i8]* @hello, i32 0,
      i32 0
  call i32 @puts(i8* %0)
  ret i32 0
}
```

Get Element Pointer?

- Often shortened to GEP (in code as well as documentation)
- Represents pointer arithmetic
- Translated to complex addressing modes for the CPU

F!@£ing GEPs! HOW DO THEY WORK?!?

```
struct a {
    int c;
    int b[128];
} a;
int get(int i) { return a.b[i]; }
```

```
%struct.a = type { i32, [128 x i32] }

define i32 @get(i32 %i) nounwind readonly {
entry:
  %arrayidx = getelementptr %struct.a* @a, i32
      0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

As x86 Assembly

```
define i32 @get(i32 %i) nounwind readonly {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
      @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx, align 4, !tbaa !0
  ret i32 %0
}
```

```
get:
    movl    4(%esp), %eax           # load parameter
    movl    a+4(,%eax,4), %eax      # GEP + load
    ret
```

As ARM Assembly

```
define i32 @get(i32 %i) nounwind readonly {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
      @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx, align 4, !tbaa !0
  ret i32 %0
}
```

```
get:
    ldr     r1, .LCPI0_0        // Load global address
    add     r0, r1, r0, lsl #2 // GEP
    ldr     r0, [r0, #4]        // load return value
    bx      lr
.LCPI0_0:
    .long    a
```

Questions?