Go 1.7's New Compiler Backend

Evan Shaw

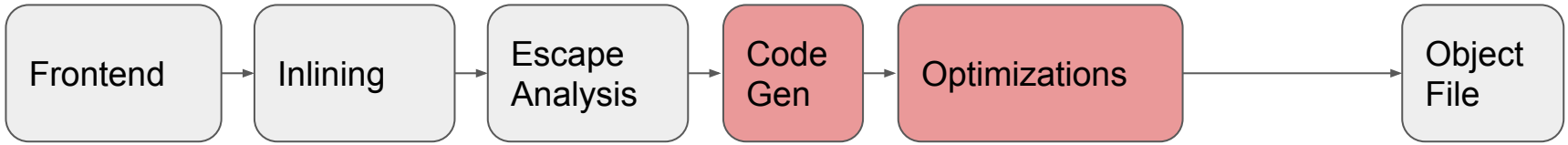
Go 1.7 Compiler Improvements

Compiled programs up to 20-30% smaller

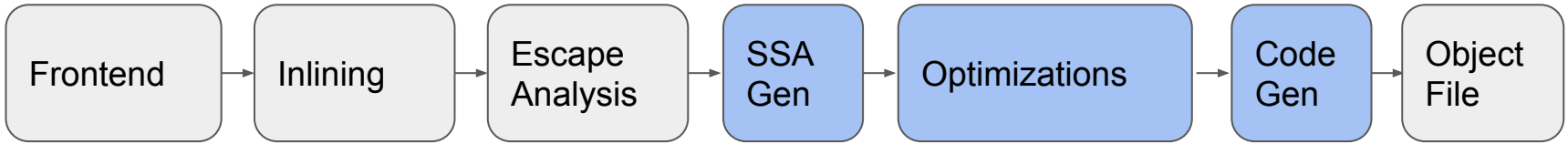
Compiled programs up to 35% faster

Compile times decreased by about 20%

Pre-Go 1.7 Compiler



Go 1.7 Compiler



Static Single Assignment (SSA) Form

Used by most modern compilers (gcc, llvm, hotspot, and more)

A variable becomes one or more values.

A value can only be assigned once.

Each assignment is a simple expression and does exactly one thing.

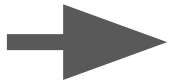
Code is divided into basic blocks.

Simple SSA Example

Go

```
x := 1
y := 2*x + 3
x += 4
```

SSA Conversion



SSA

```
x1   := 1
tmp1 := 2 * x1
y1   := tmp1 + 3
x2   := x1 + 4
```

Basic Blocks and Φ (Phi) Instructions

A basic block is a chunk of code that executes in sequence.

It's terminated by a branch, panic, or function call.

Basic blocks form the vertices of a graph called a **Control Flow Graph** (CFG).

Φ chooses a value from a predecessor block based on control flow.

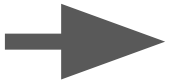
Φ helps when it comes time to allocate registers for values.

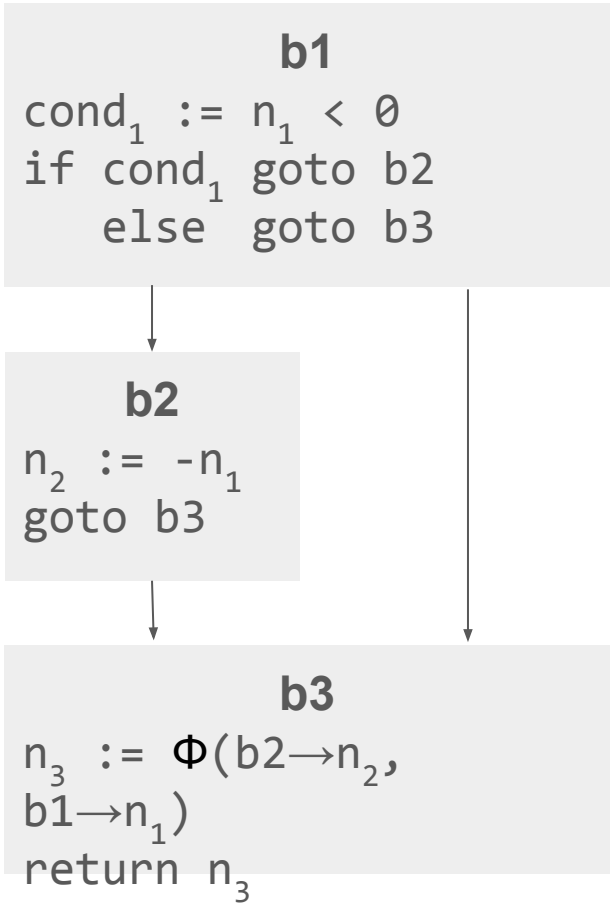
(Don't worry, examples are coming.)

Branch CFG Example

```
func abs(n int) int {
    if n < 0 {
        n = -n
    }
    return n
}
```

SSA Conversion

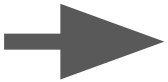


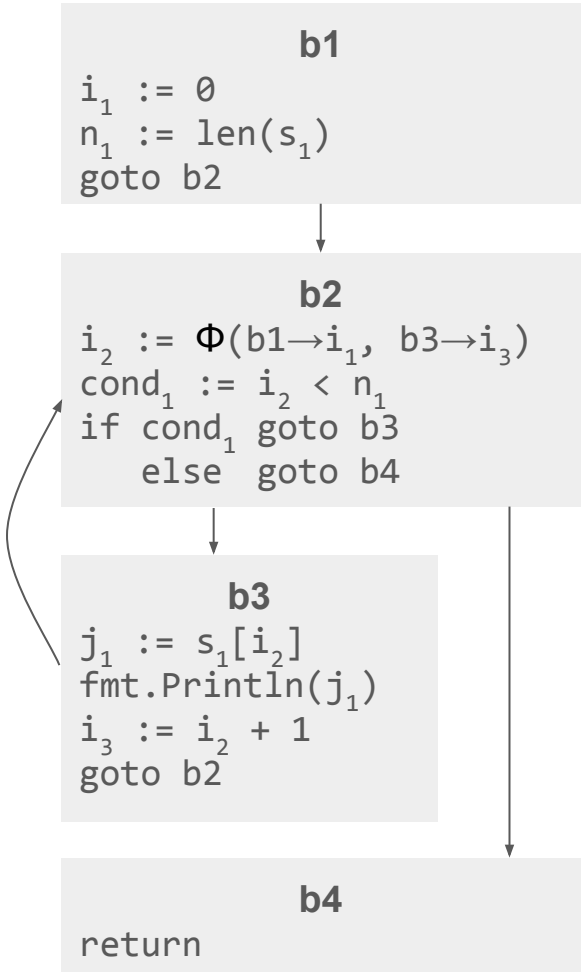


Loop CFG Example

```
func printSlice(s []int) {
    for _, j := range s {
        fmt.Println(j)
    }
}
```

SSA Conversion





Old: `peep.go` Optimizations

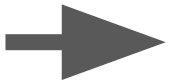
Simple, CPU-specific optimizations.

Name comes from the term "peephole optimization".

Examples:

```
ADDQ 1, R11
```

Peep



```
INCQ R11
```

```
MOVQ 3, R13
MOVQ 2, R13
```

```
MOVQ 2, R13
```

New: Pass Pipeline

Optimization and code gen are performed as a pipeline of passes over a function.

There are nearly 40 passes; each one has a very specific purpose.

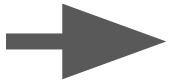
The majority of passes are CPU-independent.

Common Subexpression Elimination (CSE)

Go

```
x := 2*i + 1
y := 2*i + 2
```

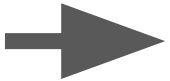
SSA



SSA

```
tmp1 := 2 * i
x1   := tmp1 + 1
tmp2 := 2 * i
y1   := tmp2 + 2
```

CSE Pass



SSA

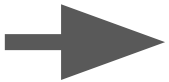
```
tmp1 := 2 * i
x1   := tmp1 + 1
tmp2 := 2 * i
y1   := tmp1 + 2
```

Dead Code Elimination (DCE)

SSA

tmp_1 := 2 * i
x_1 := tmp_1 + 1
tmp_2 := 2 * i
y_1 := tmp_1 + 2

DCE Pass



SSA

tmp_1 := 2 * i
x_1 := tmp_1 + 1
y_1 := tmp_1 + 2

Rewrite Rules

There is a Lisp-like DSL for certain optimizations, including constant propagation and strength reduction.

```
(Add8 (Const8 [c]) (Const8 [d]))
    -> (Const8 [int64(int8(c+d))])

(Mod64u <t> n (Const64 [c])) && isPowerOfTwo(c)
    -> (And64 n (Const64 <t> [c-1]))
```

These rules are used to generate Go code that runs inside the compiler.

Lowering Rules

The same DSL is used to lower instructions from CPU-independent SSA to CPU instructions.

```
(Add64 x y) -> (ADDQ x y)

(Eq64  x y) -> (SETEQ (CMPQ x y))
```

Demo: See SSA Output

(Use GOSSAFUNC env variable)

Compiler's Future

More CPU architectures

More optimizations (bounds check elimination!)

New parser

Move inlining and escape analysis?

The End.
Questions?

Appendix 1: 30-Second Overview of (Most) CPUs

Memory is a giant array of bytes; pointers are indexes into the array

There are also registers, which can hold values (usually 32-bit or 64-bit)

Values generally need to be in registers for you to be able to operate on them

CPU instructions roughly fall into three categories: memory, logic, and control flow