Optimization Overview

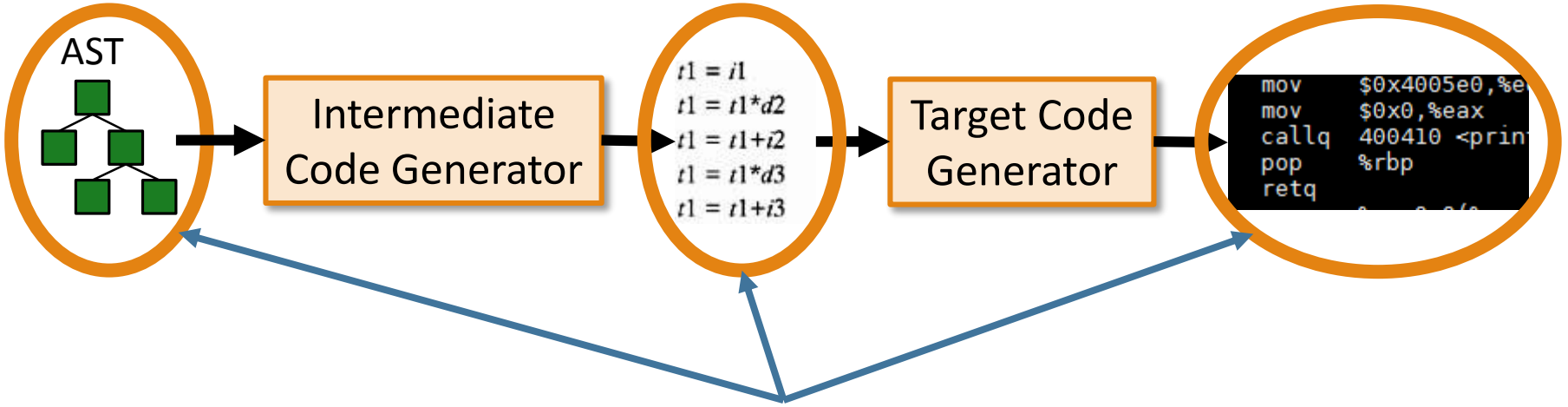
But first…

Questions about code generation?

Optimization Overview

FOR REAL THIS TIME

Back-End Overview



Apply optimizations to any/all of these representations.

Intermediate Representations

Abbreviated ***IR*** (or ***IL*** for Intermediate Language).

Generally each compiler has its own.

- We basically use Cool Asm as our IR.

Advantages:

- Machine independent: one optimization for Cool and x86_64.
- Exposes more opportunities than AST.

Optimization Goals

1. **Get the right answer**.
   - No, really. Get the right answer.
2. Get it quickly:
   - Remove redundant work.
   - Do the remaining work “better”.

Optimization Goals

1. **Get the right answer**.
   - Or don't bother with goal 2.
2. Get it quickly:
   - Remove redundant work.
   - Do the remaining work "better".

"Better":
- In fewer cycles.
- Using less power.
- With fewer instructions.
- Less network traffic.
- …

Sort Analogy

How is qsort implemented in practice?

1. Use quicksort for high-level sort.
   - Low complexity (Remove redundant work).
2. Use insertion sort for base case.
   - Small constant (Make remaining work fast).

We will follow a similar approach in our optimizing compiler.

Peephole Optimizations

1. Slide window (peephole) over representation.
2. Pattern match and replace with optimized code.
3. Repeat.

What can we do with this?

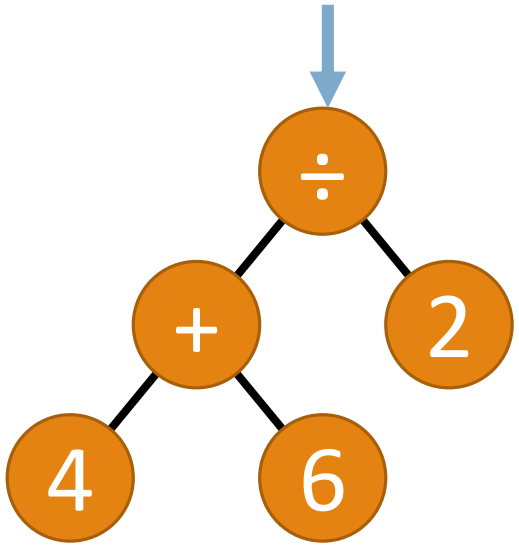
- Constant folding.
- Eliminate redundant ops.
- Strength reduction.
- Algebraic simplification.
- Apply machine idioms.

Constant Folding

Depth-first, post-order walk.

Match subtree and replace.

- E.g. binop with two constants.
- E.g. if-expression with constant predicate.

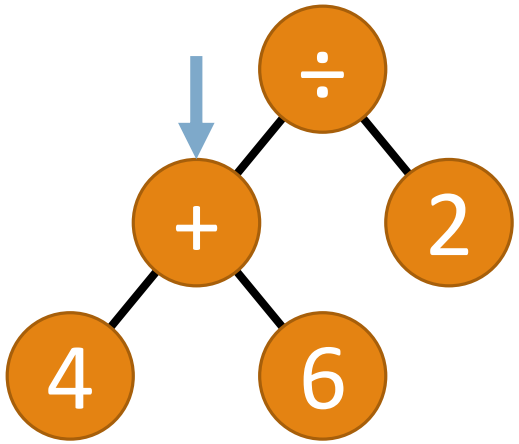


Constant Folding

Depth-first, post-order walk.

Match subtree and replace.

- E.g. binop with two constants.
- E.g. if-expression with constant predicate.

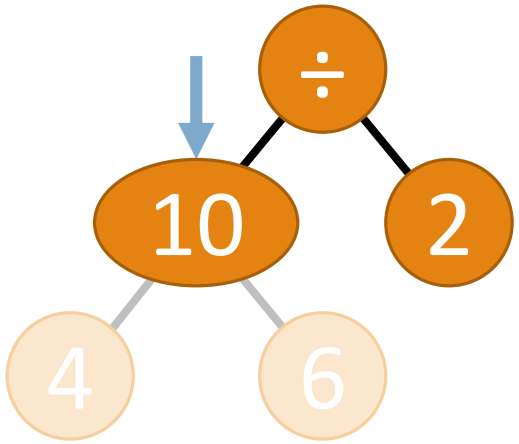


Constant Folding

Depth-first, post-order walk.

Match subtree and replace.

- E.g. binop with two constants.
- E.g. if-expression with constant predicate.

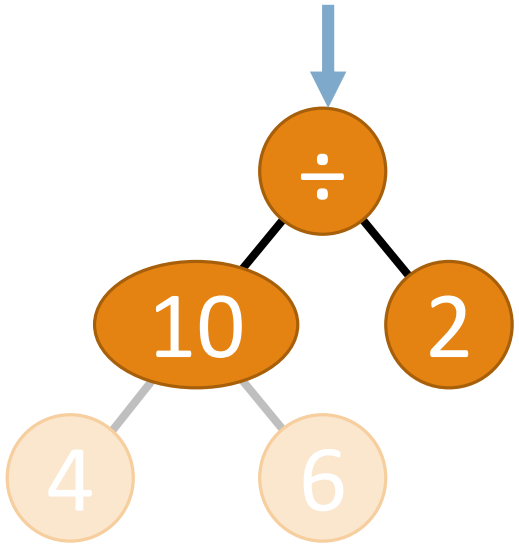


Constant Folding

Depth-first, post-order walk.

Match subtree and replace.

- E.g. binop with two constants.
- E.g. if-expression with constant predicate.

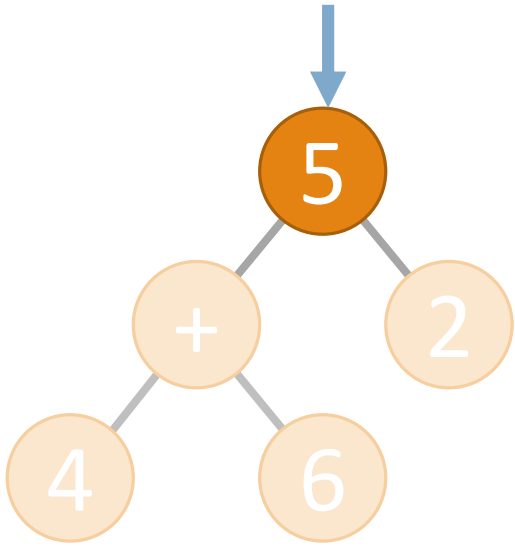


Constant Folding

Depth-first, post-order walk.

Match subtree and replace.

- E.g. binop with two constants.
- E.g. if-expression with constant predicate.



Eliminating Redundant Ops

Many sequences, e.g.:

- ~~`pop rX; push rX`~~
- ~~`push rX; pop rX`~~
- `ld rX <- rY[Z]`
  ~~`st rY[Z] <- rX`~~

can be deleted.

One pass may enable others.

```
…
push r0
push r1
call foo
pop r1
pop r0
push r0
push r1
```

```
call bar
pop r1
pop r0
…
```

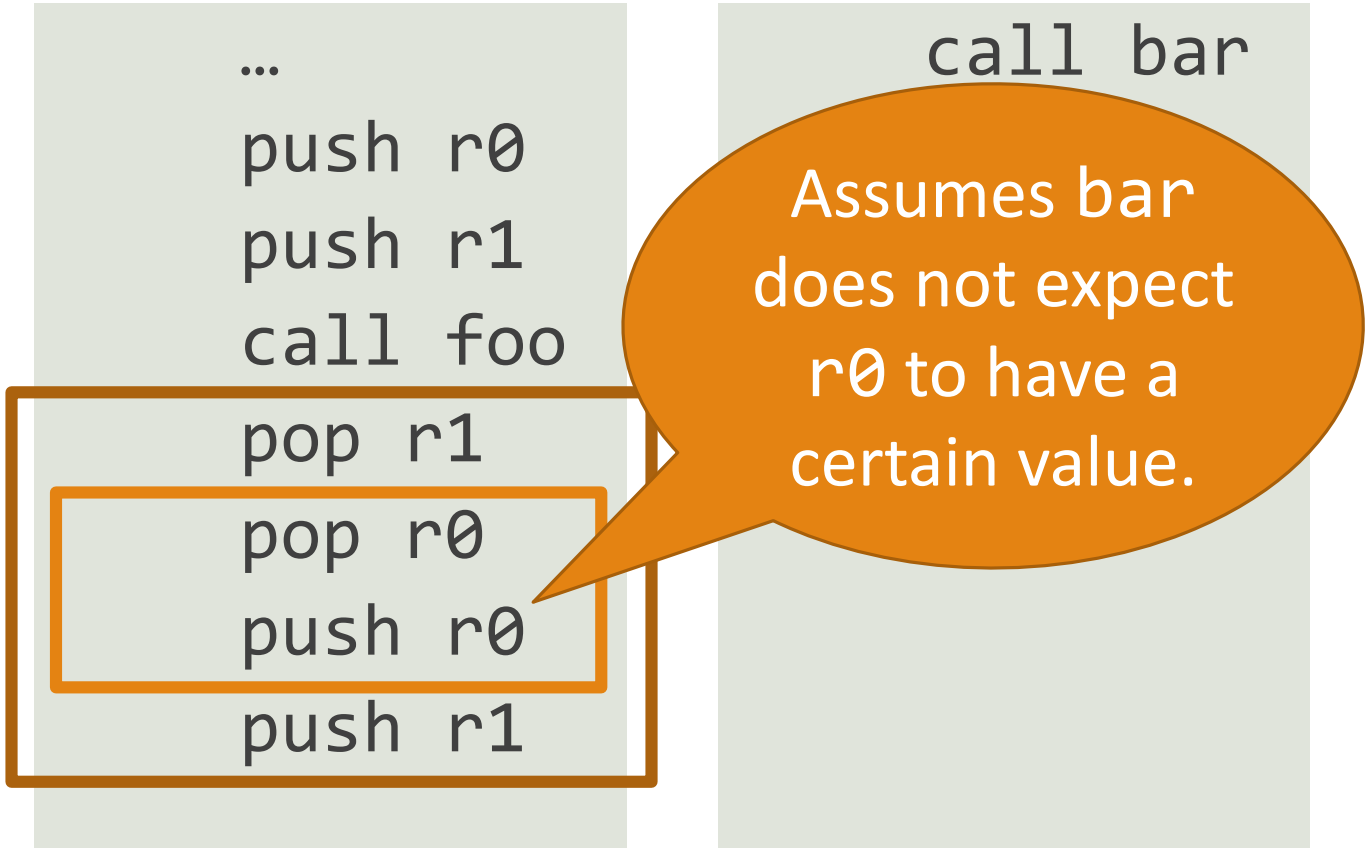
Eliminating Redundant Ops

Many sequences, e.g.:

- ~~`pop rX; push rX`~~
- ~~`push rX; pop rX`~~
- `ld rX <- rY[Z]`
  ~~`st rY[Z] <- rX`~~

can be deleted.

One pass may enable others.



Eliminating Redundant Ops

Many sequences, e.g.:

- ~~`pop rX; push rX`~~
- ~~`push rX; pop rX`~~
- `ld rX <- rY[Z]`
  ~~`st rY[Z] <- rX`~~

can be deleted.

Be careful of labels!

```
        …
        push r0
        push r1
        call foo
        pop r1
        pop r0
X:      push r0
        push r1
```

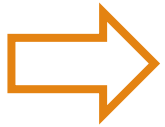
```
        call bar
        pop r1
        pop r0
        …
```

Strength Reduction & Algebraic Simplification

Replace expensive operations with cheaper equivalents.

```
x *= 3
```

```
li r1 <- 3
mul r0 <- r0 r1
```

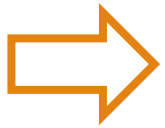


```
t = x + x; x += t
```

```
add r1 <- r0 r0
add r0 <- r1 r0
```

```
x *= 16
```

```
imulq $16, %rax
```



```
x = x << 4
```

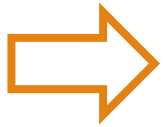
```
salq $4, %rax
```

Strength Reduction & Algebraic Simplification

Replace expensive operations with cheaper equivalents.

```
x *= 3
```

```
li r1 <- 3
mul r0 <- r0 r1
```



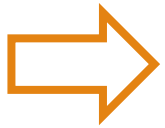
```
t = x + x; x += t
```

```
add r1 <- r0 r0
add r0 <- r1 r0
```

Saves 10 cycles!

```
x *= 16
```

```
imulq $16, %rax
```



```
x = x << 4
```

```
salq $4, %rax
```

Saves ?? cycles?

Machine Idioms

Hardware-specific alternatives.

- Smaller code (CISC).
- Faster hardware paths.

```
imulq $3, %rax
add $1, %rax
```

```
leaq 1(%rax,%rax,2), %rax
```

```
mov %rax, $0
```

```
xor %rax, %rax
```

Optimization Classification

Optimizations are classified by scope or increasing complexity:

1. ***Local***: small blocks of code ("basic blocks").
   - Includes peephole optimizations.
2. "***Global***:" method bodies ("control flow graph").
3. ***Inter-procedural***: compilation unit.
   - Crosses method boundaries.

Basic Blocks

Maximal sequence of IR instructions with:

- ***No jumps*** (except optionally at last instruction).
- ***No labels*** (except optionally at first instruction).

Control can only ***enter*** block through ***first instruction***.

Control can only ***leave*** block through ***last instruction***.

Therefore, if ***any*** instructions are executed, ***all are***.

Identifying Basic Blocks

1. Identify leaders (first instruction in each basic block).
   - First instruction.
   - Targets of branches ($\subseteq$ labeled instructions).
   - Instructions following branches.
2. Block contains leader up to (but excluding) next leader.

Local Optimizations in Basic Blocks

Useful property: **If any instructions are executed, all are**.

For example, determine when values will be next used.

- Keep frequently used values in registers.
- Reuse registers holding “dead” values.

These two are weaker versions of global optimizations.

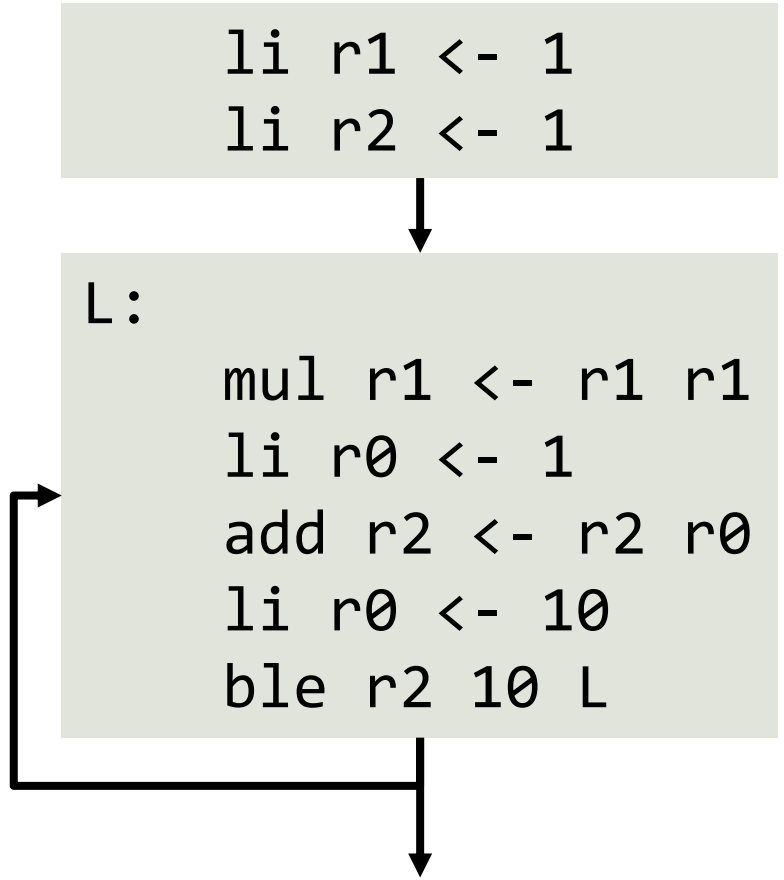
- So let’s talk about those instead.

Control Flow Graphs

Nodes: basic blocks.

Edge from B1 to B2 if:

- Conditional or unconditional jump from B1 to B2.
- B2 follows B1 and B1 does not end in conditional jump.



Next Week…

Data-flow analysis.