Local Optimizations

Martin Kellogg

Course Announcements

- Graded **midterms** are at the front of the room
  - If you don't have it yet, pick it up after class
  - If you take it with you, I won't accept regrade requests

Course Announcements

- Graded **midterms** are at the front of the room
  - If you don't have it yet, pick it up after class
  - If you take it with you, I won't accept regrade requests
- A problem with the PA3c3 autograder was found over the weekend
  - I've therefore granted an extension to **Wednesday** (AoE)
    - Same extension for PA3

Course Announcements

- Graded **midterms** are at the front of the room
  - If you don’t have it yet, pick it up after class
  - If you take it with you, I won’t accept regrade requests
- A problem with the PA3c3 autograder was found over the weekend
  - I’ve therefore granted an extension to **Wednesday** (AoE)
    - Same extension for PA3
- We recently fixed a bug in the reference compiler’s x86-64 module. Only use Cool **version 1.39** for compiling to x86.

Agenda

- Introduction to optimization
    - Goals
    - Taxonomy
    - Problems

Agenda

- Introduction to optimization
  - Goals
  - Taxonomy
  - Problems
- Local Optimizations
  - Survey of local optimizations
  - Some examples

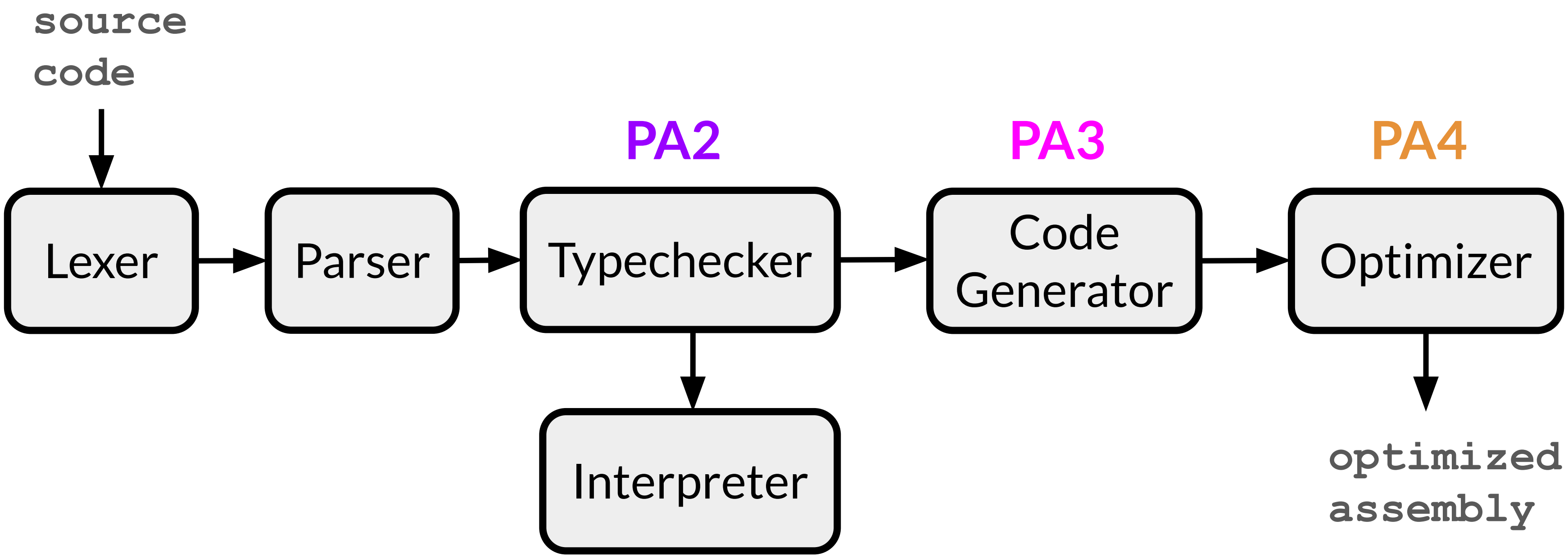
Agenda

- Introduction to optimization
  - Goals
  - Taxonomy
  - Problems
- Local Optimizations
  - Survey of local optimizations
  - Some examples
- Next time: local value numbering in more detail
  - plus some regional optimizations

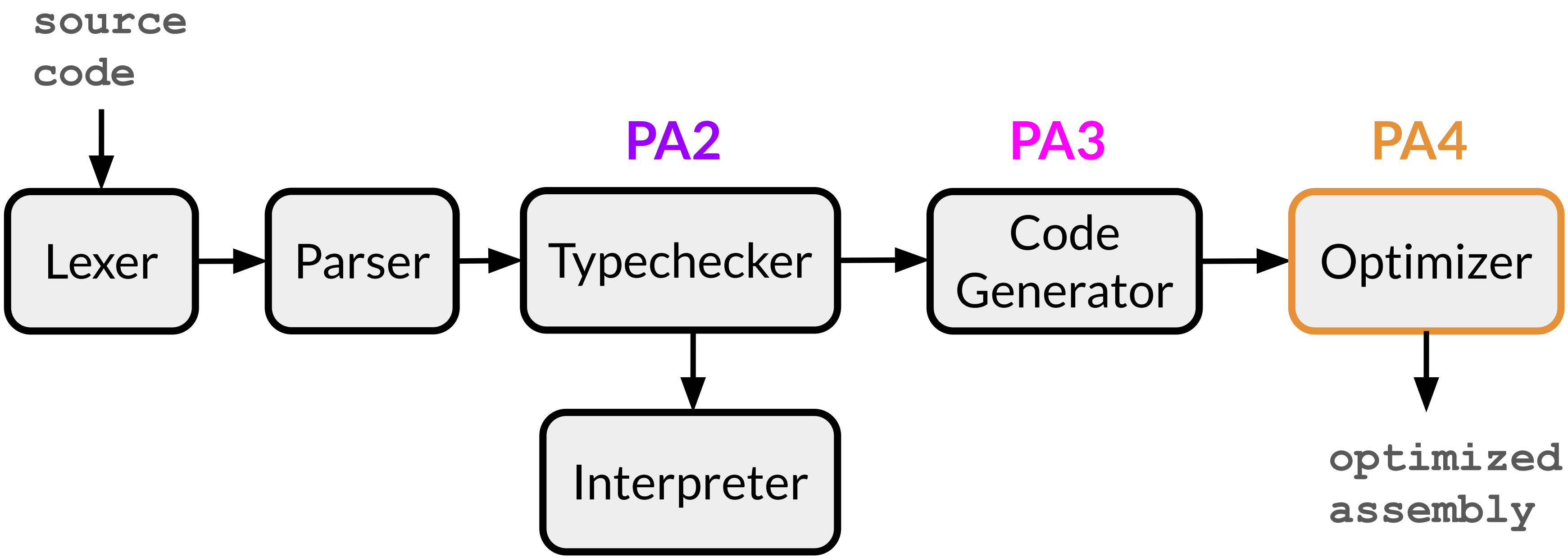
Agenda

- **Introduction to optimization**
  - Goals
  - Taxonomy
  - Problems
- Local Optimizations
  - Survey of local optimizations
  - Some examples
- Next time: local value numbering in more detail
  - plus some regional optimizations

Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Optimization

- A compiler has two main goals:

Optimization

- A compiler has two main goals:
  1. to **translate** the program to a “lower” level of abstraction

Optimization

- A compiler has two main goals:
  1. to **translate** the program to a “lower” level of abstraction
  2. to **improve** the program in some way (while preserving semantics)

Optimization

- A compiler has two main goals:
  1. to **translate** the program to a “lower” level of abstraction
  2. to **improve** the program in some way (while preserving semantics)
- The latter–improving the program–is the ***optimization*** stage of the compiler

Optimization

- A compiler has two main goals:
  1. to **translate** the program to a “lower” level of abstraction
  2. to **improve** the program in some way (while preserving semantics)
- The latter–improving the program–is the ***optimization*** stage of the compiler
  - “optimization” is a bad name: there’s nothing “optimal” about the result

Optimization

- A compiler has two main goals:
  1. to **translate** the program to a “lower” level of abstraction
  2. to **improve** the program in some way (while preserving semantics)
- The latter–improving the program–is the ***optimization*** stage of the compiler
  - “optimization” is a bad name: there’s nothing “optimal” about the result
  - “**program improvement**” is probably more apt

Optimization: Formalizing Our Goals

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:
  - Is the optimization ***safe***: that is, does it preserve the semantics of **this program**?

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:
    - Is the optimization ***safe***: that is, does it preserve the semantics of **this program**?
        - note that this means we can specialize to the program of interest!

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:
  - Is the optimization ***safe***: that is, does it preserve the semantics of **this program**?
    - note that this means we can specialize to the program of interest!
    - usually, we show this via a **proof**

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:
  - Is the optimization ***safe***: that is, does it preserve the semantics of **this program**?
    - note that this means we can specialize to the program of interest!
    - usually, we show this via a **proof**
  - Is the optimization ***profitable***: that is, does it make the program better in some way?

Optimization: Formalizing Our Goals

- For any optimization that we are considering, we have two high-level questions we need to consider:
    - Is the optimization ***safe***: that is, does it preserve the semantics of **this program**?
        - note that this means we can specialize to the program of interest!
        - usually, we show this via a **proof**
    - Is the optimization ***profitable***: that is, does it make the program better in some way?
        - common ways to define profit: fewer clock cycles, smaller binary size, uses less battery, etc.

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!
  - This is one of the major sources of optimization opportunities

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!
  - This is one of the major sources of optimization opportunities
- Many optimizations have **preconditions** under which they're safe

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!
  - This is one of the major sources of optimization opportunities
- Many optimizations have **preconditions** under which they're safe
  - you already have one tool for proving such preconditions: **abstract interpretation**

Optimization: Specialization

- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!
  - This is one of the major sources of optimization opportunities
- Many optimizations have **preconditions** under which they're safe
  - you already have one tool for proving such preconditions: **abstract interpretation**
    - later, we'll cover ***dataflow analysis***, a closely-related topic

Optimization: Specialization

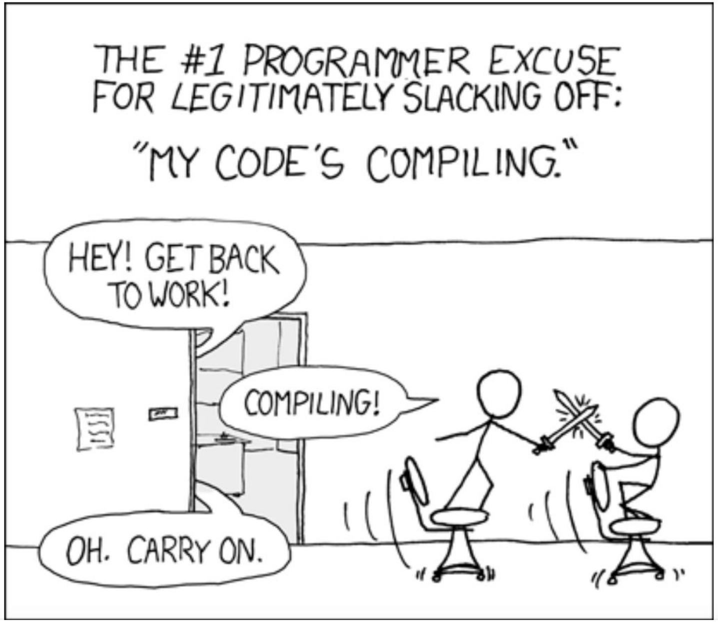
- Proving an optimization safe only requires us to show that it is safe for the **specific program** that we are compiling
  - For example, it's not generally safe to convert the expression `a + 3` into the constant `5`
    - but it is safe if we know that `a = 2` was just executed!
  - This is one of the major sources of optimization opportunities
- Many optimizations have **preconditions** under which they're safe
  - you already have one tool for proving such preconditions: **abstract interpretation**
    - later, we'll cover ***dataflow analysis***, a closely-related topic
  - you can also use facts from any previous compilation stage

Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
    - Why?

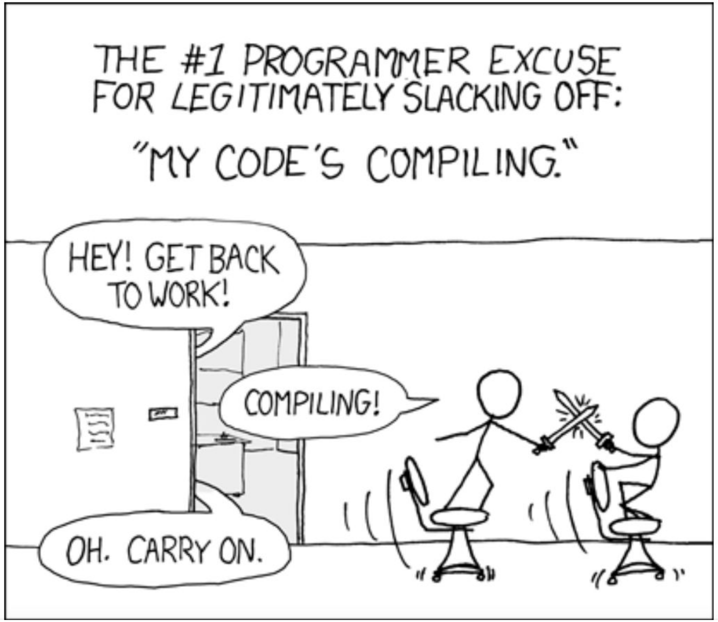
Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
  - Why?



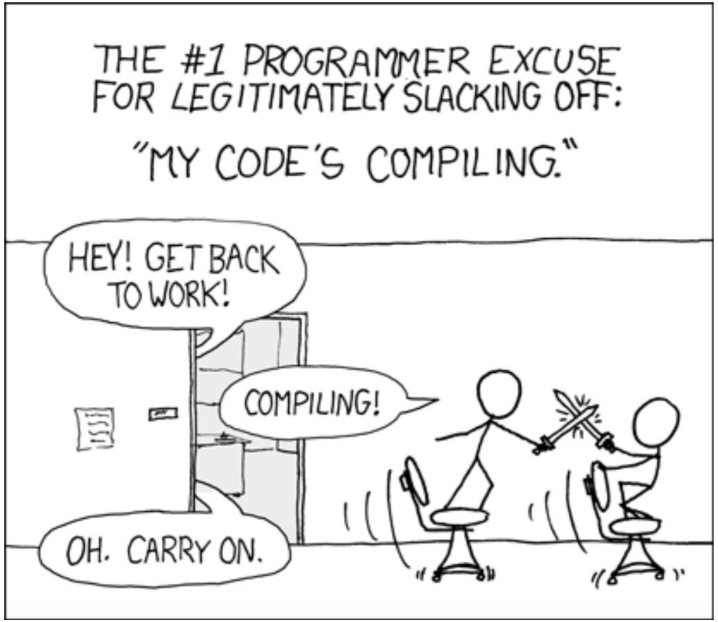
Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
  - Why?
- Some optimizations are hard to implement



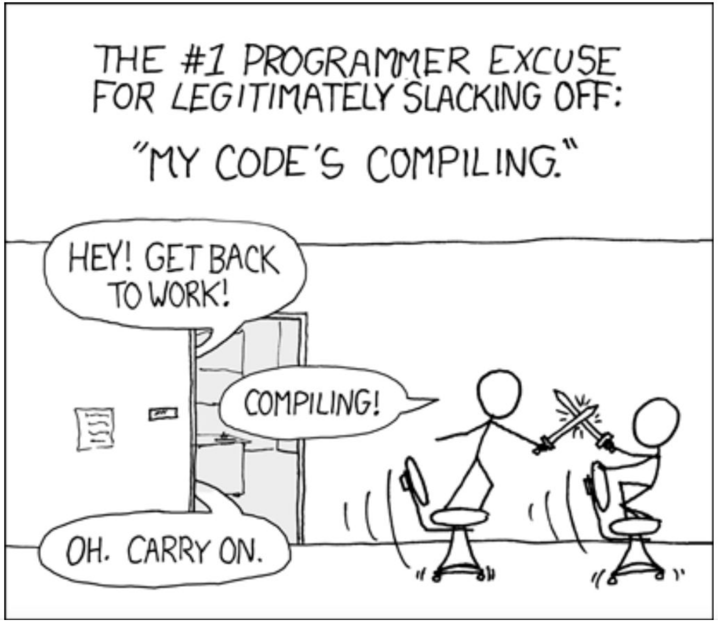
Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
    - Why?
- Some optimizations are hard to implement
- Some optimizations are costly in terms of compilation time



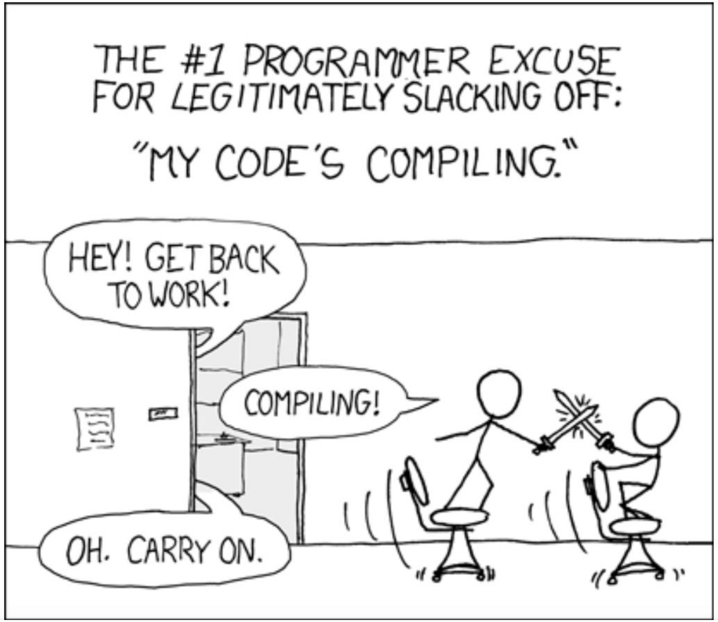
Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
  - Why?
- Some optimizations are hard to implement
- Some optimizations are costly in terms of compilation time
- The fancy optimizations are both hard and costly



Optimization: Cost

- In practice, a conscious decision is made **not** to implement the fanciest optimization known
  - Why?
- Some optimizations are hard to implement
- Some optimizations are costly in terms of compilation time
- The fancy optimizations are both hard and costly
- Our goal: maximum improvement with minimum of cost (no risk!)



Optimization: Sources of Opportunities

Optimization: Sources of Opportunities

- We’ve already seen one: ***specialization*** to the specific program
  - e.g., compute the results of constant expressions at compile time, just load the result into a register when you need it

Optimization: Sources of Opportunities

- We've already seen one: ***specialization*** to the specific program
    - e.g., compute the results of constant expressions at compile time, just load the result into a register when you need it
- A related, but somewhat orthogonal idea is ***reducing the overhead of abstractions***

Optimization: Sources of Opportunities

- We’ve already seen one: ***specialization*** to the specific program
  - e.g., compute the results of constant expressions at compile time, just load the result into a register when you need it
- A related, but somewhat orthogonal idea is ***reducing the overhead of abstractions***
  - Code gen (e.g., your PA3) is initially designed to produce the “most general form” of each language construct

Optimization: Sources of Opportunities

- We've already seen one: ***specialization*** to the specific program
  - e.g., compute the results of constant expressions at compile time, just load the result into a register when you need it
- A related, but somewhat orthogonal idea is ***reducing the overhead of abstractions***
  - Code gen (e.g., your PA3) is initially designed to produce the "most general form" of each language construct
  - Often, a simpler version will work instead
    - e.g., if you know that a pointer is definitely non-void, do you need to emit the code for dispatch on void?

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit “slow” code
  - e.g., the stack machine we looked at last week

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit “slow” code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer’s role is to “***clean up***” the overhead introduced by IRs to simplify code generation

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit “slow” code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer’s role is to “***clean up***” the overhead introduced by IRs to simplify code generation
  - sometimes that means modifying code gen (e.g., with an accumulator register), but not always

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit "slow" code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer's role is to "***clean up***" the overhead introduced by IRs to simplify code generation
  - sometimes that means modifying code gen (e.g., with an accumulator register), but not always
- The specific architecture of the ***target machine*** also offers opportunities for optimization

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit “slow” code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer’s role is to “***clean up***” the overhead introduced by IRs to simplify code generation
  - sometimes that means modifying code gen (e.g., with an accumulator register), but not always
- The specific architecture of the ***target machine*** also offers opportunities for optimization
  - e.g., if you know the machine has *k* physical registers, you can try to only use *k* registers at a time

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit "slow" code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer's role is to "***clean up***" the overhead introduced by IRs to simplify code generation
  - sometimes that means modifying code gen (e.g., with an accumulator register), but not always
- The specific architecture of the ***target machine*** also offers opportunities for optimization
  - e.g., if you know the machine has *k* physical registers, you can try to only use *k* registers at a time
  - some machines will have specific hardware, like floating point registers, that can accelerate certain operations

Optimization: Sources of Opportunities (2)

- Many IRs you might consider intentionally emit "slow" code
  - e.g., the stack machine we looked at last week
  - a big part of the optimizer's role is to "*clean up*" the overhead introduced by IRs to simplify code generation
  - sometimes that means modifying code gen (e.g., with an accumulator register), but not always
- The spe… *machine* also offers opportu…
  - e.g.,… physical registers, you can try t…
  - som… ardware, like floating point registers, that can accelerate certain operations

This list is not exhaustive! For PA4, you have a lot of **creative freedom** to choose what you want to optimize.

Optimization: When To Optimize

Optimization: When To Optimize

- On **AST** (just like type checking)

Optimization: When To Optimize

- On **AST** (just like type checking)
  - Pro: Machine independent
  - Con: Too high level

Optimization: When To Optimize

- On **AST** (just like type checking)
  - Pro: Machine independent
  - Con: Too high level
- On **assembly language** directly

Optimization: When To Optimize

- On **AST** (just like type checking)
  - Pro: Machine independent
  - Con: Too high level
- On **assembly language** directly
  - Pro: Exposes hardware optimization opportunities
  - Con: Machine dependent
  - Con: Must reimplement optimizations when retargeting

Optimization: When To Optimize

- On **AST** (just like type checking)
  - Pro: Machine independent
  - Con: Too high level
- On **assembly language** directly
  - Pro: Exposes hardware optimization opportunities
  - Con: Machine dependent
  - Con: Must reimplement optimizations when retargeting
- On an **intermediate language**

Optimization: When To Optimize

- On **AST** (just like type checking)
  - Pro: Machine independent
  - Con: Too high level
- On **assembly language** directly
  - Pro: Exposes hardware optimization opportunities
  - Con: Machine dependent
  - Con: Must reimplement optimizations when retargeting
- On an **intermediate language**
  - Pro: Machine independent
  - Pro: Exposes optimization opportunities
  - Con: One more language to worry about

Optimization vs "Backend"

- Most of the optimizations we're going to talk about today are "**IR-level**" optimizations

Optimization vs “Backend”

- Most of the optimizations we’re going to talk about today are “**IR-level**” optimizations
  - They make assumptions like “there are infinitely many registers and mapping those to physical registers is somebody else’s problem”

Optimization vs “Backend”

- Most of the optimizations we’re going to talk about today are “**IR-level**” optimizations
  - They make assumptions like “there are infinitely many registers and mapping those to physical registers is somebody else’s problem”
- The ***compiler backend*** is responsible for such decisions

Optimization vs “Backend”

- Most of the optimizations we’re going to talk about today are “**IR-level**” optimizations
  - They make assumptions like “there are infinitely many registers and mapping those to physical registers is somebody else’s problem”
- The ***compiler backend*** is responsible for such decisions
  - e.g., mapping abstract registers to “real” x86-64 registers (***register allocation***)

Optimization vs “Backend”

- Most of the optimizations we’re going to talk about today are “**IR-level**” optimizations
    - They make assumptions like “there are infinitely many registers and mapping those to physical registers is somebody else’s problem”
- The ***compiler backend*** is responsible for such decisions
    - e.g., mapping abstract registers to “real” x86-64 registers (***register allocation***)
- Many optimizations are possible at the backend level, too:
    - spill fewer registers
    - select “better” instructions
    - schedule instructions in a way that wastes fewer clock cycles

Optimization: Problems

Optimization: Problems

- Many optimizations possible, some depend on others
  - Which should we do first?

Optimization: Problems

- Many optimizations possible, some depend on others
  - Which should we do first?
  - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization

Optimization: Problems

- Many optimizations possible, some depend on others
  - Which should we do first?
  - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization
- Unclear profitability of some optimizations

Optimization: Problems

- Many optimizations possible, some depend on others
    - Which should we do first?
    - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization
- Unclear profitability of some optimizations
    - There is a ***risk*** that some optimizations may even make performance worse!

Optimization: Problems

- Many optimizations possible, some depend on others
  - Which should we do first?
  - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization
- Unclear profitability of some optimizations
  - There is a ***risk*** that some optimizations may even make performance worse!
    - Also unsolvable in general, and has bad interaction with phase-ordering

Optimization: Problems

- Many optimizations possible, some depend on others
    - Which should we do first?
    - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization
- Unclear profitability of some optimizations
    - There is a ***risk*** that some optimizations may even make performance worse!
        - Also unsolvable in general, and has bad interaction with phase-ordering
- Many optimizations are only safe if some condition is met

Optimization: Problems

- Many optimizations possible, some depend on others
  - Which should we do first?
  - This ***phase-ordering problem*** is one of the most difficult problems to solve in compiler optimization
- Unclear profitability of some optimizations
  - There is a ***risk*** that some optimizations may even make performance worse!
    - Also unsolvable in general, and has bad interaction with phase-ordering
- Many optimizations are only safe if some condition is met
  - we'll use **static analysis** to check -> we inherit all problems of static analysis (such as?)

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
    1. ***Peephole*** optimizations
        - Apply to a pair of adjacent instructions

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
    1. ***Peephole*** optimizations
        - Apply to a pair of adjacent instructions
    2. ***Local*** optimizations
        - Apply to a basic block in isolation

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
  1. ***Peephole*** optimizations
     - Apply to a pair of adjacent instructions
  2. ***Local*** optimizations
     - Apply to a basic block in isolation
  3. ***Regional*** optimizations
     - Apply to an “extended” basic block (e.g., a loop body)

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
  1. ***Peephole*** optimizations
     - Apply to a pair of adjacent instructions
  2. ***Local*** optimizations
     - Apply to a basic block in isolation
  3. ***Regional*** optimizations
     - Apply to an “extended” basic block (e.g., a loop body)
  4. ***Global*** optimizations (also “***intra-procedural***”)
     - Apply to a control-flow graph (method body) in isolation

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
    1. ***Peephole*** optimizations
        - Apply to a pair of adjacent instructions
    2. ***Local*** optimizations
        - Apply to a basic block in isolation
    3. ***Regional*** optimizations
        - Apply to an "extended" basic block (e.g., a loop body)
    4. ***Global*** optimizations (also "***intra-procedural***")
        - Apply to a control-flow graph (method body) in isolation
    5. ***Inter-procedural*** optimizations (also "***whole-program***")
        - Apply across method boundaries

Optimization: Taxonomy

- For languages like Cool there are 5 granularities of optimizations:
    1. ***Peephole*** optimizations
        - Apply to a pair of adjacent instructions
    2. ***Local*** optimizations
        - Apply to a basic block in isolation
    3. ***Regional*** optimizations
        - Apply to an "extended" basic block (e.g., a loop body)
    4. ***Global*** optimizations (also "***intra-procedural***")
        - Apply to a control-flow graph (method body) in isolation
    5. ***Inter-procedural*** optimizations (also "***whole-program***")
        - Apply across method boundaries
- Most compilers do (1) and (2), many do (3) and (4), and few do (5)

Optimization: Taxonomy

Today: some **peephole**
+ local optimizations

- For languages like Cool there are 5 g… …ons:
  1. *Peephole* optimizations
     - Apply to a pair of adjacent instructions
  2. *Local* optimizations
     - Apply to a basic block in isolation
  3. *Regional* optimizations
     - Apply to an “extended” basic block (e.g., a loop body)
  4. ***Global*** optimizations (also “***intra-procedural***”)
     - Apply to a control-flow graph (method body) in isolation
  5. *Inter-procedural* optimizations (also “*whole-program*”)
     - Apply across method boundaries
- Most compilers do (1) and (2), many do (3) and (4), and few do (5)

Peephole Optimizations

Peephole Optimizations

- After code generation, look at adjacent instructions (a "***peephole***" on the code stream)
    - try to replace adjacent instructions with something faster

Peephole Optimizations

- After code generation, look at adjacent instructions (a "***peephole***" on the code stream)
    - try to replace adjacent instructions with something faster
- Example:

```
movq %r9, 16(%rsp)
movq 16(%rsp), %r12
```

Peephole Optimizations

- After code generation, look at adjacent instructions (a "***peephole***" on the code stream)
    - try to replace adjacent instructions with something faster
- Example:

```
movq %r9, 16(%rsp)          movq %r9, 16(%rsp)
movq 16(%rsp), %r12   →     movq %r9, %r12
```

Peephole Optimizations

- After code generation, look at adjacent instructions (a “***peephole***” on the code stream)
  - try to replace adjacent instructions with something faster
- Example:

```
movq %r9, 16(%rsp)              movq %r9, 16(%rsp)
movq 16(%rsp), %r12     →       movq %r9, %r12
```

- Another example: “jump chains” (of unconditional jumps) can be replaced with a single jump

Peephole Optimizations: More Examples

```
subq $8, %rax
movq %r2, 0(%rax)
# %rax modified
# before next read
```

→

Peephole Optimizations: More Examples

```
subq $8, %rax
movq %r2, 0(%rax)
# %rax modified
# before next read
```

→

```
movq %r2, -8(%rax)
```

Peephole Optimizations: More Examples

```
subq $8, %rax
movq %r2, 0(%rax)
# %rax modified
# before next read
```

→

```
movq %r2, -8(%rax)
```

```
movq 16(%rsp), %rax
addq $1, %rax
movq %rax, 16(%rsp)
# %rax modified
# before next read
```

→

Peephole Optimizations: More Examples

```
subq $8, %rax
movq %r2, 0(%rax)
# %rax modified
# before next read
```

→

```
movq %r2, -8(%rax)
```

```
movq 16(%rsp), %rax
addq $1, %rax
movq %rax, 16(%rsp)
# %rax modified
# before next read
```

→

```
incq 16(%rsp)
```

Peephole Optimizations: More Examples

```
subq $8, %rax
movq %r2, 0(%rax)
# %rax modified
# before next read
```

→

```
movq %r2, -8(%rax)
```

```
movq 16(%rsp), %rax
addq $1, %rax
movq %rax, 16(%rsp)
# %rax modified
# before next read
```

→

```
incq 16(%rsp)
```

One way to do **complex instruction** selection

Trivia Break: Computer Science

This graph algorithm was independently invented by four different computer scientists in the 1950s; it is usually named after two or sometimes three of its inventors. Djikstra's shortest path algorithm can solve most instances of the problem that this algorithm addresses (and is faster). However, the advantage of this algorithm is that it can handle graphs with negative edge weights, including detecting "negative cycles" with infinite profitability (i.e., graphs in which no path is "shortest", because of the negative cycle).

Trivia Break: Medicine

This woman from Roanoke, Virginia died of cervical cancer in 1951 (aged just 31) at Johns Hopkins University in Baltimore. Without her consent, the medical researcher George Otto Gey cultured a line of cells from a tumor biopsy taken while she was being treated for cervical cancer. This cell line was the first "immortalized" human cell line that could be cloned indefinitely, and continues to be used in medical research to this day. Its cells are durable and prolific, and by 2009 more than 60,000 scientific articles had been published about research using the line. Name either the cell line or the woman whose cells were used to create it.

Local Optimizations

Local Optimizations

- A simple form of optimizations
  - No need to analyze the whole procedure body
  - Just the **basic block** in question

Local Optimizations

- A simple form of optimizations
  - No need to analyze the whole procedure body
  - Just the **basic block** in question
- Every (production) compiler does these
  - For many, there is no real danger of unprofitability and they're easy to prove safe

Local Optimizations

- A simple form of optimizations
  - No need to analyze the whole procedure body
  - Just the **basic block** in question
- Every (production) compiler does these
  - For many, there is no real danger of unprofitability and they're easy to prove safe
- Examples:
  - algebraic simplification
  - constant folding
  - local value numbering
  - …

Local Optimizations: Algebraic Simplification

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0            ->
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0          ->          x := 0
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0          ->          x := 0
y := y ** 2         ->
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0          ->          x := 0
y := y ** 2         ->          y := y * y
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0      ->      x := 0
y := y ** 2     ->      y := y * y
x := x * 8      ->
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0        ->        x := 0
y := y ** 2       ->        y := y * y
x := x * 8        ->        x := x << 3
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0      ->      x := 0
y := y ** 2     ->      y := y * y
x := x * 8      ->      x := x << 3
x := x * 15     ->
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0      ->      x := 0
y := y ** 2     ->      y := y * y
x := x * 8      ->      x := x << 3
x := x * 15     ->      t := x << 4;
                        x := t - x
```

Local Optimizations: Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
x := x * 0      ->      x := 0
y := y ** 2     ->      y := y * y
x := x * 8      ->      x := x << 3
x := x * 15     ->      t := x << 4;
                        x := t - x
```

- (watch out: on some machines `<<` is faster than `*`; but not on all!)

Local Optimizations: Constant Folding

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

```
x := y op z
```

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,
  - then `y op z` can be computed early

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,
  - then `y op z` can be computed early
- Example: `x := 2 + 2` ->

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,
  - then `y op z` can be computed early
- Example: `x := 2 + 2` -> `x:=4`

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,
  - then `y op z` can be computed early
- Example: `x := 2 + 2` -> `x:=4`
- Example: `if 2 < 0 jump L` can be deleted

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a statement

  `x := y op z`

  - ...and `y` and `z` are constants,
  - then `y op z` can be computed early
- Example: `x := 2 + 2` -> `x:=4`
- Example: `if 2 < 0 jump L` can be deleted
- When might constant folding be dangerous?

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a st

  `x := y op z`

  - ...and `y` and `z` are co
  - then `y op z` can be
- Example: `x := 2 + 2`
- Example: `if 2 < 0 jump L` can be deleted
- When might constant folding be dangerous?

Consider two machines X and Y with different FPUs:

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a st

  `x := y op z`

  - ...and `y` and `z` are co
  - then `y op z` can be
- Example: `x := 2 + 2`
- Example: `if 2 < 0 jump L` can be deleted
- When might constant folding be dangerous?

Consider two machines X and Y with different FPUs:

- `a := 1.5 + 3.7` =>
  `a := 5.2` on X

Local Optimizations: Constant Folding

- Operations on **constants** can be computed before the code executes
- In general, if there is a st
  `x := y op z`
  - ...and `y` and `z` are con
  - then `y op z` can be
- Example: `x := 2 + 2`
- Example: `if 2 < 0 jump L` can be deleted
- When might constant folding be dangerous?

Consider two machines X and Y with different FPUs:

- `a := 1.5 + 3.7` =>
  `a := 5.2` on X
- `a := 1.5 + 3.7` =>
  `a := 5.1999` on Y

Local Optimizations: Skipping Dead Code

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated
- Why would such basic blocks occur?

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or "fall through" from a conditional
  - Such basic blocks can be eliminated
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster

Local Optimizations: Skipping Dead Code

- Eliminating **unreachable code**:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)

Local Optimizations: SSA

- Most optimizations are simplified if each assignment is to a temporary that **has not appeared already** in the basic block

Local Optimizations: SSA

- Most optimizations are simplified if each assignment is to a temporary that **has not appeared already** in the basic block
- Intermediate code can be rewritten to be in **single static assignment** form (it's just another IR)

Local Optimizations: SSA

- Most optimizations are simplified if each assignment is to a temporary that **has not appeared already** in the basic block
- Intermediate code can be rewritten to be in **single static assignment** form (it's just another IR)
- Example (x_1 and a_1 are fresh temporaries):

```
x := a + y
a := x
x := a * x
b := x + a
```

→

Local Optimizations: SSA

- Most optimizations are simplified if each assignment is to a temporary that **has not appeared already** in the basic block
- Intermediate code can be rewritten to be in **single static assignment** form (it's just another IR)
- Example (x_1 and a_1 are fresh temporaries):

```
x  := a + y                x   := a + y
a  := x          →         a1  := x
x  := a * x                x1  := a1 * x
b  := x + a                b   := x1 + a1
```

Single Static Assignment (SSA)

- IR where each variable has **only one definition** in the program text

Single Static Assignment (SSA)

- IR where each variable has **only one definition** in the program text
    - A single **static** definition, but that definition can be in a loop, function, or other code that is executed dynamically many times

Single Static Assignment (SSA)

- IR where each variable has **only one definition** in the program text
    - A single **static** definition, but that definition can be in a loop, function, or other code that is executed dynamically many times
- Makes many analyses (and related optimizations) more efficient

Single Static Assignment (SSA)

- IR where each variable has **only one definition** in the program text
  - A single **static** definition, but that definition can be in a loop, function, or other code that is executed dynamically many times
- Makes many analyses (and related optimizations) more efficient
- Separates values from memory storage locations

Single Static Assignment (SSA)

- IR where each variable has **only one definition** in the program text
    - A single **static** definition, but that definition can be in a loop, function, or other code that is executed dynamically many times
- Makes many analyses (and related optimizations) more efficient
- Separates values from memory storage locations
- Complementary to CFG or data dependency graph
    - better for some things, but cannot do everything

SSA vs Functional Programming

- In **functional programming** variable values do not change
  - “referential transparency”

SSA vs Functional Programming

- In **functional programming** variable values do not change
    - “referential transparency”
- Instead you make a new variable with a similar name

SSA vs Functional Programming

- In **functional programming** variable values do not change
  - "referential transparency"
- Instead you make a new variable with a similar name
- Single assignment form is just like that!

SSA vs Functional Programming

- In **functional programming** variable values do not change
  - "referential transparency"
- Instead you make a new variable with a similar name
- Single assignment form is just like that!

$$
\begin{array}{l}
\texttt{x := a + y} \\
\texttt{a}_1\ \texttt{:= x} \\
\texttt{x}_1\ \texttt{:=a}_1\ \texttt{* x} \\
\texttt{b := x}_1\ \texttt{+ a}_1
\end{array}
\quad \cong \quad
\begin{array}{l}
\texttt{let x = a + y in} \\
\texttt{let a}_1\ \texttt{= x in} \\
\texttt{let x}_1\ \texttt{=a}_1\ \texttt{* x in} \\
\texttt{let b = x}_1\ \texttt{+ a}_1\ \texttt{in} \\
\ldots
\end{array}
$$

SSA: Basic Idea

- Basic Idea: for each original variable *v*, create a new variable v_n at the *n*th definition of the original *v*. Subsequent uses of *v* use v_n until the next definition point. E.g.:

SSA: Basic Idea

- Basic Idea: for each original variable *v*, create a new variable v_n at the *n*th definition of the original *v*. Subsequent uses of *v* use v_n until the next definition point. E.g.:

Original:

```
a := x + y
b := a – 1
a := y + b
b := x * 4
a := a + b
```

SSA: Basic Idea

- Basic Idea: for each original variable *v*, create a new variable v_n at the *n*th definition of the original *v*. Subsequent uses of *v* use v_n until the next definition point. E.g.:

Original:

a := x + y
b := a – 1
a := y + b
b := x * 4
a := a + b

SSA:

$a_1 := x_0 + y_0$
$b_1 := a_1 - 1$
$a_2 := y_0 + b_1$
$b_2 := x_0 * 4$
$a_3 := a_2 + b_2$

SSA: Merges

- This is fine until we reach a merge point:

```
if (...)
   a = x;
else
   a = y;
b = a;
```

SSA: Merges

- This is fine until we reach a merge point:

```
if (...)
   a = x;
else
   a = y;
b = a;
```

$$
\begin{array}{l}
\text{if } (\ldots) \\
\quad a_1 = x_0; \\
\text{else} \\
\quad a_2 = y_0; \\
b_1 = a_{???};
\end{array}
$$

SSA: Merges

- This is fine until we reach a merge point:

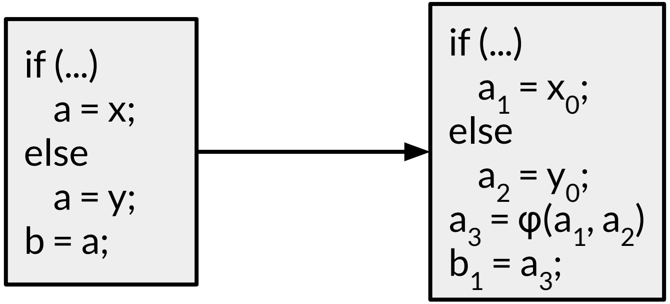
```
if (...)
   a = x;
else
   a = y;
b = a;
```

if (...)
$\quad a_1 = x_0;$
else
$\quad a_2 = y_0;$
$a_3 = \varphi(a_1, a_2)$
$b_1 = a_3;$

- Solution: introduce a ***φ-function*** (“phi function”)

SSA: Merges

- This is fine until we reach a merge point:



- Solution: introduce a ***φ-function*** (“phi function”)
  - semantics: a_3 is assigned to either a_1 or a_2, depending on which control flow path us used to reach the φ-function

How does the φ-function “know” what to pick?

How does the φ-function “know” what to pick?

- It doesn’t!

How does the φ-function “know” what to pick?

- It doesn’t!
- φ-functions **don’t actually exist** at run time

How does the φ-function “know” what to pick?

- It doesn’t!
- φ-functions **don’t actually exist** at run time
  - when we’re done using the SSA IR, we translate back out of SSA form, removing all φ-functions

How does the φ-function "know" what to pick?

- It doesn't!
- φ-functions **don't actually exist** at run time
  - when we're done using the SSA IR, we translate back out of SSA form, removing all φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA variable x

How does the φ-function “know” what to pick?

- It doesn’t!
- φ-functions **don’t actually exist** at run time
  - when we’re done using the SSA IR, we translate back out of SSA form, removing all φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA variable x
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to “execute” anything

How does the φ-function "know" what to pick?

- It doesn't!
- φ-functions **don't actually exist** at run time
  - when we're done using the SSA IR, we translate back out of SSA form, removing all φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA variable x
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to "execute" anything
    - So φ-functions are (only) compile-time bookkeeping

Converting to SSA

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
  - called ***maximal SSA***

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
    - called ***maximal SSA***
    - this wastes way too much space and time to be useful in practice

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
  - called ***maximal SSA***
  - this wastes way too much space and time to be useful in practice
- Instead, use the ***path convergence criterion***: insert a φ-function for variable *a* at program point *z* when:

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
    - called ***maximal SSA***
    - this wastes way too much space and time to be useful in practice
- Instead, use the ***path convergence criterion***: insert a φ-function for variable *a* at program point *z* when:
    - There are blocks x and y, both containing definitions of a, and x != y

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
  - called ***maximal SSA***
  - this wastes way too much space and time to be useful in practice
- Instead, use the ***path convergence criterion***: insert a φ-function for variable *a* at program point *z* when:
  - There are blocks x and y, both containing definitions of a, and x != y
  - There are non-empty paths from x to z and from y to z

Converting to SSA

- Could simply add φ-functions for every variable at every join point(!)
  - called ***maximal SSA***
  - this wastes way too much space and time to be useful in practice
- Instead, use the ***path convergence criterion***: insert a φ-function for variable *a* at program point *z* when:
  - There are blocks x and y, both containing definitions of a, and x != y
  - There are non-empty paths from x to z and from y to z
  - These paths have no common nodes other than z

Common Subexpression Elimination

Common Subexpression Elimination

- Assume:
  - Basic block is in single assignment form

Common Subexpression Elimination

- Assume:
  - Basic block is in single assignment form
- Then all assignments with same right-hand side compute the same value (why?)

Common Subexpression Elimination

- Assume:
  - Basic block is in single assignment form
- Then all assignments with same right-hand side compute the same value (why?)
- Example:

```
x := y + z
…
w := y + z
```

→

Common Subexpression Elimination

- Assume:
    - Basic block is in single assignment form
- Then all assignments with same right-hand side compute the same value (why?)
- Example:

```
x := y + z                 x := y + z
…              →           …
w := y + z                 w := x
```

Common Subexpression Elimination

- Assume:
  - Basic block is in single assignment form
- Then all assignments with same right-hand side compute the same value (why?)
- Example:

```
x := y + z                    x := y + z
…                →            …
w := y + z                    w := x
```

- Why is SSA form important here?

Copy Propagation

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`
- Example:

```
b := z + y
a := b
x := 2 * a
```

$\longrightarrow$

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`
- Example:

```
b := z + y
a := b
x := 2 * a
```

→

```
b := z + y
a := b
x := 2 * b
```

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`
- Example:

```
b := z + y
a := b
x := 2 * a
```

$\longrightarrow$

```
b := z + y
a := b
x := 2 * b
```

- This does not make the program smaller or faster

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`
- Example:

```
b := z + y
a := b
x := 2 * a
```

→

```
b := z + y
a := b
x := 2 * b
```

- This does not make the program smaller or faster
  - But it might enable other optimizations!
    - Constant folding, dead code elimination

Copy Propagation

- If `w := x` appears in a block, then all subsequent uses of `w` can be replaced by `x`
- Example:

```
b := z + y                b := z + y
a := b         →          a := b
x := 2 * a                x := 2 * b
```

- This does not make the program smaller or faster
  - But it might enable other optimizations!
    - Constant folding, dead code elimination
- Again, being in SSA is critical to the proof that this is safe

Example: Copy Propagation + Constant Folding

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

$\longrightarrow$

Example: Copy Propagation + Constant Folding

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

$\longrightarrow$

```
a := 5
x := 10
y := 16
t := x << 4
```

Dead Code Elimination (DCE)

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that a is not used anywhere else)

```
x := z + y
a := x
x := 2 * a
```

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that `a` is not used anywhere else)

```
x := z + y            b := z + y
a := x       →        a := b       →
x := 2 * a            x := 2 * b
```

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that `a` is not used anywhere else)

```
x := z + y          b := z + y          b := z + y
a := x       →      a := b       →      x := 2 * b
x := 2 * a          x := 2 * b
```

Applying Local Optimizations

Applying Local Optimizations

- Each local optimization does very little by itself

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
  - Performing one optimization enables (or disables!) other optimizations

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
  - Performing one optimization enables (or disables!) other optimizations
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - **Phase ordering problem** again: must beware of local minima

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
  - Performing one optimization enables (or disables!) other optimizations
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - **Phase ordering problem** again: must beware of local minima
- Interpreters and JITs must be fast!
  - The optimizer can also be stopped at any time to limit the compilation time

An Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic simplification:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic simplification:

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Dead code elimination:

```
a := x * x
~~b := 3~~
~~c := x~~
~~d := a~~
~~e := 6~~
f := a + d
g := e * f
```

An Example

- Dead code elimination:

```
a := x * x
~~b := 3~~
~~c := x~~
~~d := a~~
~~e := 6~~
f := a + d
g := e * f
```

Could we get to `g = 12 * a`?

Local Optimization Notes

Local Optimization Notes

- Intermediate code is helpful for many optimizations

Local Optimization Notes

- Intermediate code is helpful for many optimizations
  - Basic Blocks: known entry and exit
  - Single Static Assignment form: one definition per variable

Local Optimization Notes

- Intermediate code is helpful for many optimizations
    - Basic Blocks: known entry and exit
    - Single Static Assignment form: one definition per variable
- “Program optimization” is grossly misnamed

Local Optimization Notes

- Intermediate code is helpful for many optimizations
  - Basic Blocks: known entry and exit
  - Single Static Assignment form: one definition per variable
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense

Local Optimization Notes

- Intermediate code is helpful for many optimizations
  - Basic Blocks: known entry and exit
  - Single Static Assignment form: one definition per variable
- "Program optimization" is grossly misnamed
  - Code produced by "optimizers" is not optimal in any reasonable sense
  - "Program improvement" is a more appropriate term

Course Announcements

- Graded **midterms** are at the front of the room
    - If you don't have it yet, pick it up now
    - If you take it with you, I won't accept regrade requests
- A problem with the PA3c3 autograder was found over the weekend
    - I've therefore granted an extension to **Wednesday** (AoE)
        - Same extension for PA3
- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.

Example: Local Value Numbering

-