Intermediate Representations (IRs)

Martin Kellogg

Agenda

- quiz
- what is an IR?
  - a taxonomy and toolbox
- three-address code (“TAC”, relevant to PA3c2)
- single static assignment form (“SSA”, maybe relevant to other PAs…)

Agenda

- **quiz**
- what is an IR?
  - a taxonomy and toolbox
- three-address code (“TAC”, relevant to PA3c2)
- single static assignment form (“SSA”, maybe relevant to other PAs…)

Agenda

- quiz
- **what is an IR?**
  - a taxonomy and toolbox
- three-address code (“TAC”, relevant to PA3c2)
- single static assignment form (“SSA”, maybe relevant to other PAs…)

What's an Intermediate Representation

Definition: an ***intermediate representation*** (***IR***) is any internal data structure that a compiler uses for the facts that it derives about the program.

What's an Intermediate Representation

Definition: an ***intermediate representation*** (***IR***) is any internal data structure that a compiler uses for the facts that it derives about the program.

- this definition is intentionally broad, and it includes data structures we've already talked about, including ASTs, CFGs, the implementation/parent/class maps from PA2, etc.

What's an Intermediate Representation

Definition: an ***intermediate representation*** (***IR***) is any internal data structure that a compiler uses for the facts that it derives about the program.

- this definition is intentionally broad, and it includes data structures we've already talked about, including ASTs, CFGs, the implementation/parent/class maps from PA2, etc.
- the IR is generally **the canonical form** that the compiler reasons about (i.e., the compiler discards the source code)

What's an Intermediate Representation

Definition: an ***intermediate representation*** (***IR***) is any internal data structure that a compiler uses for the facts that it derives about the program.

- this definition is intentionally broad, and it includes data structures we've already talked about, including ASTs, CFGs, the implementation/parent/class maps from PA2, etc.
- the IR is generally **the canonical form** that the compiler reasons about (i.e., the compiler discards the source code)
  - this makes IR choice important - facts that are not in the IR aren't available to the compiler!

What's an Intermediate Representation

Definition: an ***intermediate representation*** (***IR***) is any internal data structure that a compiler uses for the facts that it derives about the program.

- this definition is intentionally broad, and it includes data structures we've already talked about, including ASTs, CFGs, the implementation/parent/class maps from PA2, etc.
- the IR is generally **the canonical form** that the compiler reasons about (i.e., the compiler discards the source code)
  - this makes IR choice important - facts that are not in the IR aren't available to the compiler!
- typically, a compiler uses **different IRs** for different tasks

How many IRs does a compiler need?

How many IRs does a compiler need?

- As many as are useful!
  - I know that’s not a very helpful answer…

How many IRs does a compiler need?

- As many as are useful!
  - I know that's not a very helpful answer…
- Typically, a compiler will process IRs in **stages**
  - i.e., it will transform one IR into another, into another, etc., before eventually emitting code

How many IRs does a compiler need?

- As many as are useful!
  - I know that's not a very helpful answer…
- Typically, a compiler will process IRs in **stages**
  - i.e., it will transform one IR into another, into another, etc., before eventually emitting code
- Different IRs are **better suited to different tasks**

How many IRs does a compiler need?

- As many as are useful!
  - I know that's not a very helpful answer…
- Typically, a compiler will process IRs in **stages**
  - i.e., it will transform one IR into another, into another, etc., before eventually emitting code
- Different IRs are **better suited to different tasks**
  - for example, a CFG is useful for abstract interpretation or dataflow analysis…

How many IRs does a compiler need?

- As many as are useful!
  - I know that's not a very helpful answer…
- Typically, a compiler will process IRs in **stages**
  - i.e., it will transform one IR into another, into another, etc., before eventually emitting code
- Different IRs are **better suited to different tasks**
  - for example, a CFG is useful for abstract interpretation or dataflow analysis…
  - …but would you use it for peephole optimizations? No, you want something much closer to the target assembly

How many IRs does a compiler need?

- As many as are useful!
  - I know that's not a very helpfu
- Typically, a compiler will process I
  - i.e., it will transform one IR int
    before eventually emitting code
- Different IRs are **better suited to different tasks**
  - for example, a CFG is useful for abstract interpretation or dataflow analysis…
  - …but would you use it for peephole optimizations? No, you want something much closer to the target assembly

An example of a ***peephole optimization*** is removing a redundant store/load pair that access the same address

IR Design

- The design space of IRs is **very large**

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones
  - You are welcome to choose to use any or all of these in your compiler (or invent your own)

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones
  - You are welcome to choose to use any or all of these in your compiler (or invent your own)
- IR **design decisions** impact the whole compiler

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones
  - You are welcome to choose to use any or all of these in your compiler (or invent your own)
- IR **design decisions** impact the whole compiler
  - desirable properties:

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones
  - You are welcome to choose to use any or all of these in your compiler (or invent your own)
- IR **design decisions** impact the whole compiler
  - desirable properties:
    - easy to generate
    - easy to manipulate
    - expressive
    - appropriate level of abstraction

IR Design

- The design space of IRs is **very large**
  - In this lecture, I will survey some common ones
  - You are welcome to choose to use any or all of these in your compiler (or invent your own)
- IR **design decisions** impact the whole compiler
  - desirable properties:
    - easy to generate
    - easy to manipulate
    - expressive
    - appropriate level of abstraction
  - tradeoffs between these motivate different IRs at different compilation stages

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)
    - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)
    - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)
  - ***Abstraction Level***

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)
    - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)
  - ***Abstraction Level***
    - High-level, near to source language

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)
    - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)
  - ***Abstraction Level***
    - High-level, near to source language
    - Low-level, closer to machine (exposes more details to compiler)

IR Design Taxonomy

- A non-exhaustive list of ways that IRs vary:
  - ***Structure***
    - Graphical (trees, graphs, etc.)
    - Linear (code for some **abstract machine**)
    - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)
  - ***Abstraction Level***
    - High-level, near to source language
    - Low-level, closer to machine (exposes more details to compiler)
  - ***Naming conventions***

IR Design Taxonomy

- A non-exhaustive list of w
  - ***Structure***
    - Graphical (trees, g
    - Linear (code for s
    - Hybrids are comm
      nodes are basic blocks of linear code)
  - ***Abstraction Level***
    - High-level, near to source language
    - Low-level, closer to machine (exposes more details to compiler)
  - ***Naming conventions***

Think of our discussion today as a **toolbox** of various well-known IRs. When building your compiler, pick and choose which ones are useful for each task

Example: Representing an Array Reference

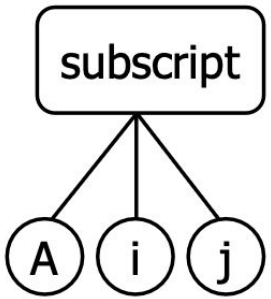
Example: Representing an Array Reference

source code: `A[i,j]`

Example: Representing an Array Reference

source code: `A[i,j]`

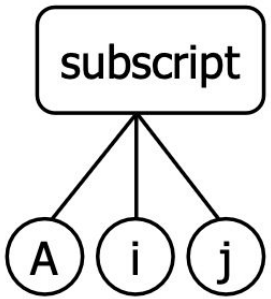
graphical IR (tree):



Example: Representing an Array Reference

source code: `A[i,j]`

graphical IR (tree):

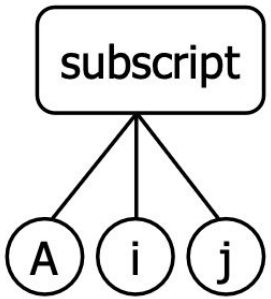


high-level linear IR: t_1 `<- A[i,j]`

Example: Representing an Array Reference

source code: `A[i,j]`

graphical IR (tree):



low-level linear IR:

```
loadI 1 => r1
sub rj,r1 => r2
loadI 10 => r3
mult r2,r3 => r4
sub ri,r1 => r5
add r4,r5 => r6
loadI @A => r7
add r7,r6 => r8
load r8 => r9
```

high-level linear IR: t_1 `<- A[i,j]`

Levels of Abstraction

- Key design decision: how much **detail** to expose

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:
    - Some semantic analysis & optimizations are easier with high-level IRs close to the source code (e.g., an AST)

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:
    - Some semantic analysis & optimizations are easier with high-level IRs close to the source code (e.g., an AST)
    - Low-level usually preferred for other optimizations, register allocation, code generation, etc.

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:
    - Some semantic analysis & optimizations are easier with high-level IRs close to the source code (e.g., an AST)
    - Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level – but are also used for low-level

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:
    - Some semantic analysis & optimizations are easier with high-level IRs close to the source code (e.g., an AST)
    - Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level – but are also used for low-level
  - Linear IRs are typically low-level

Levels of Abstraction

- Key design decision: how much **detail** to expose
  - Affects possibility and profitability of various optimizations
  - Depends on compiler phase:
    - Some semantic analysis & optimizations are easier with high-level IRs close to the source code (e.g., an AST)
    - Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level – but are also used for low-level
  - Linear IRs are typically low-level
  - But these generalizations don't always hold

Levels of Abstraction

- Key design decision: how much detail to expose
  - Affects possibility an
  - Depends on compile
    - Some semantic a
      high-level IRs clo
    - Low-level usually
      register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level – but are also used for low-level
  - Linear IRs are typically low-level
  - But these generalizations don't always hold

One view: **source code** is *just another IR* that we happen to expose to programmers

- high-level of abstraction, linear

Graphical IRs

Graphical IRs

- A graphical IR is any IR that is represented as a graph (or tree, flowchart, or other “graphical” structure)

Graphical IRs

- A graphical IR is any IR that is represented as a graph (or tree, flowchart, or other “graphical” structure)
- **Nodes and edges** typically reflect **structure** of the program
  - E.g., control flow, data dependence, caller/callee

Graphical IRs

- A graphical IR is any IR that is represented as a graph (or tree, flowchart, or other “graphical” structure)
- **Nodes and edges** typically reflect **structure** of the program
  - E.g., control flow, data dependence, caller/callee
- May be large (especially syntax trees)

Graphical IRs

- A graphical IR is any IR that is represented as a graph (or tree, flowchart, or other “graphical” structure)
- **Nodes and edges** typically reflect **structure** of the program
  - E.g., control flow, data dependence, caller/callee
- May be large (especially syntax trees)
- High-level examples: syntax trees, ***directed-acyclic graphs*** (***DAGs***)
  - Common in early phases of compilers

Graphical IRs

- A graphical IR is any IR that is represented as a graph (or tree, flowchart, or other “graphical” structure)
- **Nodes and edges** typically reflect **structure** of the program
  - E.g., control flow, data dependence, caller/callee
- May be large (especially syntax trees)
- High-level examples: syntax trees, ***directed-acyclic graphs*** (***DAGs***)
  - Common in early phases of compilers
- Other examples: control flow graphs, ***data dependency graphs***, and ***call graphs***
  - Often used in semantic analysis, optimization, and code generation

DAGs?

DAGs?

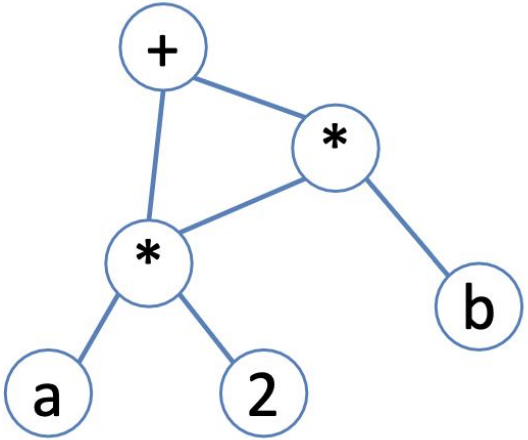
- Can use a directed acyclic graph to store a variation on the AST

DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures

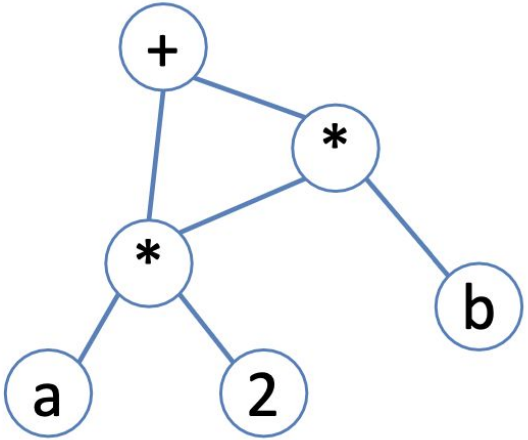
DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures
- Example: (a * 2) + ((a * 2) * b)



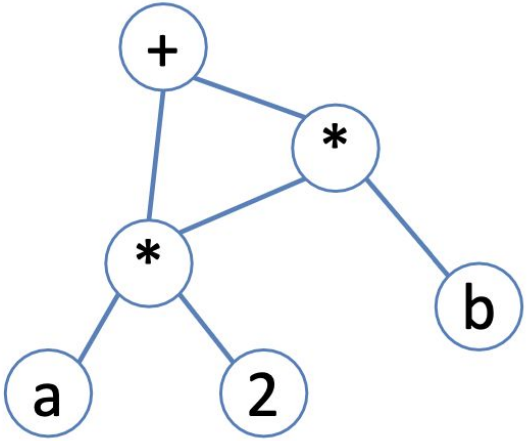
DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures
- Example: (a * 2) + ((a * 2) * b)
- Pros: saves space, exposes **redundant sub-expressions**



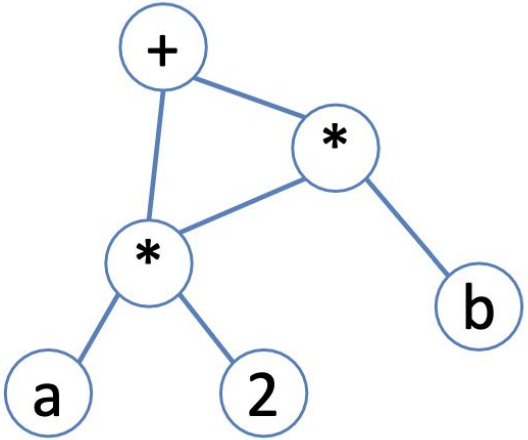
DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures
- Example: (a * 2) + ((a * 2) * b)
- Pros: saves space, exposes **redundant sub-expressions**
  - why might it be useful to expose redundant sub-expressions?



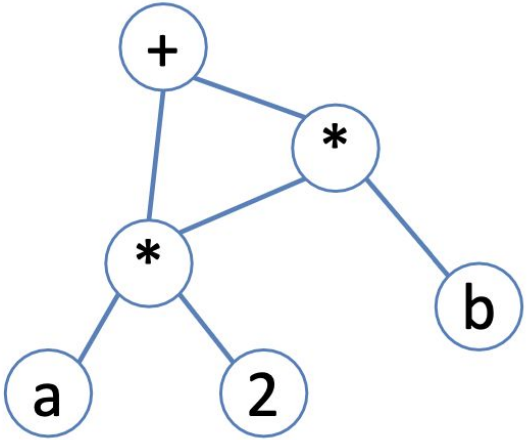
DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures
- Example: (a * 2) + ((a * 2) * b)
- Pros: saves space, exposes **redundant sub-expressions**
  - why might it be useful to expose redundant sub-expressions?
- Cons: less flexibility if part of tree should be changed



DAGs?

- Can use a directed acyclic graph to store a variation on the AST
  - to capture shared substructures
- Example: (a * 2) + ((a * 2) * b)
- Pros: saves space, exposes **redundant sub-expressions**
  - why might it be useful to expose redundant sub-expressions?
- Cons: less flexibility if part of tree should be changed
- Ask me about egraphs in OH



Data Dependency Graphs

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:
  - Block A defines x then B reads it (***RAW*** – ***read after write***)

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:
  - Block A defines x then B reads it (***RAW*** – ***read after write***)
  - Block A reads x then B writes it (***WAR*** – “anti- dependence”)

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:
  - Block A defines x then B reads it (***RAW*** – ***read after write***)
  - Block A reads x then B writes it (***WAR*** – “anti- dependence”)
  - Blocks A and B both write x (***WAW***)
    - order of blocks must reflect original program semantics

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:
  - Block A defines x then B reads it (***RAW*** – ***read after write***)
  - Block A reads x then B writes it (***WAR*** – “anti- dependence”)
  - Blocks A and B both write x (***WAW***)
    - order of blocks must reflect original program semantics
- These dependencies restrict what **reorderings** the compiler can do

Data Dependency Graphs

- In a ***data dependency graph***, an edge between a pair of nodes indicates that they reference common data
- Examples:
  - Block A defines x then B reads it (***RAW*** – ***read after write***)
  - Block A reads x then B writes it (***WAR*** – “anti- dependence”)
  - Blocks A and B both write x (***WAW***)
    - order of blocks must reflect original program semantics
- These dependencies restrict what **reorderings** the compiler can do
- Data dependency graph is most often used in conjunction with another IR to facilitate optimizations, but it has other uses too…

Call Graphs

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures
  - one node for each procedure and one edge for each distinct procedure call site

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures
  - one node for each procedure and one edge for each distinct procedure call site
  - e.g., if the code calls *q* from three textually distinct sites in *p*; the call graph has three edges (*p*, *q*), one for each call site

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures
  - one node for each procedure and one edge for each distinct procedure call site
  - e.g., if the code calls *q* from three textually distinct sites in *p*; the call graph has three edges (*p*, *q*), one for each call site
- Call graphs are useful if you want to do ***inter-procedural*** analysis

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures
  - one node for each procedure and one edge for each distinct procedure call site
  - e.g., if the code calls *q* from three textually distinct sites in *p*; the call graph has three edges (*p*, *q*), one for each call site
- Call graphs are useful if you want to do ***inter-procedural*** analysis
  - that is, analysis that requires you to reason about more than one procedure at the same time

Call Graphs

- A ***call graph*** represents the runtime transfers of control between procedures
  - one node for each procedure and one edge for each distinct procedure call site
  - e.g., if the code calls *q* from three textually distinct sites in *p*; the call graph has three edges (*p*, *q*), one for each call site
- Call graphs are useful if you want to do ***inter-procedural*** analysis
  - that is, analysis that requires you to reason about more than one procedure at the same time
  - during codegen, you *probably* don't need to do this, but it might be useful for optimization

Trivia Break: International Relations

This ancient Athenian historian and general chronicled the fifth-century BC war between Athens and Sparta in his *History of the Peloponnesian War*. He is considered by some to be the first modern political theorist, because of his claims to have applied strict standards of impartiality and evidence-gathering and analysis of cause and effect, without reference to intervention by the gods. The Melian dialogue from his work is regarded as a seminal text of international relations theory.

Linear IRs

Linear IRs

- Pseudo-code for some **abstract machine**

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples:

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples:
  - ***three-address code*** (“***TAC***”), which we’ll see some examples of in a few minutes (and which you must generate for PA3c2

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples:
  - ***three-address code*** (“***TAC***”), which we’ll see some examples of in a few minutes (and which you must generate for PA3c2
  - ***stack machine code***, which we’ll see a long example of in a later lecture

Linear IRs

- Pseudo-code for some **abstract machine**
  - “Linear” because, like source code, it has a textual structure
  - Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples:
  - ***three-address code*** (“***TAC***”), which we’ll see some examples of in a few minutes (and which you must generate for PA3c2
  - ***stack machine code***, which we’ll see a long example of in a later lecture
  - ***single static assignment form***, which we’ll see briefly today too

Linear IRs: abstraction level

- Linear IRs can be close to the source language, very low-level, or somewhere in between.

Linear IRs: abstraction level

- Linear IRs can be close to the source language, very low-level, or somewhere in between.
- Consider, for example, linear IRs for C array reference

```
a[i][j+2]
```

Linear IRs: abstraction level

- Linear IRs can be close to the source language, very low-level, or somewhere in between.
- Consider, for example, linear IRs for C array reference

```
a[i][j+2]
```

- A high-level linear IR might represent this very similarly to source code:

```
t1 <- a[i,j+2]
```

Linear IRs: abstraction level

- A “medium level” IR:

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

Linear IRs: abstraction level

- A “medium level” IR:

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

still retains basic symbolic info about variables

Linear IRs: abstraction level

- A “medium level” IR:

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

still retains basic symbolic info about variables

- A “low level” IR:

```
r1 <- [fp-4]
r2 <- r1 + 2
r3 <- [fp-8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp – 216
f1 <- [r7+r6]
```

Linear IRs: abstraction level

- A “medium level” IR:

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

still retains basic symbolic info about variables

- A “low level” IR:

```
r1 <- [fp-4]
r2 <- r1 + 2
r3 <- [fp-8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp – 216
f1 <- [r7+r6]
```

exposes all details of the low-level layout; explicit memory references and calculations

Linear IRs: abstraction level tradeoffs

Linear IRs: abstraction level tradeoffs

- **High-level**: good for some high-level optimizations, semantic checking; but can't optimize things that are hidden – like address arithmetic for array subscripting

Linear IRs: abstraction level tradeoffs

- **High-level**: good for some high-level optimizations, semantic checking; but can't optimize things that are hidden – like address arithmetic for array subscripting
- **Low-level**: need for good code generation and resource utilization in back end but loses some semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)

Linear IRs: abstraction level tradeoffs

- **High-level**: good for some high-level optimizations, semantic checking; but can't optimize things that are hidden – like address arithmetic for array subscripting
- **Low-level**: need for good code generation and resource utilization in back end but loses some semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)
- **Medium-level**: more detail but keeps more higher-level semantic information – great for machine-independent optimizations. Many (all?) optimizing compilers work at this level

Linear IRs: abstraction level tradeoffs

- **High-level**: good for some high-level optimizations, semantic checking; but can't optimize things that are hidden – like address arithmetic for array subscripting
- **Low-level**: need for good code generation and resource utilization in back end but loses some semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)
- **Medium-level**: more detail but keeps more higher-level semantic information – great for machine-independent optimizations. Many (all?) optimizing compilers work at this level
- Many compilers **use all 3** in different phases

Linear IRs: stack machines

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)
  - **Easy to generate**; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)
  - **Easy to generate**; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT
  - **Simple to interpret** or compile to machine code

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)
  - **Easy to generate**; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT
  - **Simple to interpret** or compile to machine code
- Disadvantages:

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)
  - **Easy to generate**; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT
  - **Simple to interpret** or compile to machine code
- Disadvantages:
  - Somewhat **inconvenient/difficult to optimize** directly

Linear IRs: stack machines

- Originally used for stack-based computers
  - famous example: B5000, ~1961
- Often used for virtual machines
  - Classic examples: Pascal's pcode and Java bytecode
- Advantages:
  - **Compact**; mostly 0-address opcodes (great if sent via network)
  - **Easy to generate**; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT
  - **Simple to interpret** or compile to machine code
- Disadvantages:
  - Somewhat **inconvenient/difficult to optimize** directly
  - Does not match up with modern chip architectures

Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

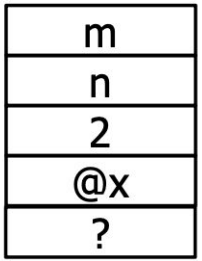
```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```

Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

push all operands onto stack

```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```

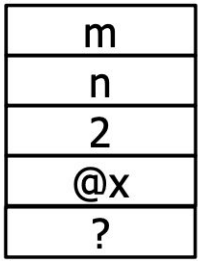


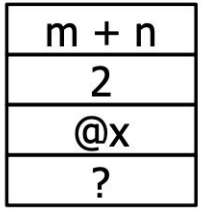
Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

do the add on top two operands

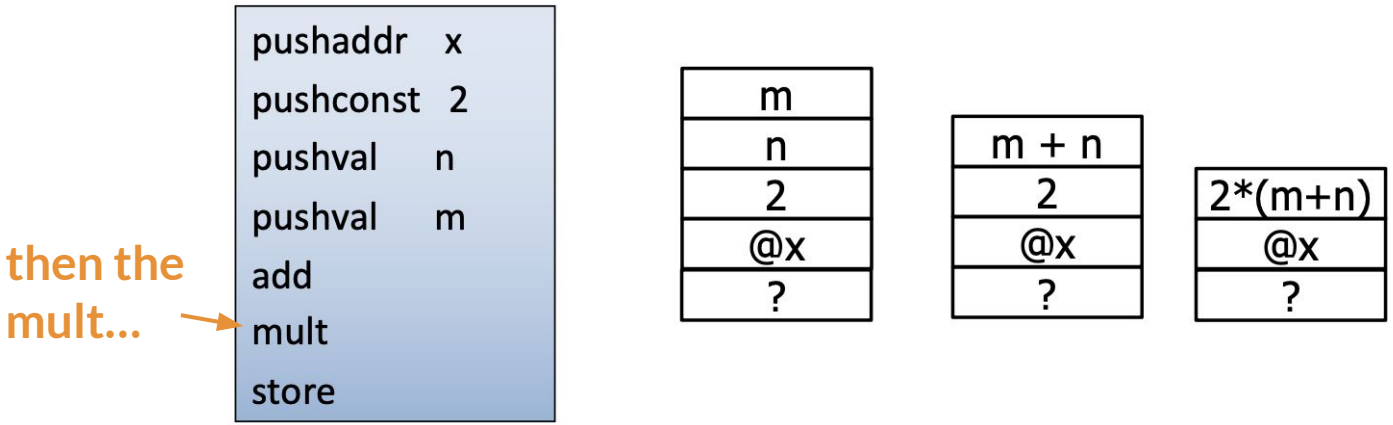
```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```





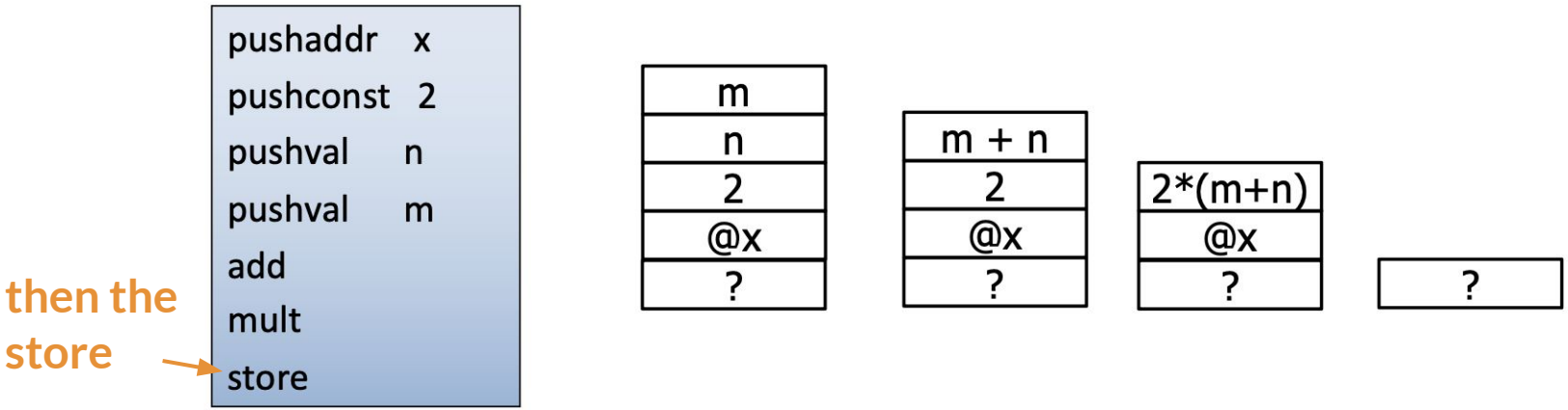
Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:



Linear IRs: stack machines: example

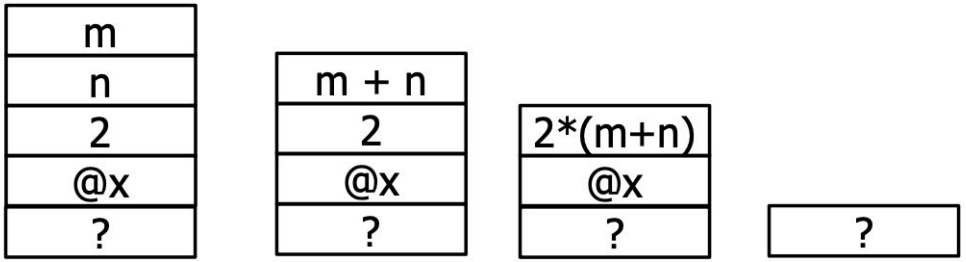
- Hypothetical code for `x = 2 * (m + n)`:



Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```

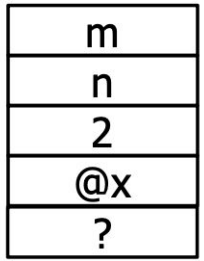


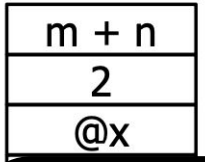
- Note compactness:
  - common opcodes just 1 byte wide
  - instructions have 0 or 1 operand

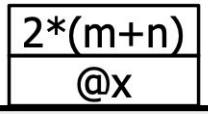
Linear IRs: stack machines: example

- Hypothetical code for `x = 2 * (m + n)`:

```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```







In our “code generation” lectures next week we will talk explicitly about how you’d build a **stack machine for Cool**

- Note compactness:
  - common opcodes just 1 byte
  - instructions have 0 or 1 operand

“Hybrid” IRs

“Hybrid” IRs

- Combinations of linear and graphical IRs are common

“Hybrid” IRs

- Combinations of linear and graphical IRs are common
  - for example, a CFG’s basic blocks usually contain code in some linear IR (e.g., TAC)

“Hybrid” IRs

- Combinations of linear and graphical IRs are common
  - for example, a CFG’s basic blocks usually contain code in some linear IR (e.g., TAC)
  - we call these ***hybrid IRs***

“Hybrid” IRs

- Combinations of linear and graphical IRs are common
  - for example, a CFG’s basic blocks usually contain code in some linear IR (e.g., TAC)
  - we call these ***hybrid IRs***
- Level of abstraction varies; you can mix and match based on your needs

“Hybrid” IRs

- Combinations of linear and graphical IRs are common
  - for example, a CFG’s basic blocks usually contain code in some linear IR (e.g., TAC)
  - we call these ***hybrid IRs***
- Level of abstraction varies; you can mix and match based on your needs
  - when designing your own compiler’s internals, it’s okay to **be creative**: pick the representation that makes your life easiest

Three-Address Code

Three-Address Code

- Usual form: `x <- y` *op* `z`

Three-Address Code

- Usual form: `x <- y op z`
  - One operator

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)
- Eg: `x = 2 * (m + n)` becomes

```
t1 <- m + n; t2 <- 2 * t1; x  <- t2
```

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)
- Eg: `x = 2 * (m + n)` becomes

  `t1 <- m + n; t2 <- 2 * t1; x  <- t2`
  - You may prefer:

    `add t1, m, n; mul t2, 2, t1; movx, t2`

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)
- Eg: `x = 2 * (m + n)` becomes

  `t1 <- m + n; t2 <- 2 * t1; x  <- t2`
  - You may prefer:

    `add t1, m, n; mul t2, 2, t1; movx, t2`
- Invent as **many new temp names** as needed

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)
- Eg: `x = 2 * (m + n)` becomes

  ```
  t1 <- m + n; t2 <- 2 * t1; x  <- t2
  ```

  - You may prefer:

    ```
    add t1, m, n; mul t2, 2, t1; movx, t2
    ```
- Invent as **many new temp names** as needed
  - “expression temps” don’t correspond to any user variables; de-anonymize expressions

Three-Address Code

- Usual form: `x <- y op z`
  - One operator
  - **Maximum of 3 names** (thus the name)
- Eg: `x = 2 * (m + n)` becomes
  `t1 <- m + n; t2 <- 2 * t1; x <- t2`
  - You may prefer:
    `add t1, m, n; mul t2, 2, t1; movx, t2`
- Invent as **many new temp names** as needed
  - "expression temps" don't correspond to any user variables; de-anonymize expressions
- Store in a quadruple: `<lhs,rhs1,op,rhs2>`

Why TAC?

Why TAC?

- Advantages:

Why TAC?

- Advantages:
  - Resembles code for actual machines

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Why does PA3c2 require TAC?

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Why does PA3c2 require TAC?
  - I want you to build *an* IR early. You'll need several before you finish PA3.

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Why does PA3c2 require TAC?
  - I want you to build *an* IR early. You'll need several before you finish PA3.
  - I think TAC is **basically guaranteed to be useful** no matter how you design the rest of your compiler
    - and I had to pick something…

Why TAC?

- Advantages:
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Why does PA3c2 require TAC?
  - I want you to build *an* IR early. You'll need several before you finish PA3.
  - I think TAC is **basically**
    how you design the re
    - and I had to pick something…

The PA3c2 page has a long discussion on Cool's TAC format, including pseudocode to generate it.

Single Static Assignment (SSA)

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: 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:

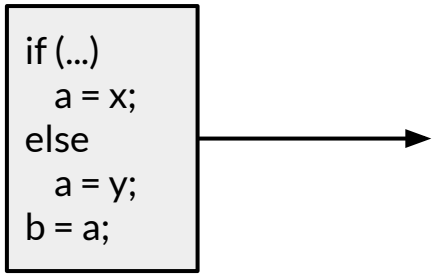
a := x + y
b := a – 1
a := y + b
b := x * 4
a := a + b

SSA:

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

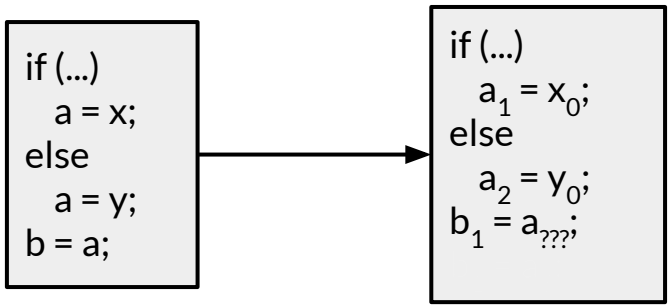
SSA: Merges

- This is fine until we reach a merge point:



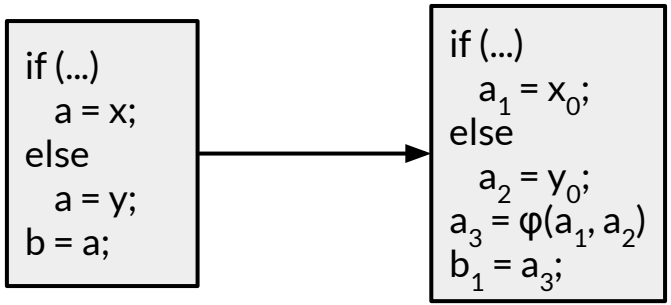
SSA: Merges

- This is fine until we reach a merge point:



SSA: Merges

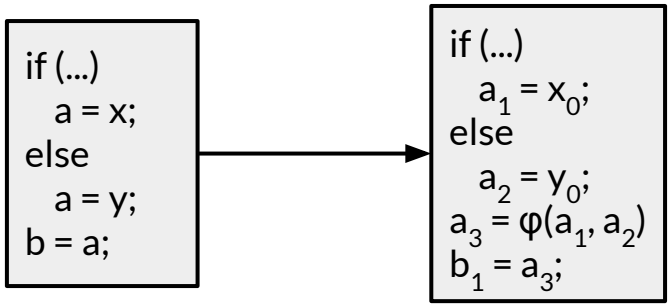
- This is fine until we reach a merge point:



- 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

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 p
  - There are blocks x and
    x != y
  - There are non-empty p
  - These paths have no common nodes other than z

We'll come back to SSA form when it's relevant for optimizations we're talking about, and revisit how you actually do this

Course Announcements

- PA2 due today!
- PA3c1 (codegen testing) is due on Friday
  - all gas, no brakes
- My OH on Wednesday will be later than usual (4-5 instead of 3:30-4:30), because of a CS faculty meeting until 4
  - might even start a little bit later…