Run-time Organization and Operational Semantics

Martin Kellogg

Review: Run-time Organization

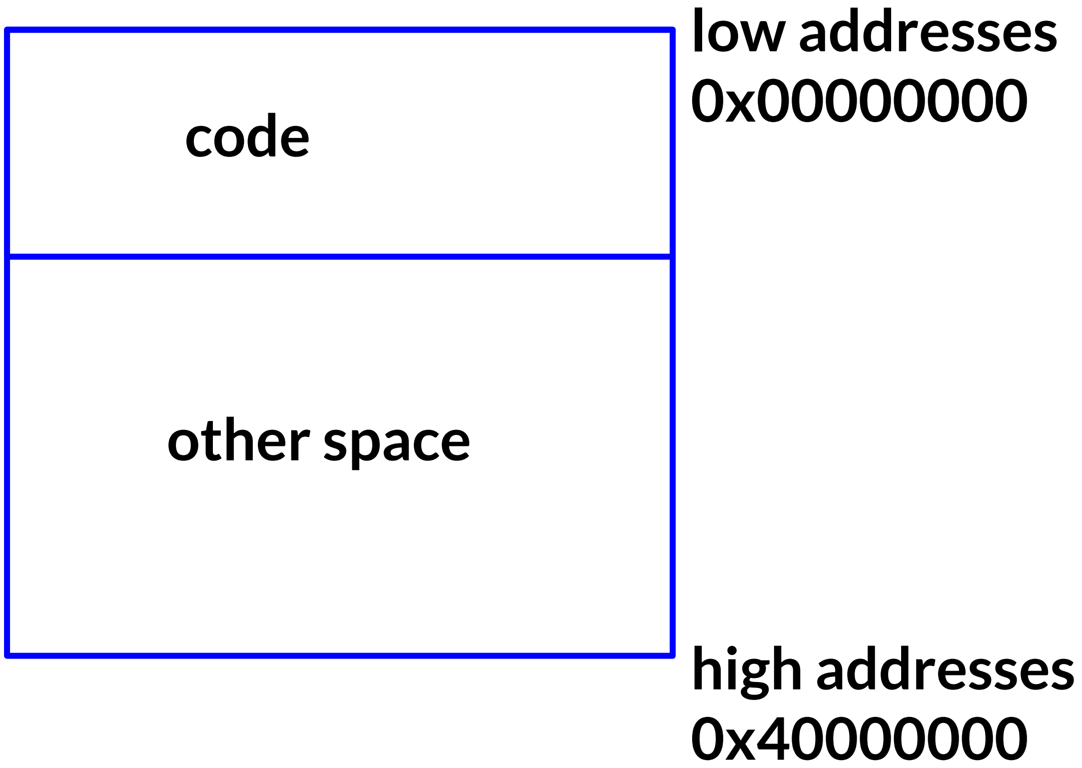
- An executing program is structured into several sections: code, data (stack, heap), etc.

Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer's entire physical memory

Review: Virtual Memory Picture

a program's virtual memory:



Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer's entire physical memory
  - the OS and the hardware work hard to give us this abstraction

Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer’s entire physical memory
  - the OS and the hardware work hard to give us this abstraction
- Our compiler needs to **predict** what the program will do and where it will store data to emit the right instructions

Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer's entire physical memory
  - the OS and the hardware work hard to give us this abstraction
- Our compiler needs to **predict** what the program will do and where it will store data to emit the right instructions
  - we generally use **conventions** to enable separate compilation

Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer's entire physical memory
  - the OS and the hardware work hard to give us this abstraction
- Our compiler needs to **predict** what the program will do and where it will store data to emit the right instructions
  - we generally use **conventions** to enable separate compilation
- We'd like to generate code is that is both **correct** and **fast**

Review: Run-time Organization

- An executing program is structured into several sections: code, data (stack, heap), etc.
- **Virtual memory** lets us pretend that each program has access to the computer's entire physical memory
  - the OS and the hardware work hard to give us this abstraction
- Our compiler needs to **predict** what the program will do and where it will store data to emit the right instructions
  - we generally use **conventions** to enable separate compilation
- We'd like to generate code is that is both **correct** and **fast**
  - when these conflict, **correctness** comes first

Review: Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order

(note that we know these assumptions are false in real life! See last lecture for details...)

Review: Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**

(note that we know these assumptions are false in real life! See last lecture for details...)

Review: Activation Trees

- **Definition**: Each invocation of some procedure P is an ***activation*** of P

Review: Activation Trees

- **Definition**: Each invocation of some procedure P is an ***activation*** of P
- Assumption (2) requires that procedure activations are **properly nested**

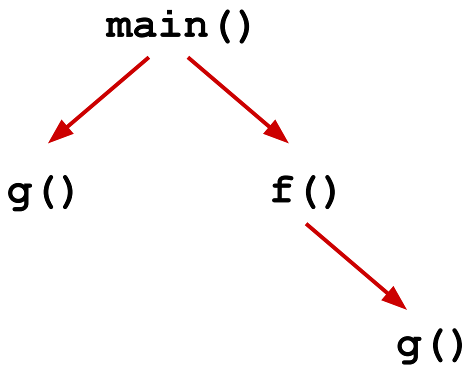
Review: Activation Trees

- **Definition**: Each invocation of some procedure P is an ***activation*** of P
- Assumption (2) requires that procedure activations are **properly nested**
- As a result, we can depict activation **lifetimes** as a **tree**

Review: Activation Trees

- **Definition**: Each invocation of some procedure P is an *activation* of P
- Assumption (2) requires that procedure activations are **properly nested**
- As a result, we can depict activation **lifetimes** as a **tree**
- Example ->

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```



Activation Tree Notes

- The activation tree **depends on run-time behavior**
  - The activation tree may be different for every program input
- Since activations are properly nested, a **stack** can track currently active procedures
  - This is the ***call stack***

Activation Tree Example Revisited

- Let's track activations with a stack on the example from before:

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

<u>Stack</u>

Activation Tree Example Revisited

- Let’s track activations with a stack on the example from before:

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

Stack

```
main()
```

Activation Tree Example Revisited

- Let's track activations with a stack on the example from before:

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

Stack

`main()`

`g()`

Activation Tree Example Revisited

- Let's track activations with a stack on the example from before:

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

Stack

`main()`

`f()`

Activation Tree Example Revisited

- Let's track activations with a stack on the example from before:

```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

Stack

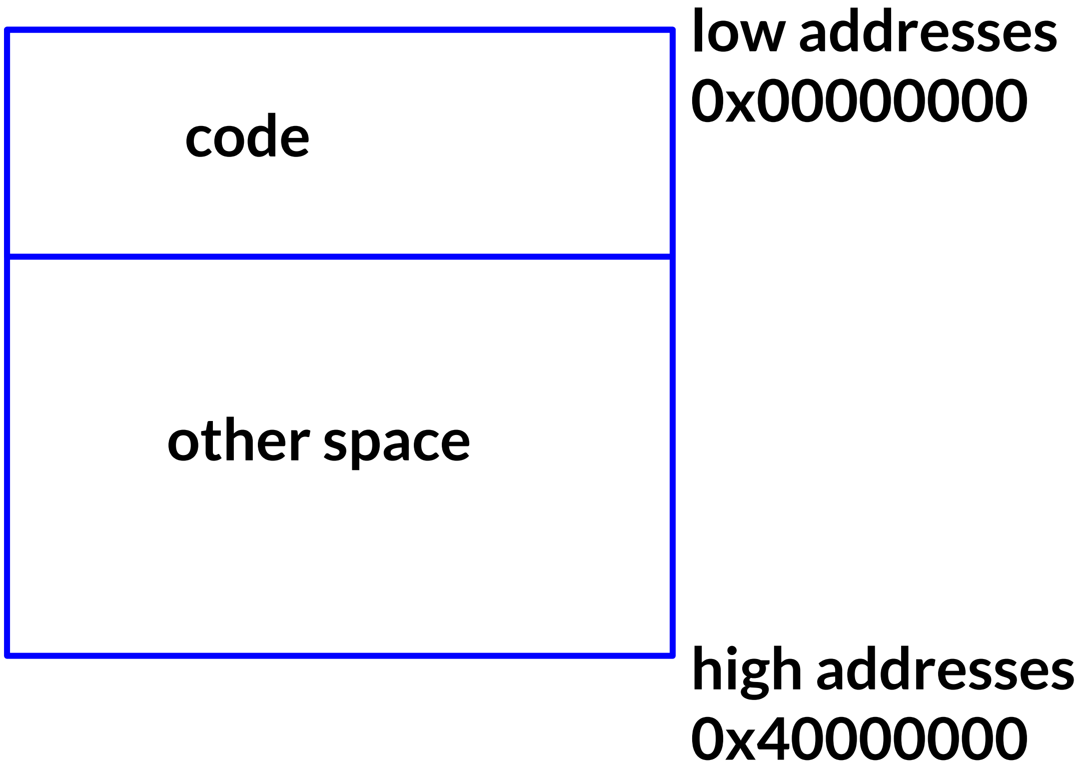
`main()`

`f()`

`g()`

Revised Memory Layout

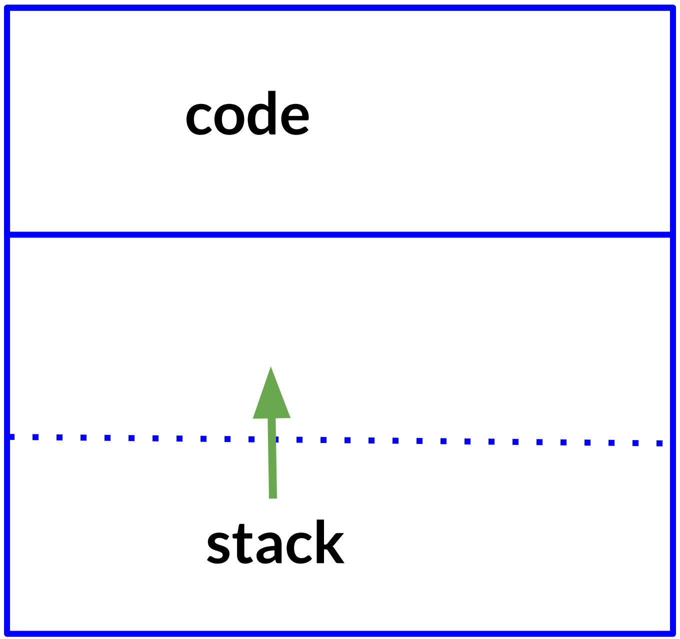
a program's virtual memory:



Revised Memory Layout

a program's virtual memory:

low addresses
0x00000000



high addresses
0x40000000

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an ***activation record*** (***AR***) or ***frame***

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an ***activation record*** (***AR***) or ***frame***
- If procedure **F** calls **G**, then **G**'s activation record contains a mix of info about **F** and **G**.

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an ***activation record*** (*AR*) or ***frame***
- If procedure **F** calls **G**, then **G**'s activation record contains a mix of info about **F** and **G**.
  - **F** is "***suspended***" until **G** completes, at which point **F** resumes.

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to m
  procedure activation is calle
  record (*AR*) or *frame*
- If procedure **F** calls **G**, then G
  contains a mix of info about
  - **F** is "*suspended*" until **G** c
    point **F** resumes.

What's in **G**'s AR when **F** calls **G**?

- G's AR contains information needed to resume execution of F
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)
  - Space for G's local variables

Activation Records

- On many machines the stack
addresses and grows toward
- The information needed to m
procedure activation is calle
record (***AR***) or ***frame***
- If procedure **F** calls **G**, then G
contains a mix of info about
  - **F** is "***suspended***" until **G** c
  point **F** resumes.

What's in **G**'s AR when **F** calls **G**?

- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)
  - Space for G's local variables

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to r
  procedure activation is calle
  record (*AR*) or *frame*
- If procedure **F** calls **G**, then **G**
  contains a mix of info about
  - **F** is "*suspended*" until **G** c
    point **F** resumes.

What's in **G**'s AR when **F** calls **G**?

- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to r
  procedure activation is calle
  record (***AR***) or ***frame***
- If procedure **F** calls **G**, then G
  contains a mix of info about
  - **F** is "***suspended***" until **G** c
    point **F** resumes.

What's in **G**'s AR when **F** calls **G**?

- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)

Activation Records

- On many machines the stack
addresses and grows toward
- The information needed to m
procedure activation is calle
record (*AR*) or *frame*
- If procedure **F** calls **G**, then **G**
contains a mix of info about
  - **F** is "*suspended*" until **G** c
point **F** resumes.

What's in **G**'s AR when **F** calls **G**?

- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)
  - Space for G's local variables

Contents of a Typical AR (for some procedure **G**)

Contents of a Typical AR (for some procedure **G**)

- Space for **G**’s return value

Contents of a Typical AR (for some procedure **G**)

- Space for **G**’s return value
- Actual parameters

Contents of a Typical AR (for some procedure **G**)

- Space for **G**’s return value
- Actual parameters
- Pointer to the previous activation record
    - This ***control link*** points back to the AR of **F** (caller of **G**)

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
    - This ***control link*** points back to the AR of **F** (caller of **G**)
        - sometimes also called the ***frame pointer***

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - This ***control link*** points back to the AR of **F** (caller of **G**)
    - sometimes also called the ***frame pointer***
- Machine status prior to calling **G**
  - Local variables
  - Register and program counter contents

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - This ***control link*** points back to the AR of **F** (caller of **G**)
    - sometimes also called the ***frame pointer***
- Machine status prior to calling **G**
  - Local variables
  - Register and program counter contents
- Other temporary values

Revisiting An Example

```
class Main {
  g() : Int { 1 };
  f(x : Int) : Int {
    if x = 0
      then g()
      else f(x - 1) (**)
    fi
  };
  main() : Int {{
    f(3); (*) }};
};
```

Revisiting An Example

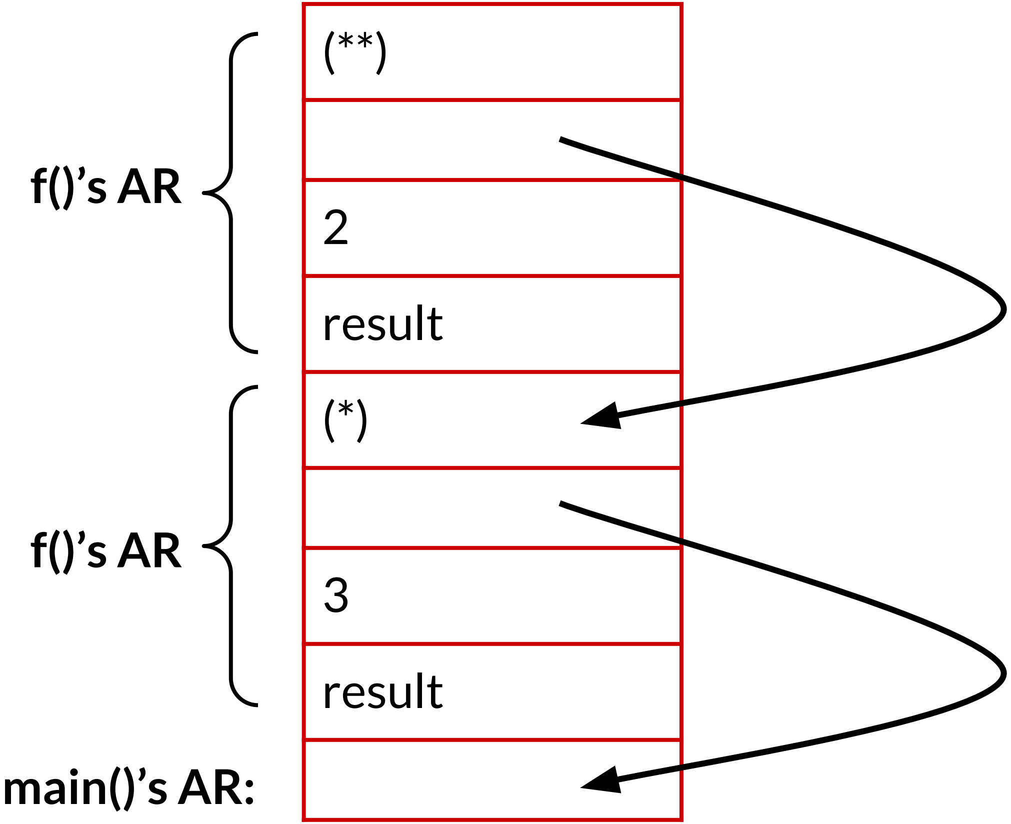
```
class Main {
  g() : Int { 1 };
  f(x : Int) : Int {
    if x = 0
      then g()
      else f(x - 1) (**)
    fi
  };
  main() : Int {{
    f(3); (*) }};
};
```

AR for **f**:

return address
control link
argument
space for result

Revisiting An Example: Stack after 2 Calls to f()

```
class Main {
  g() : Int { 1 };
  f(x : Int) : Int {
    if x = 0
      then g()
      else f(x - 1) (**)
    fi
  };
  main() : Int {{
    f(3); (*) }};
};
```



Notes on The Example

- **main()** has no argument or local variables and its result is “never” used; its AR is uninteresting

Notes on The Example

- **main()** has no argument or local variables and its result is “never” used; its AR is uninteresting
- **(*)** and **(**)** are return addresses of the invocations of f
    - The ***return address*** is where execution resumes after a procedure call finishes

Notes on The Example

- **main()** has no argument or local variables and its result is "never" used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The ***return address*** is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

The Main Point

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that, when executed at run-time, correctly accesses locations in those activation records.

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that, when executed at run-time, correctly accesses locations in those activation records.

Thus, the AR layout and the compiler must be ***designed together****!*

Discussion

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is **nothing magic** about this organization!

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is **nothing magic** about this organization!
  - Can rearrange order of frame elements

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is **nothing magic** about this organization!
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
    - The caller must write the return address there
- There is **nothing magic** about this organization!
    - Can rearrange order of frame elements
    - Can divide caller/callee responsibilities differently
    - An organization is better if it **improves execution speed** or **simplifies code generation**

Discussion

- The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is nothing magic about this organization!
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
    - This is an important tradeoff! On an embedded device with fixed software, you might make different choices!

Discussion

- The advantage of placing t
  the caller can find it at a fi
  - The caller must write t
- There is **nothing magic** ab
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it **improves execution speed** or **simplifies code generation**
    - This is an important tradeoff! On an embedded device with fixed software, you might make different choices!

- Real compilers hold as much of the frame as possible in ***registers***
  - Especially method result and arguments
- Why?

Globals

Globals

- All references to a global variable must point to the same object
    - Can’t really store a global in an activation record

Globals

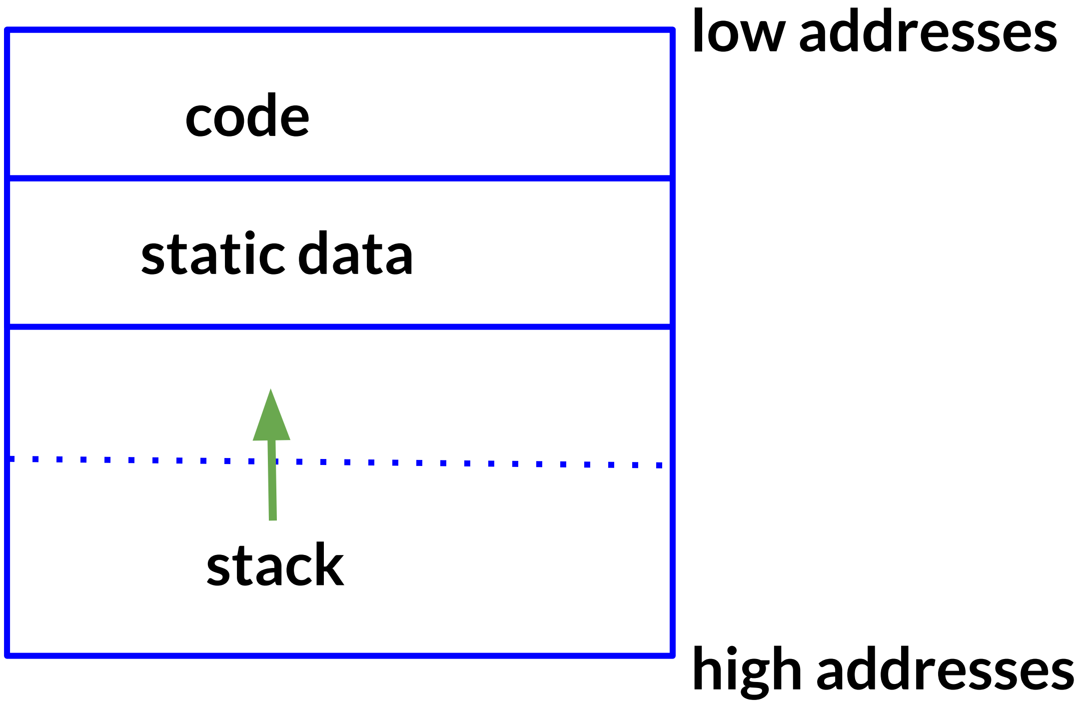
- All references to a global variable must point to the same object
  - Can't really store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "***statically allocated***"

Globals

- All references to a global variable must point to the same object
    - Can't really store a global in an activation record
- Globals are assigned a fixed address once
    - Variables with fixed address are "***statically allocated***"
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data

a program's virtual memory:



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it’s not a global

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global
  - e.g., `foo : Bar () { new Bar };`

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it’s not a global
  - e.g., `foo : Bar () { new Bar };`
    - this `Bar` value must survive deallocation of `foo`’s AR

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global
  - e.g., `foo : Bar () { new Bar };`
    - this `Bar` value must survive deallocation of `foo`'s AR
- Languages with dynamically-allocated data (such as Cool!) use a ***heap*** to store such dynamic data

Summary

Summary

- The **code area** contains object code
    - For most languages, fixed size and read only

Summary

- The code area contains object code
    - For most languages, fixed size and read only
- The **static area** contains data (not code) with fixed addresses (e.g., global data)
    - Fixed size, may be readable or writable

Summary

- The code area contains object code
  - For most languages, fixed size and read only
- The **static area** contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The **stack** contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals

Summary

- The **code area** contains object code
  - For most languages, fixed size and read only
- The **static area** contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The **stack** contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The **heap** contains all other data
  - In C, heap is managed by *malloc* and *free*

Summary

- The code area contains ob
    - For most languages, fi
- The static area contains d
  (e.g., global data)
    - Fixed size, may be rea
- The stack contains an AR for each currently active procedure
    - Each AR usually fixed size, contains locals
- The heap contains all other data
    - In C, heap is managed by *malloc* and *free*

- Both the stack and the heap grow
    - Compilers must take care that they don't grow into each other
    - Solution: start heap and stack at opposite ends of memory, let them grow towards each other

Summary

- The code area contains ob
  - For most languages, fi
- The static area contains d
  (e.g., global data)
  - Fixed size, may be read
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The heap contains all other data
  - In C, heap is managed by *malloc* and *free*

- Both the stack and the heap grow
- Compilers must take care that they don't grow into each other!

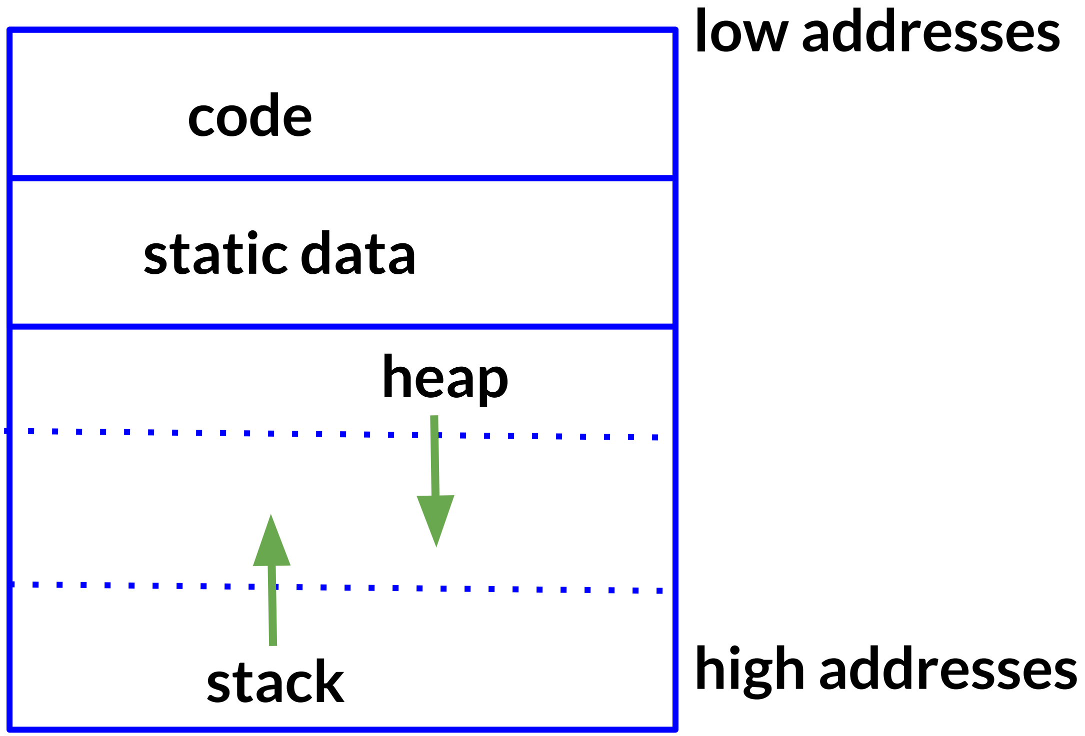
Summary

- The code area contains ob
  - For most languages, fi
- The static area contains d
  (e.g., global data)
  - Fixed size, may be read
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The heap contains all other data
  - In C, heap is managed by *malloc* and *free*

- Both the stack and the heap grow
- Compilers must take care that they don't grow into each other!
- Solution: start heap and stack at **opposite ends of memory**, let them grow towards each other

Memory Layout with Heap

a program's
virtual
memory:



Your Own Heap

- In PA3, you'll need to emit assembly code for things like:

```
let x = new Counter(5) in
let y = x in {
  x.increment(1);
  out_int( y.getCount() ); // what does this print?
}
```

Your Own Heap

- In PA3, you'll need to emit assembly code for things like:

```
let x = new Counter(5) in
let y = x in {
  x.increment(1);
  out_int( y.getCount() ); // what does this print?
}
```

- You'll need to use and manage an explicit heap (introduced today and covered in more detail in later lectures). A heap maps addresses (i.e., integers) to values.

Trivia Break: Computer Science

This living British computer scientist won the Turing award in 1980. Born in 1934 to British parents living in Sri Lanka, he studied classics as an undergraduate at Oxford and only began programming in graduate school. He spent time while a graduate student at Moscow State University in the Soviet Union as part of an exchange program, where he studied under Andrey Kolmogorov. While his work is foundational in program verification, he is best known for developing the quicksort algorithm.

Trivia Break: Gaming

This game was originally designed by Gary Gygax and Dave Arneson in the early 1970s, as a derivative of the miniature wargame *Chainmail*. The main difference between it and its predecessor wargames is that this game allows players to make a specific character to control, rather than forcing players to control an entire military formation. Its publication is commonly recognized as the beginning of the modern tabletop role-playing genre, and it has deeply influenced both modern tabletop and video games. The game is currently in its 5th edition, and is published by Wizards of the Coast.

High-level Idea

High-level Idea

- An ***operational semantics*** is a precise way of specifying how to evaluate a program.

High-level Idea

- An *operational semantics* is a precise way of specifying how to evaluate a program.
  - A **formal semantics** tells you what each expression means.

High-level Idea

- An ***operational semantics*** is a precise way of specifying how to evaluate a program.
    - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a ***variable environment*** will map variables to memory locations and a ***store*** will map memory locations to values.

High-level Idea

- An *operational semantics* is a precise way of specifying how to evaluate a program.
    - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a *variable environment* will map variables to memory locations and a ***store*** will map memory locations to values.
- We will specify Cool's semantics via **logical rules of inference** that specify how to compute the "next step" in the program
    - I warned you there was a relationship to type systems…

High-level Idea

(Rest of) today's plan:

- Motivation
- Notation
- The Rules

- An *operational semantics* is a precise w[...] evaluate a program.
  - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a *variable environment* will map variables to memory locations and a ***store*** will map memory locations to values.
- We will specify Cool's semantics via **logical rules of inference** that specify how to compute the "next step" in the program
  - I warned you there was a relationship to type systems…

Motivation

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression
- The definition of a programming language is:

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression
- The definition of a programming language is:
  - The **tokens** (lexical analysis)

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression
- The definition of a programming language is:
  - The **tokens** (lexical analysis)
  - The **grammar** (syntactic analysis/parsing)

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression
- The definition of a programming language is:
  - The **tokens** (lexical analysis)
  - The **grammar** (syntactic analysis/parsing)
  - The **type rules** (semantic analysis)

Motivation

- We must specify for every Cool expression **what happens** when it is evaluated
  - This is the **meaning** of an expression
- The definition of a programming language is:
  - The **tokens** (lexical analysis)
  - The **grammar** (syntactic analysis/parsing)
  - The **type rules** (semantic analysis)
  - The ***evaluation rules*** (interpretation)
    - also: staged hints for compilation!

“Evaluation Rules” So Far

- So far, we specified the evaluation rules **intuitively**

“Evaluation Rules” So Far

- So far, we specified the evaluation rules **intuitively**
  - We described how dynamic dispatch behaved in words
    - e.g., “just like Java”

“Evaluation Rules” So Far

- So far, we specified the evaluation rules **intuitively**
  - We described how dynamic dispatch behaved in words
    - e.g., “just like Java”
  - We talked about scoping, variables, arithmetic expressions
    - e.g., “they work as expected”

“Evaluation Rules” So Far

- So far, we specified the evaluation rules **intuitively**
  - We described how dynamic dispatch behaved in words
    - e.g., “just like Java”
  - We talked about scoping, variables, arithmetic expressions
    - e.g., “they work as expected”
- Why isn’t this description good enough?

Assembly Language Semantics

- One option: just tell you **how to compile it**

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)
- But assembly-language descriptions of language implementation have too many **irrelevant details**

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)
- But assembly-language descriptions of language implementation have too many **irrelevant details**
  - Which way the stack grows

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)
- But assembly-language descriptions of language implementation have too many **irrelevant details**
  - Which way the stack grows
  - How integers are represented on a particular machine

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)
- But assembly-language descriptions of language implementation have too many **irrelevant details**
  - Which way the stack grows
  - How integers are represented on a particular machine
  - The particular instruction set of the architecture

Assembly Language Semantics

- One option: just tell you **how to compile it**
  - (but that would be helpful...)
- But assembly-language descriptions of language implementation have too many **irrelevant details**
  - Which way the stack grows
  - How integers are represented on a particular machine
  - The particular instruction set of the architecture
- We need a **complete** but **not overly restrictive** specification

Programming Language Semantics

Programming Language Semantics

- There are many ways to specify programming language semantics

Programming Language Semantics

- There are many ways to specify programming language semantics
  - They are all equivalent but some are more suitable to various tasks than others

Programming Language Semantics

- There are many ways to specify programming language semantics
  - They are all equivalent but some are more suitable to various tasks than others

Definition: an ***operational semantics*** for a programming language L describes the evaluation of programs in L on an ***abstract machine***

Programming Language Semantics

- There are many ways to specify programming language semantics
    - They are all equivalent but some are more suitable to various tasks than others

Definition: an ***operational semantics*** for a programming language L describes the evaluation of programs in L on an ***abstract machine***

- the **abstract machine** is a **mathematical representation of computation** (where have we seen one of those before?)

Programming Language Semantics

- There are many ways to specify programming language semantics
  - They are all equivalent but some are more suitable to various tasks than others

Definition: an *operational semantics* for a programming language L describes the evaluation of programs in L on an ***abstract machine***

- the **abstract machine** is a **mathematical representation of computation** (where have we seen one of those before?)
- this semantics is most useful for **specifying an implementation**
  - and it's what we'll use for Cool

Aside: Other Kinds of Semantics

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**
  - Elegant but quite complicated

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**
  - Elegant but quite complicated
  - Popular among functional programmers (why?)

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**
    - Elegant but quite complicated
    - Popular among functional programmers (why?)
- In an ***axiomatic semantics***, the meaning of a program is expressed by describing its effect on **assertions about the program state**

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**
  - Elegant but quite complicated
  - Popular among functional programmers (why?)
- In an ***axiomatic semantics***, the meaning of a program is expressed by describing its effect on **assertions about the program state**
  - Useful for checking certain program correctness properties
    - e.g., that quicksort returns a sorted array

Aside: Other Kinds of Semantics

- In a ***denotational semantics***, the meaning of a program is expressed directly as a **mathematical object**
  - Elegant but quite complicated
  - Popular among functional programmers (why?)
- In an ***axiomatic semantics***, the meaning of a program is expressed by describing its effect on **assertions about the program state**
  - Useful for checking certain program correctness properties
    - e.g., that quicksort returns a sorted array
  - The foundation for many **program verification** tools
    - Ask me about Hoare logic in office hours to learn more!

Operational Semantics

Operational Semantics

- Once again we introduce a **formal notation**

Operational Semantics

- Once again we introduce a **formal notation**
    - Using **logical rules of inference**, cf. type rules

Operational Semantics

- Once again we introduce a **formal notation**
  - Using **logical rules of inference**, cf. type rules
- Recall the typing judgment:

$$\textit{context} \vdash \mathbf{e} : \mathbf{T}$$

(read as "in the given **context**, expression **e** has type **T**")

Operational Semantics

- Once again we introduce a **formal notation**
  - Using **logical rules of inference**, cf. type rules
- Recall the typing judgment:

  $$\textit{context} \vdash \mathbf{e} : \mathbf{T}$$

  (read as "in the given **context**, expression **e** has type **T**")

- We try something similar for evaluation:

  $$\textit{context} \vdash \mathbf{e} : \mathbf{v}$$

  (read as "in the given **context**, expression **e** evaluates to value **v**")

Example Operational Semantics Rule

$$\frac{\begin{array}{c}\textbf{context} \vdash e_1 : 5 \\ \textbf{context} \vdash e_2 : 7\end{array}}{\textbf{context} \vdash e_1 + e_2 : 12}$$

Example Operational Semantics Rule

$$\frac{\textbf{context} \vdash e_1 : 5 \qquad \textbf{context} \vdash e_2 : 7}{\textbf{context} \vdash e_1 + e_2 : 12}$$

- In general the result of evaluating an expression **depends on** the result of evaluating its subexpressions

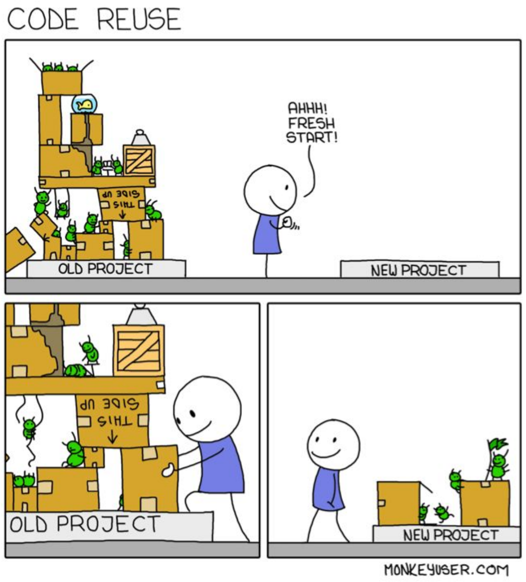
Example Operational Semantics Rule

$$\frac{\begin{array}{c}\textbf{context} \vdash e_1 : 5 \\ \textbf{context} \vdash e_2 : 7\end{array}}{\textbf{context} \vdash e_1 + e_2 : 12}$$

- In general the result of evaluating an expression **depends on** the result of evaluating its subexpressions
- The logical rules specify **everything** that is needed to evaluate an expression

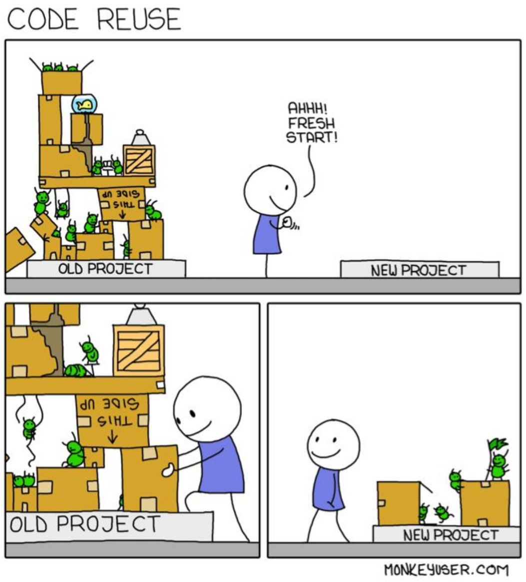
Aside: Complexity of OpSem Rules

- The operational semantics inference rules for Cool will become **complicated**
  - i.e., many hypotheses



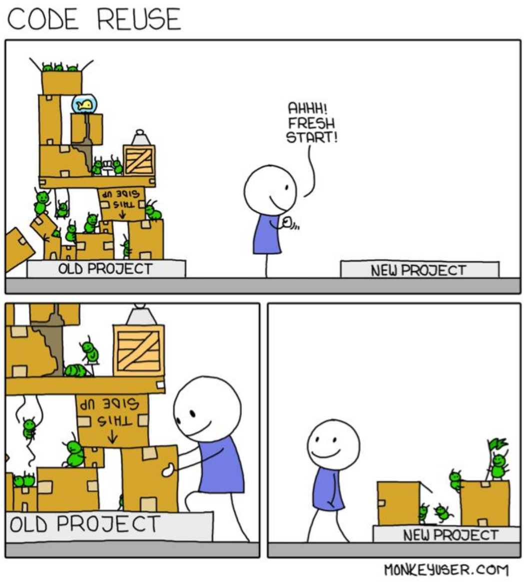
Aside: Complexity of OpSem Rules

- The operational semantics inference rules for Cool will become **complicated**
  - i.e., many hypotheses
- This may initially look daunting



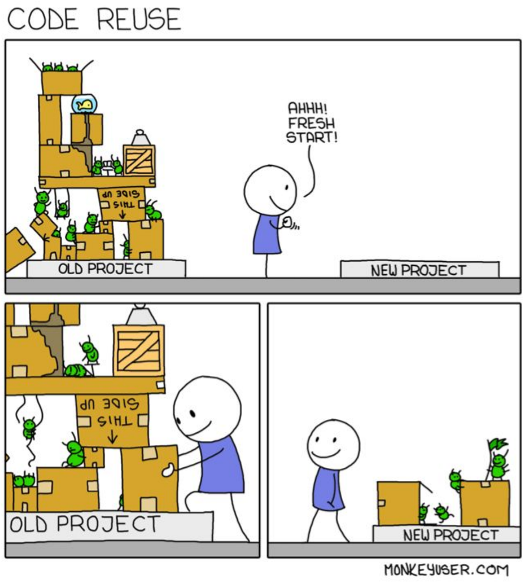
Aside: Complexity of OpSem Rules

- The operational semantics inference rules for Cool will become **complicated**
  - i.e., many hypotheses
- This may initially look daunting
  - Until you realize that the opsem rules specify **exactly how to build an interpreter**



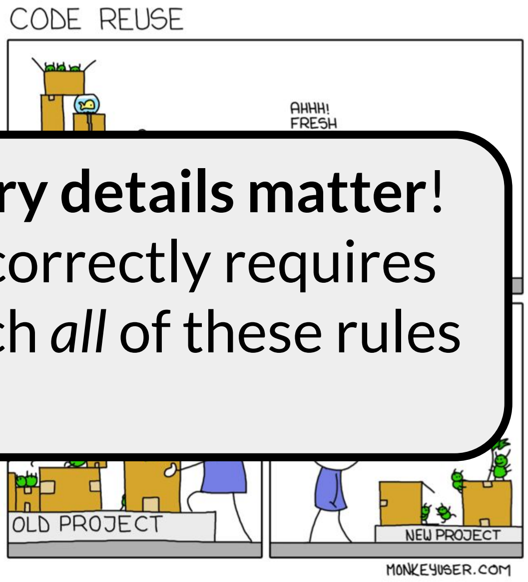
Aside: Complexity of OpSem Rules

- The operational semantics inference rules for Cool will become **complicated**
  - i.e., many hypotheses
- This may initially look daunting
  - Until you realize that the opsem rules specify **exactly how to build an interpreter**
- That is, every rule of inference in this lecture is pseudocode for an interpreter



Aside: Complexity of OpSem Rules

- The operational semantics inference rules for Cool will become **complicated**
  - i.e., many hypotheses
- This may initially look daunting
  - Until you realize that the ops specify **exactly how to build** **interpreter**
- That is, every rule of inference in this lecture is pseudocode for an interpreter



These **theory details matter!** Compiling correctly requires you to match *all* of these rules exactly…

What “Context” Is Needed?

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment

What "Context" Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`
  - We need to keep track of values of variables

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`
  - We need to keep track of values of variables
  - We need to let variables change their values during execution

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`
  - We need to keep track of values of variables
  - We need to let variables change their values during execution
- We track variables and their values with:

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`
  - We need to keep track of values of variables
  - We need to let variables change their values during execution
- We track variables and their values with:
  - an ***environment***, which tells us at what address in memory is the value of a variable stored; and

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type environment
- Consider the evaluation of `y <- x + 1`
  - We need to keep track of values of variables
  - We need to let variables change their values during execution
- We track variables and their values with:
  - an ***environment***, which tells us at what address in memory is the value of a variable stored; and
  - a ***store***, which tells us what contents each memory location holds

What “Context” Is Needed?

- Operational semantics requires contexts to handle **variables**
  - Analogy: Γ, the type e
- Consider the evaluation
  - We need to keep trac
  - We need to let variab cution

> Why do you think we need to **separate** the environment and the store? (Hint: which is static? Dynamic?)

- We track variables and their values with:
  - an ***environment***, which tells us at what address in memory is the value of a variable stored; and
  - a ***store***, which tells us what contents each memory location holds

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

- Tells in what memory location the value of a variable is stored

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

- Tells in what memory location the value of a variable is stored
    - "***Locations***" = Memory Addresses

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

- Tells in what memory location the value of a variable is stored
    - "***Locations***" = Memory Addresses
- Environment tracks in-scope variables only

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

- Tells in what memory location the value of a variable is stored
  - "***Locations***" = Memory Addresses
- Environment tracks in-scope variables only
- Example environment:

$$\mathbf{E} = [\ a : l_1, b : l_2\ ]$$

Variable Environments

Definition: A ***variable environment*** is a map from variable names to locations.

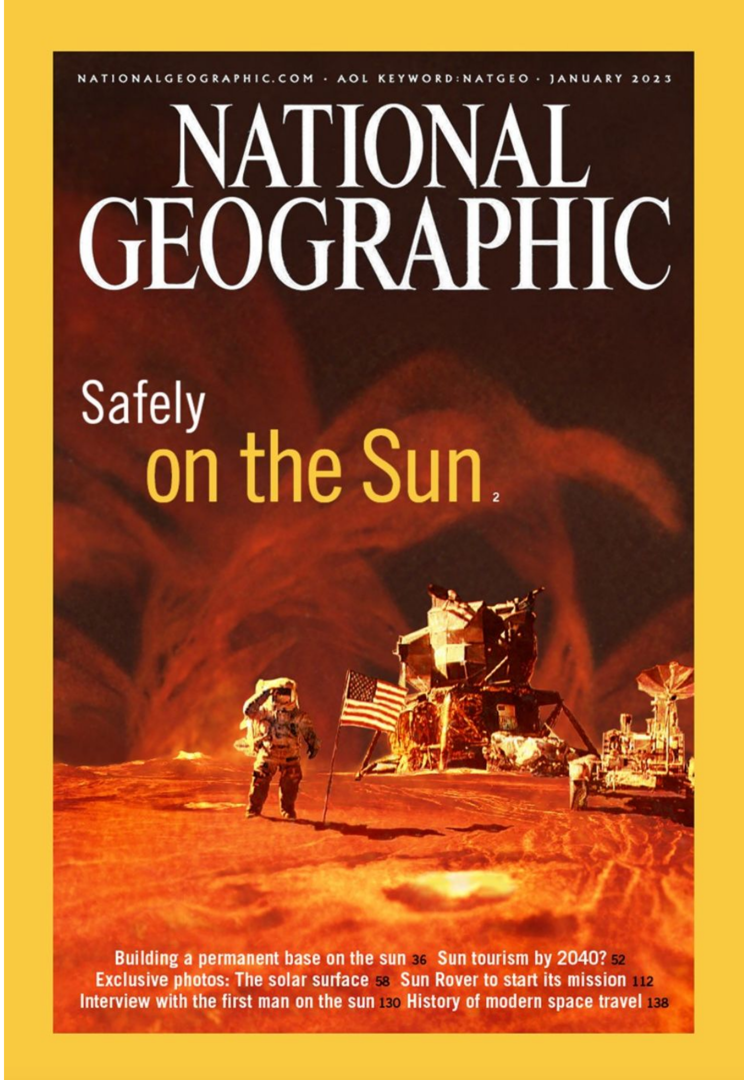
- Tells in what memory location the value of a variable is stored
  - "***Locations***" = Memory Addresses
- Environment tracks in-scope variables only
- Example environment:

  $E = [\ a : l_1, b : l_2\ ]$

- To lookup a variable a in environment E we write E(a)

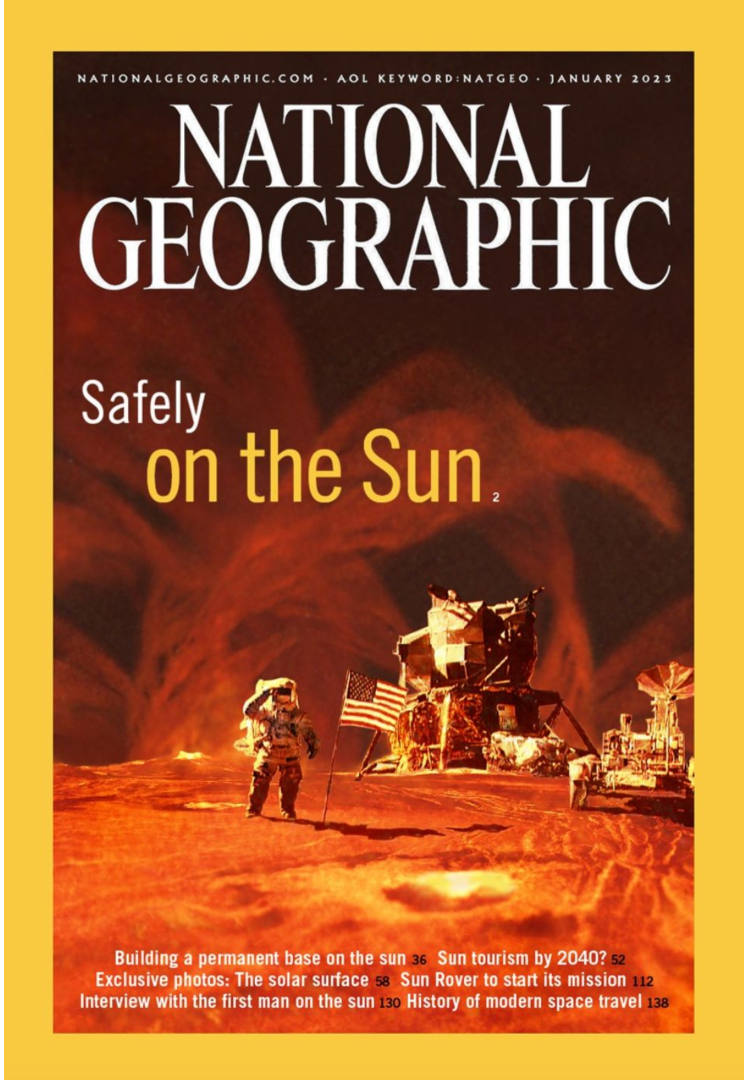
Lost?

- Environments may seem hostile and unforgiving



Lost?

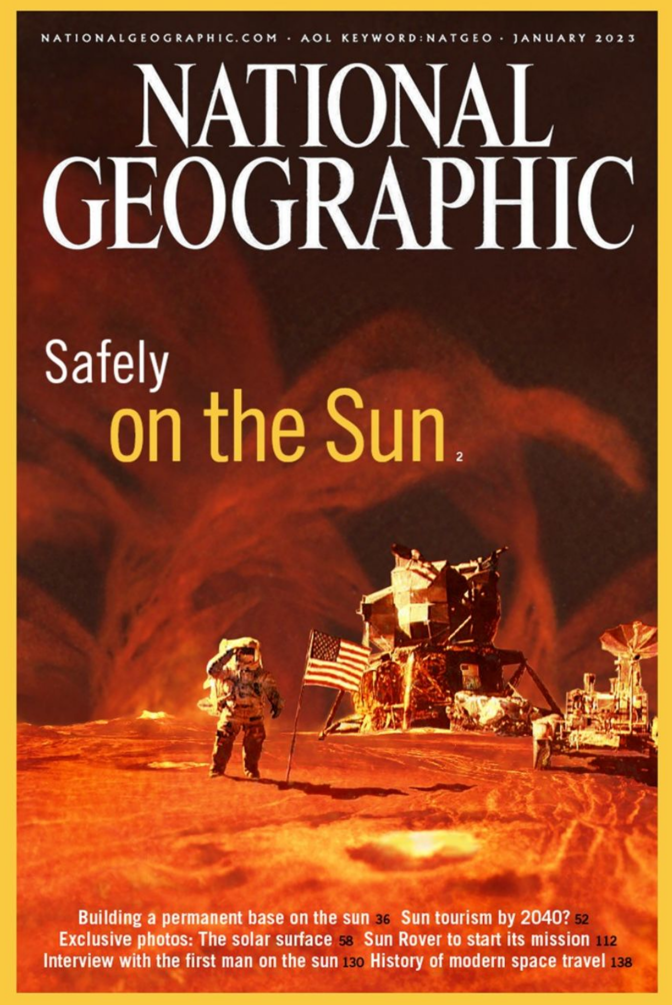
- Environments may seem hostile and unforgiving
- But soon, they'll feel just like home!



Lost?

- Environments may seem hostile and unforgiving
- But soon, they’ll feel just like home!
- Remember, an ***environment*** is:

  Names -> Locations



Stores

Definition: a ***store*** maps memory locations to values

Stores

Definition: a ***store*** maps memory locations to values

- Example store:

  $S = [\ \mathtt{l}_1 \text{ -> } 5,\ \mathtt{l}_2 \text{ -> } 7\ ]$

Stores

Definition: a ***store*** maps memory locations to values

- Example store:

  $S = [\ \mathtt{l}_1 \text{ -> } 5,\ \mathtt{l}_2 \text{ -> } 7\ ]$

- To lookup the contents of a location $\mathtt{l}_1$ in store **S** we write $S(\mathtt{l}_1)$

Stores

Definition: a ***store*** maps memory locations to values

- Example store:

  $S = [\, l_1 \texttt{->} 5,\ l_2 \texttt{->} 7 \,]$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of the value 23 to location l_1 we write $S[23/l_1]$

Stores

Definition: a ***store*** maps memory locations to values

- Example store:

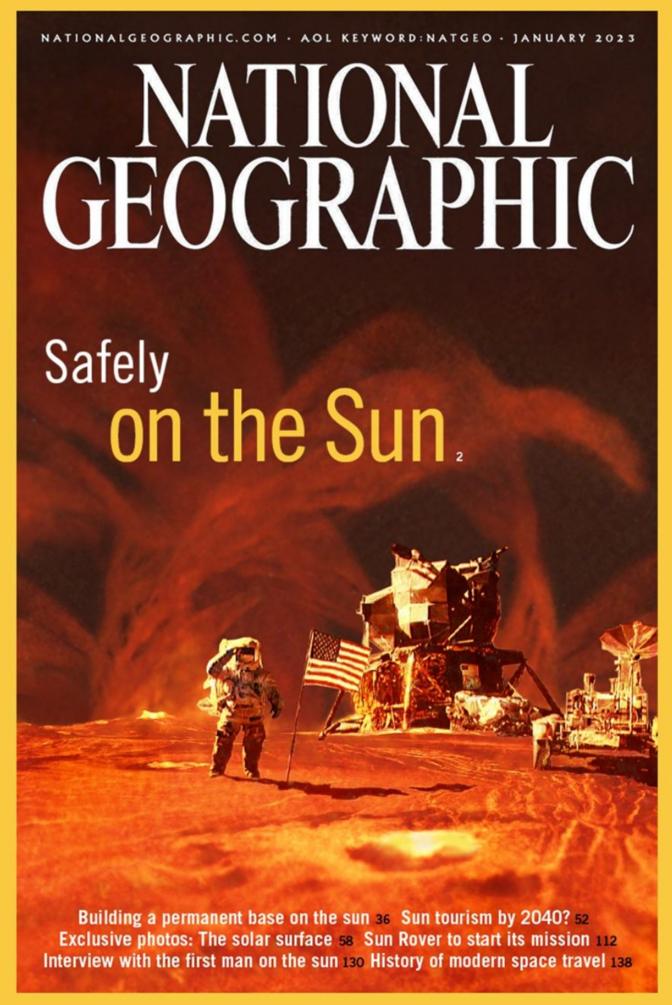
  $S = [\, l_1 \text{ -> } 5,\ l_2 \text{ -> } 7 \,]$
- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of the value 23 to location l_1 we write $S[23/l_1]$
  - This denotes a new store S' such that $S'(l_1) = 23$ and $S'(l) = S(l)$ if $l \neq l_1$

Lost?

- Environments may seem hostile and unforgiving
- But soon, they'll feel just like home!
- Remember, an ***environment*** is:

  Names -> Locations
- And a ***store*** is:

  Locations -> Values



Cool Values

Cool Values

- All **values** in Cool are objects

Cool Values

- All values in Cool are objects
    - All objects are instances of some class (the dynamic type of the object)

Cool Values

- All values in Cool are objects
  - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $\mathbf{X(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n)}$ where:

Cool Values

- All values in Cool are objects
  - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $\mathbf{X(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n)}$ where:
  - **X** is the **dynamic type** of the object (type tag)

Cool Values

- All values in Cool are objects
  - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $\mathbf{X(a_1 = l_1, ..., a_n = l_n)}$ where:
  - **X** is the **dynamic type** of the object (type tag)
  - the $\mathbf{a_i}$ are the **attributes** (including those inherited)

Cool Values

- All values in Cool are objects
  - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $\mathbf{X(a_1} = \mathtt{l_1}, \ldots, \mathbf{a_n} = \mathtt{l_n})$ where:
  - **X** is the **dynamic type** of the object (type tag)
  - the $\mathbf{a_i}$ are the **attributes** (including those inherited)
  - the $\mathtt{l_i}$ are the **locations** where the values of attributes are stored

Cool Values (continued)

- Special cases (without named attributes):

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5
  - `Bool(true)` the boolean true

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5
  - `Bool(true)` the boolean true
  - `String(4, “Cool”)` the string “Cool” of length 4

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5
  - `Bool(true)` the boolean true
  - `String(4, “Cool”)` the string “Cool” of length 4
- There is a special value `void` that is a member of all types

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5
  - `Bool(true)` the boolean true
  - `String(4, "Cool")` the string "Cool" of length 4
- There is a special value `void` that is a member of all types
  - No operations can be performed on it
    - Except for the test `isvoid`

Cool Values (continued)

- Special cases (without named attributes):
  - `Int(5)` the integer 5
  - `Bool(true)` the boolean true
  - `String(4, "Cool")` the string "Cool" of length 4
- There is a special value `void` that is a member of all types
  - No operations can be performed on it
    - Except for the test `isvoid`
  - Concrete implementations might use `NULL` here

Operational Rules of Cool

Operational Rules of Cool

- The *evaluation judgment* is

  so, **E**, **S** ⊢ **e** : **v**, **S'**

- read as:

Operational Rules of Cool

- The ***evaluation judgment*** is

  so, E, S ⊢ e : v, S'

- read as:
  - Given **so**, the current value of the `self` object;

Operational Rules of Cool

- The ***evaluation judgment*** is

  so, E, S ⊢ e : v, S'

- read as:
  - Given **so**, the current value of the `self` object;
  - and E, the current variable environment;

Operational Rules of Cool

- The ***evaluation judgment*** is

  $\textbf{so}, E, S \vdash e : v, S'$

- read as:
  - Given **so**, the current value of the `self` object;
  - and **E**, the current variable environment;
  - and **S**, the current store;

Operational Rules of Cool

- The ***evaluation judgment*** is

  $$\mathbf{so}, E, S \vdash e : v, S'$$

- read as:
  - Given **so**, the current value of the `self` object;
  - and **E**, the current variable environment;
  - and **S**, the current store;
  - and if the evaluation of **e** ***terminates***, then

Operational Rules of Cool

- The ***evaluation judgment*** is

  so, **E**, **S** ⊢ **e** : **v**, **S'**

- read as:
  - Given **so**, the current value of the `self` object;
  - and **E**, the current variable environment;
  - and **S**, the current store;
  - and if the evaluation of **e** ***terminates***, then
  - the returned value is **v**

Operational Rules of Cool

- The *evaluation judgment* is

  $so, E, S \vdash e : v, S'$

- read as:
  - Given **so**, the current value of the `self` object;
  - and **E**, the current variable environment;
  - and **S**, the current store;
  - and if the evaluation of **e** ***terminates***, then
  - the returned value is **v**
  - and the new store is **S'**

Notes on Evaluation Judgment

- The “result” of evaluating an expression is **not only** a value **but also** a new store

Notes on Evaluation Judgment

- The “result” of evaluating an expression is **not only** a value **but also** a new store
- Changes to the store model side-effects
    - ***side-effects*** = assignments to mutable variables

Notes on Evaluation Judgment

- The "result" of evaluating an expression is **not only** a value **but also** a new store
- Changes to the store model side-effects
    - ***side-effects*** = assignments to mutable variables
- The variable environment does not change
    - Nor does the value of "self"

Notes on Evaluation Judgment

- The “result” of evaluating an expression is **not only** a value **but also** a new store
- Changes to the store model side-effects
    - ***side-effects*** = assignments to mutable variables
- The variable environment does not change
    - Nor does the value of “self”
- The operational semantics specifically allows for **non-terminating evaluations**

Notes on Evaluation Judgment

- The “result” of evaluating an expression is **not only** a value **but also** a new store
- Changes to the store model side-effects
  - ***side-effects*** = assignments to mutable variables
- The variable environment does not change
  - Nor does the value of “self”
- The operational semantics specifically allows for **non-terminating evaluations**
- We’ll define one rule for each kind of expression

Operational Semantics for Base Values

$$\frac{}{\text{so}, E, S \vdash \texttt{true} : \text{Bool}(\text{true}), S}$$

Operational Semantics for Base Values

$$\frac{}{so, E, S \vdash \texttt{true} : \text{Bool(true)}, S}$$

$$\frac{}{so, E, S \vdash \texttt{false} : \text{Bool(false)}, S}$$

Operational Semantics for Base Values

$$\frac{}{so, E, S \vdash \texttt{true} : Bool(true), S}$$

$$\frac{}{so, E, S \vdash \texttt{false} : Bool(false), S}$$

$$\frac{i \text{ is any integer literal}}{so, E, S \vdash \texttt{i} : Int(i), S}$$

Operational Semantics for Base Values

$$\frac{}{\text{so}, E, S \vdash \texttt{true} : \text{Bool(true)}, S}$$

$$\frac{}{\text{so}, E, S \vdash \texttt{false} : \text{Bool(false)}, S}$$

$$\frac{i \text{ is any integer literal}}{\text{so}, E, S \vdash \texttt{i} : \text{Int}(i), S}$$

$$\frac{\begin{array}{l} s \text{ is any string literal} \\ n \text{ is the length of } s \end{array}}{\text{so}, E, S \vdash \texttt{s} : \text{String}(s, n), S}$$

Operational Semantics for Base Values

$$\frac{}{so, E, S \vdash \texttt{true} : \text{Bool(true)}, S}$$

$$\frac{}{so, E, S \vdash \texttt{false} : \text{Bool(false)}, S}$$

$$\frac{i \textit{ is any integer literal}}{so, E, S \vdash \texttt{i} : \text{Int}(i), S}$$

$$\frac{\begin{array}{l} s \textit{ is any string literal} \\ n \textit{ is the length of } s \end{array}}{so, E, S \vdash \texttt{s} : \text{String}(s, n), S}$$

- note: no side-effects in these cases (i.e., the store doesn't change)

Operational Semantics for Variables

Operational Semantics for Variables

$$\frac{E(id) = l_{id} \qquad S(l_{id}) = v}{so, E, S \vdash id : v, S}$$

Operational Semantics for Variables

$$\frac{E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S(\mathtt{l}_{\mathtt{id}}) = v}{so, E, S \vdash \mathtt{id} : v, S}$$

- Note the **double lookup** of variables

Operational Semantics for Variables

$$\frac{E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S(\mathtt{l}_{\mathtt{id}}) = v}{so, E, S \vdash \mathtt{id} : v, S}$$

- Note the **double lookup** of variables
  - First from name to location (at compile time)

Operational Semantics for Variables

$$\frac{E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S(\mathtt{l}_{\mathtt{id}}) = \mathtt{v}}{\mathbf{so}, E, S \vdash \mathtt{id} : \mathtt{v}, S}$$

- Note the **double lookup** of variables
  - First from name to location (at compile time)
  - Then from location to value (at run time)

Operational Semantics for Variables

$$\frac{E(\texttt{id}) = \texttt{l}_{\texttt{id}} \qquad S(\texttt{l}_{\texttt{id}}) = v}{so, E, S \vdash \texttt{id} : v, S}$$

- Note the **double lookup** of variables
  - First from name to location (at compile time)
  - Then from location to value (at run time)
- The store does not change

Operational Semantics for Variables

$$\frac{E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S(\mathtt{l}_{\mathtt{id}}) = v}{so, E, S \vdash \mathtt{id} : v, S}$$

- Note the **double lookup** of variables
  - First from name to location (at compile time)
  - Then from location to value (at run time)
- The store does not change
- One special case:

$$\frac{}{so, E, S \vdash \mathtt{self} : so, S}$$

Operational Semantics for Assignments

Operational Semantics for Assignments

$$\frac{\begin{array}{c} so, E, S \vdash e : v, S_1 \\ E(\mathtt{id}) = l_{id} \qquad S_2 = S_1[v/l_{id}] \end{array}}{so, E, S \vdash \mathtt{id} \texttt{ <- } e : v, S_2}$$

Operational Semantics for Assignments

$$\frac{\text{so}, E, S \vdash e : v, S_1 \qquad E(\mathtt{id}) = l_{\mathtt{id}} \qquad S_2 = S_1[v/l_{\mathtt{id}}]}{\text{so}, E, S \vdash \mathtt{id} \texttt{ <- } e : v, S_2}$$

- A three-step process:

Operational Semantics for Assignments

$$\frac{\begin{array}{c} so, E, S \vdash e : v, S_1 \\ E(\mathtt{id}) = l_{id} \qquad S_2 = S_1[v/l_{id}] \end{array}}{so, E, S \vdash \mathtt{id} \ \texttt{<-}\ e : v, S_2}$$

- A three-step process:
  - Evaluate the right-hand side to get a value **v** *and* a new store S_1

Operational Semantics for Assignments

$$\frac{\begin{array}{c}\mathrm{so}, E, S \vdash \mathtt{e} : v, S_1 \\ E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S_2 = S_1[v/\mathtt{l}_{\mathtt{id}}]\end{array}}{\mathrm{so}, E, S \vdash \mathtt{id} \texttt{ <- } \mathtt{e} : v, S_2}$$

- A three-step process:
  - Evaluate the right-hand side to get a value **v** *and* a new store S_1
  - Fetch the location of the assigned variable

Operational Semantics for Assignments

$$\frac{\begin{array}{c} so, E, S \vdash e : v, S_1 \\ E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S_2 = S_1[v/\mathtt{l}_{\mathtt{id}}] \end{array}}{so, E, S \vdash \mathtt{id} \texttt{ <- } e : v, S_2}$$

- A three-step process:
  - Evaluate the right-hand side to get a value v *and* a new store S_1
  - Fetch the location of the assigned variable
  - The result is the value v and an updated store S_2

Operational Semantics for Assignments

$$\frac{\begin{array}{c}\text{so}, E, S \vdash e : v, S_1 \\ E(\texttt{id}) = \texttt{l}_{\texttt{id}} \qquad S_2 = S_1[v/\texttt{l}_{\texttt{id}}]\end{array}}{\text{so}, E, S \vdash \texttt{id <- } e : v, S_2}$$

- A three-step process:
  - Evaluate the right-hand side to get a value **v** *and* a new store S_1
  - Fetch the location of the assigned variable
  - The result is the value **v** and an updated store S_2
- The environment doesn't change

Course Announcements

- **PA2c2** still due next Monday
  - requires typechecking + semantic analysis of everything but expressions
  - if you haven’t started yet, I’m now **very worried** for you