

# CS 320: Principles of Programming Languages

Katie Casamento, Portland State University

Winter 2018  
Week 8b: Imperative Constructs

## Referential transparency

So far we've been working only with **expressions**, which **evaluate** to a **value**.

$$e ::= x \mid (\lambda x. e) \mid (e_1 e_2) \\ \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ \mid b \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

All closed expressions in this language are *referentially transparent*:

If expression  $e$  evaluates to value  $v$ ,  
then we can replace any occurrence of  $e$  with  $v$   
**without changing the meaning of the program.**

## Side effects

A *side effect* is anything that might happen during execution outside of expressions evaluating to values.

- Setting the value of a *mutable* variable (one whose value can change over time)
- Printing to some output stream (console, file, network bridge, ...)
- Reading from some external resource (console, file, network bridge, ...)
- ...

These can all be thought of as depending on some kind of **state**:

- Setting the value of a mutable variable changes the state of the program environment
- Printing to an output stream changes the state of the stream
- Reading from an external resource depends on the state of the resource

## Side effects

Side effects **break referential transparency!**

```
int x = 0;
int f() { return x++; }
```

The first time we call  $f()$ , it returns 0,

but we **can't** replace every call to  $f()$  with 0 without changing the meaning of a program!

```
void main() { print(f()); print(f()); } // prints "01"
```

is **not the same program as**

```
void main() { print(0); print(0); } // prints "00"
```

## Statements

In an imperative language, there are **statements** (or *commands*), which **execute** to cause **side effects**.

Some common kinds of statements:

```
x = 1
    set the value of mutable variable x to 1
if e then s1 else s2
    execute statement s1 if e evaluates to true or statement s2 if e evaluates to false
while e s
    execute statement s as long as expression e evaluates to true
s1 ; s2
    execute statement s1, then execute statement s2
```

## Modeling commands

## Modeling commands

How can we add commands to our language?

A couple approaches (not the only ones!):

- Model the whole language using a *state machine*
  - Most commonly a *stack machine* or a *register machine*
  - A program is a sequence of statements
  - Expression evaluation is described as a series of commands (as in assembly)

```
x + y + z
= mov x %eax; mov y %ebx; add %eax %ebx; mov z %eax; add %eax %ebx
(returning the result in %ebx)
```
- Model commands as expressions
  - Add expression forms for commands
  - Modify evaluation semantics to account for side effects
  - Sometimes called *expression-oriented programming*

## Commands as expressions

In the expression-oriented approach, we add a new expression form to represent mutable assignments.

$e ::= \dots \mid \mathbf{x} = e$

How do we give semantics to this assignment construct?

## Evaluating commands

What **value** should a command evaluate to?

$$\begin{array}{ll} 1 + 2 & \Rightarrow^* 3 \\ (\lambda x. x + 1) 2 & \Rightarrow^* 3 \\ (x = 3) & \Rightarrow^* ??? \end{array}$$

One approach: have every command evaluate to some (sort of arbitrary) value that's already in our language.

$$(x = 3) \Rightarrow^* 3$$

This is the approach C/C++ take for assignments: an assignment evaluates to the value that was being assigned.

## Evaluating statements

What value should a command **evaluate** to?

$$\begin{array}{ll} 1 + 2 & \Rightarrow^* 3 \\ (\lambda x. x + 1) 2 & \Rightarrow^* 3 \\ x := 3 & \Rightarrow^* ??? \end{array}$$

Another approach: C/C++/Java have a **void** type.

$$(x = 3) : \text{void}$$

But what does a **value** of type **void** look like?

## The unit type

In formal type theory, we call this the *unit* type: a type with exactly one value.

$$\begin{array}{ll} e ::= \dots \mid \mathbf{unit} & \text{T-Unit} \quad \frac{}{\Gamma \vdash \mathbf{unit} : \text{Unit}} \\ t ::= \dots \mid \mathbf{Unit} & \\ \\ (x = 3) : \text{Unit} & \text{T-Assign} \quad \frac{\Gamma(x) = t \quad \Gamma \vdash e : t}{\Gamma \vdash (x = e) : \text{Unit}} \\ (x = 3) \Rightarrow^* \mathbf{unit} & \end{array}$$

**unit** can be thought of as a zero-element struct.

Since **unit** contains **no information**, it can represent the **absence** of a return value.

This captures the **return value** of a command, but we still need a way to describe the **execution** of commands in terms of side effects.

## Mutability semantics

We need a way to represent a change in the **value environment** during reductions.

$$\text{E-Assign} \quad \frac{???}{\langle \rho, (x = e) \rangle \Downarrow ???}$$

## Mutability semantics

We need a way to represent a change in the **value environment** during reductions.

One solution: reduce an expression in an environment to a value and also a **new environment**.

$$\text{E-Assign} \quad \frac{\langle \rho, e \rangle \Downarrow v \quad \text{if } e \text{ evaluates to } v \text{ under } \rho}{\langle \rho, (x = e) \rangle \Downarrow \langle \rho[x \mapsto v], () \rangle \quad \text{then } x = e \text{ updates } \rho \text{ with } x \mapsto v \text{ and returns } ()}$$

This isn't quite right, though: what if the evaluation of  $e$  produces side effects?

$x = (y = y + 1)$

This might happen if a function mutates state or prints to output before returning a value.

## Mutability semantics

We need a way to represent a change in the **value environment** during reductions.

One solution: reduce an expression in an environment to a value and also a **new environment**.

$$\text{E-Assign} \quad \frac{\langle \rho, e \rangle \Downarrow \langle \rho', v \rangle \quad \text{if } e \text{ evaluates to } v \text{ under } \rho \text{ and updates the environment to } \rho'}{\langle \rho, (x = e) \rangle \Downarrow \langle \rho'[x \mapsto v], () \rangle \quad \text{then } x = e \text{ updates } \rho' \text{ with } x \mapsto v \text{ and returns } ()}$$

In order to account for potential side effects in  $e$ , we have to update the environment **after** evaluating  $e$ , not the environment **before** evaluating  $e$ .

## Sequencing

Variable assignments aren't very interesting without a way to *sequence* commands.

$e ::= \dots \mid e_1 ; e_2$

The **value** of a sequence is the value of the **last** expression of the sequence.

$(1 ; 2) \Rightarrow^* 2$   
 $(\text{true} ; 2) \Rightarrow^* 2$

$$\text{T-Seq} \quad \frac{\Gamma \vdash e_2 : t}{\Gamma \vdash (e_1 ; e_2) : t}$$

## Sequencing semantics

When reducing a sequence of expressions, we reduce them left to right, **propagating** the environment changes from earlier expressions to later expressions.

$$\text{E-Seq} \quad \frac{\begin{array}{l} \text{if } e_1 \text{ evaluates to } v_1 \text{ under } \rho \\ \text{and updates the environment to } \rho' \end{array} \quad \begin{array}{l} \text{and } e_2 \text{ evaluates to } v_2 \text{ under } \rho' \\ \text{and updates the environment to } \rho'' \end{array}}{\langle \rho, (e_1 ; e_2) \rangle \Downarrow \langle \rho'', v_2 \rangle \quad \text{then } (e_1 ; e_2) \text{ evaluates to } v_2 \text{ under } \rho \text{ and updates the environment to } \rho''}$$

## Imperative programs

Now we have the basic tools to write (very simple) imperative programs.

```
plusOne = λ(x: num). x = x + 1; x
```

```
plusOne 1 ⇒* 2
```

```
plusOne 2 ⇒* 3
```

## Imperative semantics example

$$\text{E-Seq} \frac{\langle \rho, e_1 \rangle \Downarrow \langle \rho', v_1 \rangle \quad \langle \rho', e_2 \rangle \Downarrow \langle \rho'', v_2 \rangle}{\langle \rho, (e_1 ; e_2) \rangle \Downarrow \langle \rho'', v_2 \rangle}$$

$$\text{E-Assign} \frac{\langle \rho, e \rangle \Downarrow \langle \rho', v \rangle}{\langle \rho, (x = e) \rangle \Downarrow \langle \rho'[x \mapsto v], () \rangle}$$

$$\begin{array}{c} \text{E-Plus} \frac{\langle (x \mapsto 1), x \rangle \Downarrow 1 \quad \langle (x \mapsto 1), 1 \rangle \Downarrow 1 \quad 1 + 1 = 2}{\langle (x \mapsto 1), (x + 1) \rangle \Downarrow \langle (x \mapsto 1), 2 \rangle} \quad \text{E-Var} \frac{(x \mapsto 2)(x) = 2}{\langle (x \mapsto 2), x \rangle \Downarrow \langle (x \mapsto 2), 2 \rangle} \\ \text{E-Assign} \frac{\langle (x \mapsto 1), (x + 1) \rangle \Downarrow \langle (x \mapsto 1), 2 \rangle}{\langle (x \mapsto 1), (x = x + 1) \rangle \Downarrow \langle (x \mapsto 2), () \rangle} \\ \text{E-Seq} \frac{\langle (x \mapsto 1), (x = x + 1) \rangle \Downarrow \langle (x \mapsto 2), () \rangle \quad \langle (x \mapsto 2), x \rangle \Downarrow \langle (x \mapsto 2), 2 \rangle}{\langle (x \mapsto 1), (x = x + 1 ; x) \rangle \Downarrow \langle (x \mapsto 2), 2 \rangle} \end{array}$$

## Imperative syntax

We can define a more traditional C-style imperative syntax as syntactic sugar:

```
num plusOne(num x) { x = x + 1; return x }
plusOne = λ(x: num). x = x + 1; x
```

(Note that we haven't implemented early returns or recursion yet, though.)

When a function's return type is `unit`, we can leave out the `return` and insert `return unit` during syntax analysis.

```
unit f(x: num) { x = x + 1 }
unit f(x: num) { x = x + 1; return unit }
f = λ(x: num). x = x + 1; unit
```

## Imperative syntax

An `if` statement where the branches have type `unit` can omit an `else` branch.

```
if x > 5 then x = x + 1
if x > 5 then x = x + 1 else unit
```

When a function takes no arguments, we model it as taking a single `unit` argument.

```
unit incX() { x = x + 1; }
incX = λu. x = x + 1
```

## Explicit vs. implicit typing

The lambda syntax we've been using is partially *implicitly* typed: we don't have to specify the argument and return types of a function explicitly. (This means the typechecker is doing some amount of type **inference**.)

```
plusOne = λx. x = x + 1; x
```

In contrast, the imperative syntax is *explicitly* typed: every variable's type and function's return type must be annotated in the syntax of the program.

```
num plusOne(num x) { x = x + 1; return x }
```

Since return type inference is decidable in this language, this is just a design decision.

- Implicit typing is less work for the programmer and less distracting syntax
- Explicit typing prevents errors where type inference infers a type the programmer didn't intend

## Environment updates

How exactly should we **update** the environment?

$(x \mapsto 1, y \mapsto 2) [a \mapsto 0] =$

err? (since there's no binding to update for a)  
 $(a \mapsto 0, x \mapsto 1, y \mapsto 2)$ ? (*declare* a new binding for a)

$(x \mapsto 1, y \mapsto 2) [y \mapsto 0] =$

$(x \mapsto 1, y \mapsto 0)$ ? (update the existing binding for y)  
 $(y \mapsto 0, x \mapsto 1, y \mapsto 2)$ ? (*declare* a new binding for y, *shadowing* the old binding)

(Remember that  $\rho(x)$  returns the **leftmost** binding for x in  $\rho$ .)

And when should we **remove** values from the environment?

Python:

```
if __name__ == "__main__":  
  
    x = 1 # new binding x1  
    y = 2 # new binding y1  
    x = 3 # update x1  
  
    def f(x): # new binding x2  
        x = 4 # update x2  
        y = 5 # new binding y2  
        y = 6 # update y2  
        z = 7 # new binding z1  
        z = 8 # update z1  
  
    x = 9 # update x1  
    y = 10 # update y1  
    z = 11 # new binding z2
```

C (with GCC extensions):

```
void main() {  
  
    x = 1; // error  
    int x = 2; // new binding x1  
    int x = 3; // error  
    x = 4; // update x1  
    int y = 5; // new binding y1  
  
    void f(int x) { // new binding x2  
        x = 6; // update x2  
        y = 7; // update y2  
        int z = 8; // new binding z1  
    }  
  
    x = 9; // update x1  
    z = 10; // error  
}
```

## Scope and lifetime

## Scope

The *scope* of a variable is the part of the program where references to the variable are valid.

Different languages have different *scoping rules*, which specify where variables are in scope within different language constructs.

There are two broad categories, but a lot of variation within these categories.

- *Static* (or *lexical*) scoping resolves variable references based on the position of each reference in the AST of the program
  - Almost all modern languages
- *Dynamic* scoping resolves variable references based on the runtime environment at each reference to a variable during execution
  - bash, PowerShell, Emacs Lisp, ...
  - Occasionally opt-in for individual variables (Perl, Common Lisp, Haskell (GHC), ...)
  - Sometimes used in the implementation of *exception handling* (Java, Python, ...)

## Scoping rules

For our little imperative lambda language, we'll use a simplified version of the **static** scoping rules from C.

Our language will have a special form for *declaring local* variables:

```
e ::= ... | var x : t = e | x = e
```

```
plusOne = λx. var y : num = 1; x + y
```

With C-style local variable declarations in the imperative syntax:

```
num plusOne(num x) { num y = 1; return x + y }
```

## Scoping rules

Taking inspiration from C:

- Each function body is a *block*; the function's arguments are in scope within the body
- Each branch of an *if/then/else* construct is a block
- A local variable's scope is from its declaration to the end of the block it's declared in

□

```
unit f(num x) { // x comes into scope
  num y = 1;    // y comes into scope
  if x < y {
    num z = 2;  // z comes into scope
  } else {      // z goes out of scope
    num w = 3;  // w comes into scope
  }             // w goes out of scope
}               // x, y go out of scope
```

## Lifetime

The *lifetime* of a value is the period of execution during which it's guaranteed to be in memory.

There are many approaches to lifetime management. Among the most common:

- A variable with a *static* (or *lexical*) lifetime is in memory until the end of the scope it was declared in
- A variable with a *manually managed* lifetime is in scope until a special *free* function is called on it to explicitly free up the memory
- A *garbage-collected* variable is in scope until there are no live references to it left in the runtime environment

## Procedural evaluation

## Procedural evaluation

A simple set of rules for implementing function calls under static scope/lifetime rules can be implemented with an *activation stack*.

This is a common procedure to **execute** and **compile** procedural programs.

- When a function is *invoked* (called/applied),
  - An *activation record* (or *stack frame*) is pushed onto a global stack
  - Bindings for the function arguments are added to the environment in the function's activation record
- When a variable is declared within the function, it gets added to the environment in the function's activation record
- When a function returns, its activation record is popped off the stack

The activation record has space for the function's arguments and local variables, along with a return address and sometimes other information.

## Procedural evaluation

Other blocks, like if branches and while loop bodies, are handled similarly.

- When a variable is declared within the block, it gets added to the environment
- When execution leaves the block, all variables added in the block are removed from the environment

The address of a lexically-scoped variable can be computed in constant time,

so getting the value of a lexically-scoped variable reference takes the same amount of time regardless of how far down the activation stack it is.

## Procedural evaluation example

```

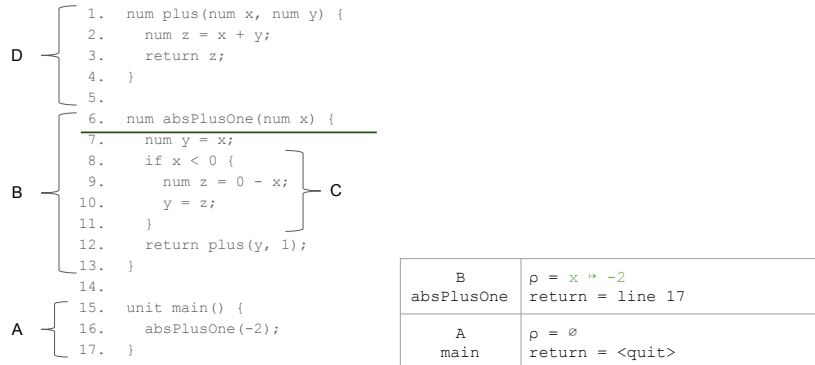
D {
  1. num plus(num x, num y) {
  2.   num z = x + y;
  3.   return z;
  4. }
  5.
  B {
    6. num absPlusOne(num x) {
    7.   num y = x;
    8.   if x < 0 {
    9.     num z = 0 - x;
   10.    y = z;
   11.   }
   12.   return plus(y, 1);
   13. }
   14.
   A {
     15. unit main() {
     16.   absPlusOne(-2);
     17. }
   }

```

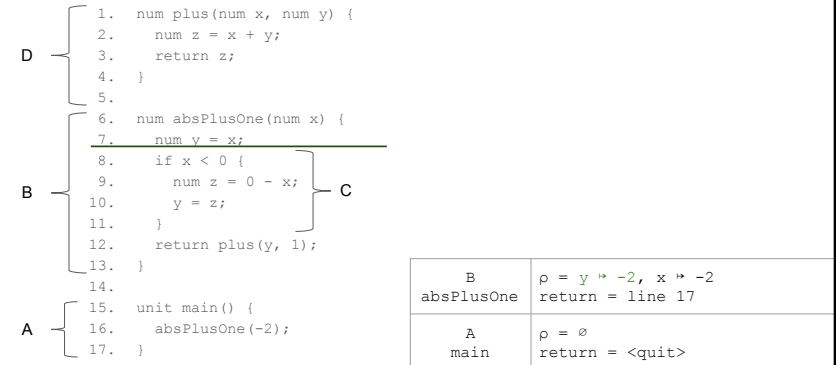
A main	$\rho = \emptyset$ return = <quit>
-----------	---------------------------------------



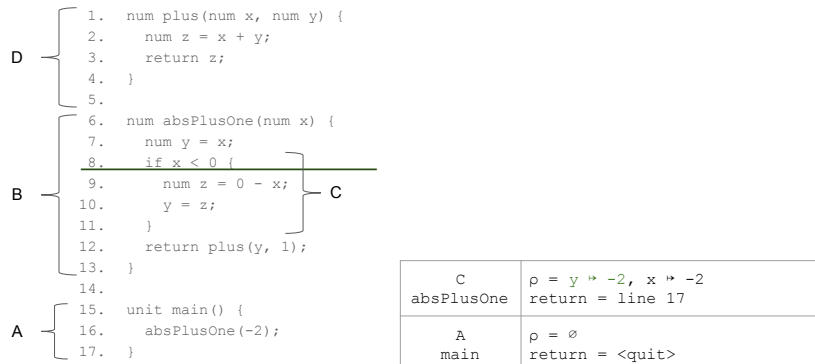
## Procedural evaluation example



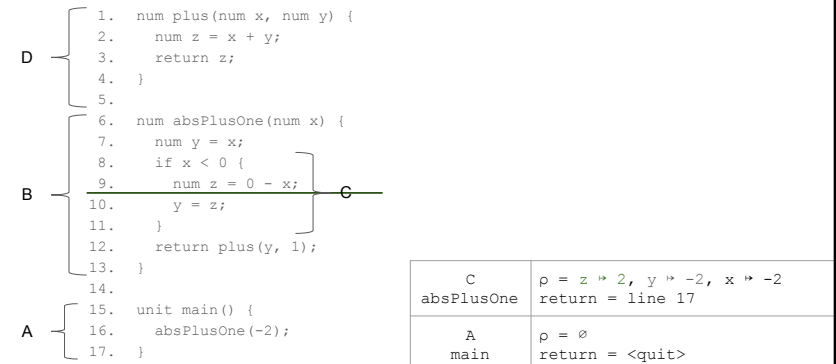
## Procedural evaluation example



## Procedural evaluation example



## Procedural evaluation example



## Procedural evaluation example

D	1.	num plus(num x, num y) {	
	2.	num z = x + y;	
	3.	return z;	
	4.	}	
	5.		
B	6.	num absPlusOne(num x) {	
	7.	num y = x;	
	8.	if x < 0 {	
	9.	num z = 0 - x;	
	10.	y = z;	C
	11.	}	
	12.	return plus(y, 1);	
	13.	}	
A	14.		
	15.	unit main() {	
	16.	absPlusOne(-2);	
	17.	}	

C	absPlusOne	$\rho = z \mapsto 2, y \mapsto 2, x \mapsto -2$ return = line 17
A	main	$\rho = \emptyset$ return = <quit>

## Procedural evaluation example

D	1.	num plus(num x, num y) {	
	2.	num z = x + y;	
	3.	return z;	
	4.	}	
	5.		
B	6.	num absPlusOne(num x) {	
	7.	num y = x;	
	8.	if x < 0 {	
	9.	num z = 0 - x;	
	10.	y = z;	C
	11.	}	
	12.	return plus(y, 1);	
	13.	}	
A	14.		
	15.	unit main() {	
	16.	absPlusOne(-2);	
	17.	}	

B	absPlusOne	$\rho = z \mapsto 2, y \mapsto 2, x \mapsto -2$ return = line 17
A	main	$\rho = \emptyset$ return = <quit>

## Procedural evaluation example

D	1.	num plus(num x, num y) {	
	2.	num z = x + y;	
	3.	return z;	
	4.	}	
	5.		
B	6.	num absPlusOne(num x) {	
	7.	num y = x;	
	8.	if x < 0 {	
	9.	num z = 0 - x;	
	10.	y = z;	C
	11.	}	
	12.	return plus(y, 1);	
	13.	}	
A	14.		
	15.	unit main() {	
	16.	absPlusOne(-2);	
	17.	}	

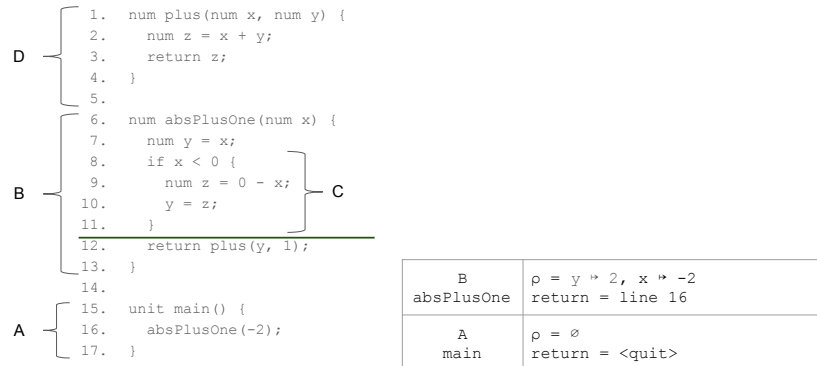
D	plus	$\rho = x \mapsto 2, y \mapsto 1$ return = line 13
B	absPlusOne	$\rho = y \mapsto 2, x \mapsto -2$ return = line 17
A	main	$\rho = \emptyset$ return = <quit>

## Procedural evaluation example

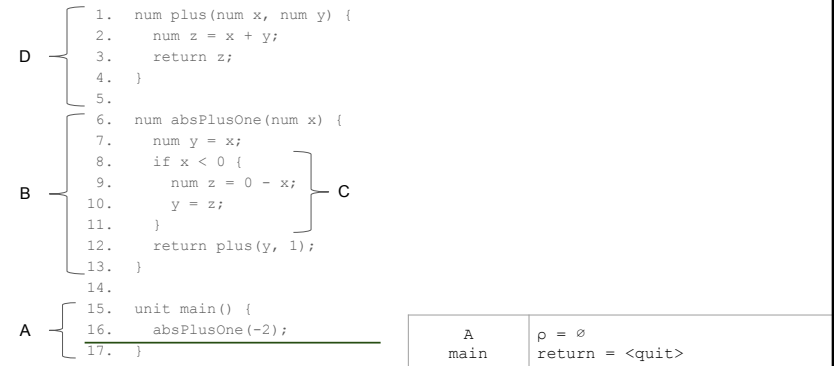
D	1.	num plus(num x, num y) {	
	2.	num z = x + y;	
	3.	return z;	
	4.	}	
	5.		
B	6.	num absPlusOne(num x) {	
	7.	num y = x;	
	8.	if x < 0 {	
	9.	num z = 0 - x;	
	10.	y = z;	C
	11.	}	
	12.	return plus(y, 1);	
	13.	}	
A	14.		
	15.	unit main() {	
	16.	absPlusOne(-2);	
	17.	}	

D	plus	$\rho = z \mapsto 3, x \mapsto 2, y \mapsto 1$ return = line 13
B	absPlusOne	$\rho = y \mapsto 2, x \mapsto -2$ return = line 17
A	main	$\rho = \emptyset$ return = <quit>

## Procedural evaluation example



## Procedural evaluation example



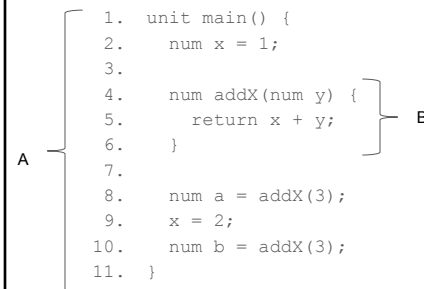
## Static and dynamic links

Scopes in a program can be syntactically *enclosed by* (or *nested in*) other scopes.

In programs with nested scopes, non-local variables are accessed through *links* between activation records.

- The *dynamic* link points to the next activation record on the stack
- The *static* link points to the nearest activation record for the enclosing static scope of the current (topmost) activation record

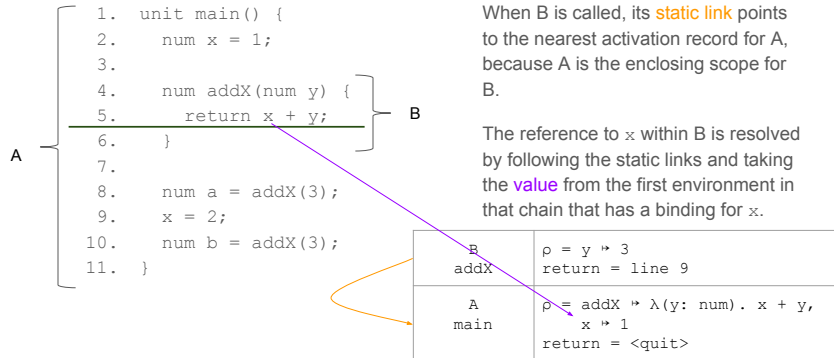
## Enclosing scopes



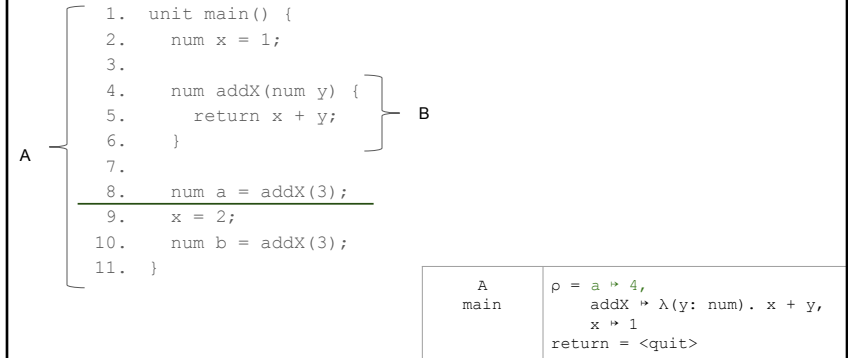
In this program, A is the (*static*) *enclosing scope* of B, because the **definition** of B is within A.



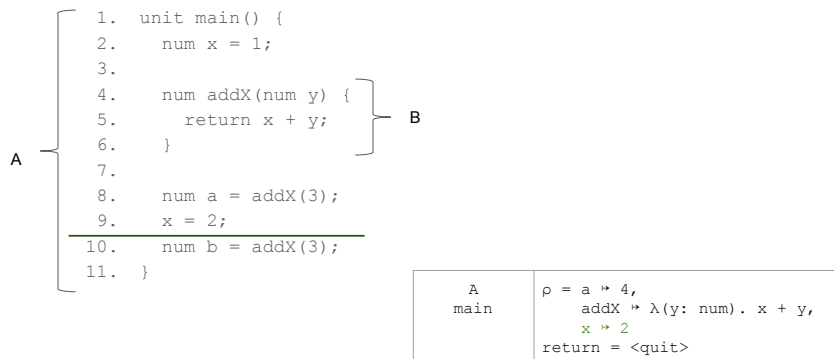
## Evaluation with static links example



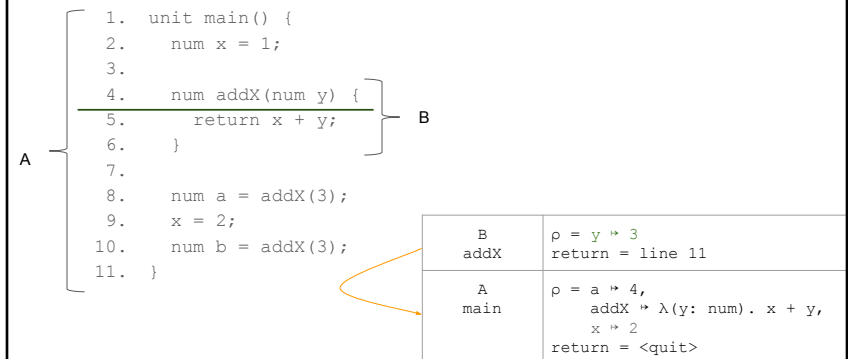
## Evaluation with static links example



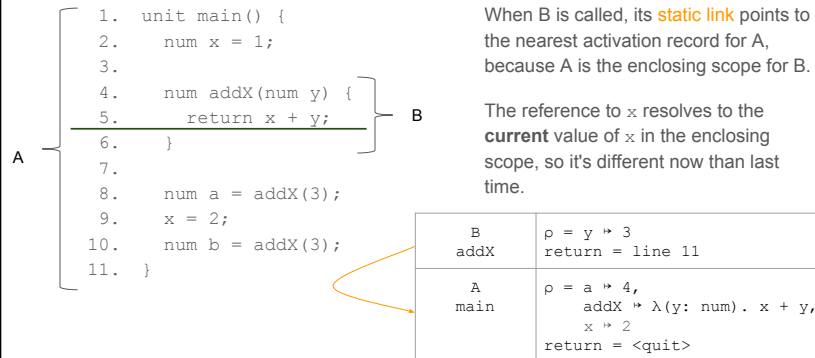
## Evaluation with static links example



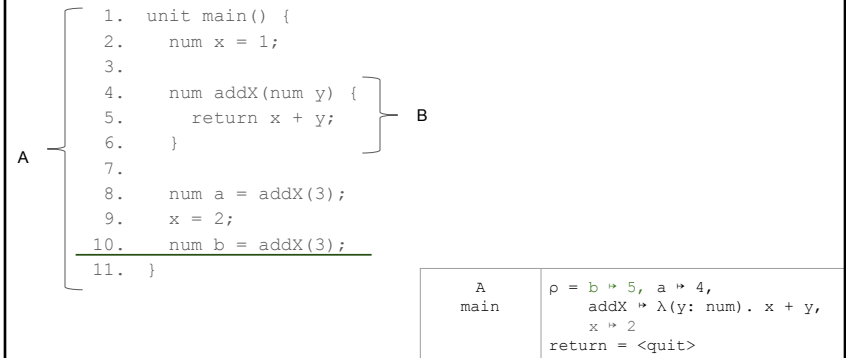
## Evaluation with static links example



## Evaluation with static links example



## Evaluation with static links example



## Calling conventions

A *calling convention* specifies how arguments are passed in to functions.

- A *call-by-value* argument gets **copied** into an activation record when it's created
  - Modifications to function arguments are "undone" when the function returns
  - Copying involves some runtime work
  - Default in most languages
- A *call-by-reference* argument is a reference to the argument's original **location**
  - Modifications to function arguments are kept when the function returns
  - Avoids the work of copying
  - Sometimes harder to reason about (especially in concurrent programs)
  - Optional in some languages (C#, Ada)
  - Can be partially simulated with pointers (C, C++) and objects (Java, Python)

## Call-by-value

In call-by-value execution of this `main` function, the value assigned to `y` is 1, because the change to the value of `x` in `f` is not visible in the calling block.

```
unit f(num x) { x = x + 1; }
```

```

unit main() {
  num x = 1;
  f(x);
  num y = x;
}
  
```

## Call-by-reference

In call-by-reference execution of this `main` function, the value assigned to `y` is 2, because the change to `x` in `f` affects the variable that was passed in from the calling block as the `x` argument.

```
unit f(num x) { x = x + 1; }

unit main() {
  num x = 1;
  f(x);
  num y = x;
}
```

## Procedural typechecking

## Procedural typechecking

Evaluation with activation records suggests a procedure for **typechecking** procedural programs.

The typechecker moves through the code top to bottom, line by line, keeping track of a typing context.

- At every line, it typechecks the expression on that line
- When it enters a function, it adds the function's arguments to the context
- When it encounters a local variable declaration, it adds the variable to the context
- When it leaves a block, it removes the variables declared in that block from the context
- When it encounters a return statement, it checks the value's type against the return type
- When it leaves a function,
  - it removes the function's arguments and local variables from the context,
  - and adds the function itself to the context

## Procedural typechecking example

```

D {
  1. num plus(num x, num y) {
  2.   num z = x + y;
  3.   return z;
  4. }
  5.
  B {
    6. num absPlusOne(num x) {
    7.   num y = x;
    8.   if x < 0 {
    9.     num z = 0 - x;
   10.     y = z;
   11.   }
   12.   return plus(y, 1);
   13. }
   14.
  A {
    15. unit main() {
    16.   absPlusOne(-2);
    17. }
  }
}
```

check	none
$\Gamma$	$\emptyset$

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);	check	none
	17.	}	$\Gamma$	x: num, y: num

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);	check	x: num && y: num
	17.	}	$\Gamma$	z: num, x: num, y: num

$$\text{T-Var} \frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\text{T-Plus} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash (e_1 + e_2) : \text{num}}$$

$$\text{T-Assign} \frac{\Gamma(x) = t \quad \Gamma \vdash e : t}{\Gamma \vdash (x = e) : \text{unit}}$$

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);	check	z: num
	17.	}	$\Gamma$	z: num, x: num, y: num

$$\text{T-Var} \frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);	check	none
	17.	}	$\Gamma$	<del>z: num, x: num, y: num,</del> plus: num $\rightarrow$ num $\rightarrow$ num



## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);		
	17.	}		

check	none
$\Gamma$	x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);		
	17.	}		

check	x: num
$\Gamma$	y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);		
	17.	}		

check	(x < 0) : bool
$\Gamma$	y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
	15.	unit main() {		
A	16.	absPlusOne(-2);		
	17.	}		

check	(0 - x) : num
$\Gamma$	z: num, y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1. num plus(num x, num y) {		T-Var	$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$
	2. num z = x + y;			
	3. return z;			
	4. }			
	5. }			
B	6. num absPlusOne(num x) {	C	T-Assign	$\frac{\Gamma(x) = t \quad \Gamma \vdash e : t}{\Gamma \vdash (x = e) : \text{unit}}$
	7. num y = x;			
	8. if x < 0 {			
	9. num z = 0 - x;			
	10. y = z;			
	11. }			
	12. return plus(y, 1);			
	13. }			
A	14. }		check	z: num
	15. unit main() {			
	16. absPlusOne(-2);			
	17. }		$\Gamma$	z: num, y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1. num plus(num x, num y) {			
	2. num z = x + y;			
	3. return z;			
	4. }			
	5. }			
B	6. num absPlusOne(num x) {	C		
	7. num y = x;			
	8. if x < 0 {			
	9. num z = 0 - x;			
	10. y = z;			
	11. }			
	12. return plus(y, 1);			
	13. }			
A	14. }		check	none
	15. unit main() {			
	16. absPlusOne(-2);			
	17. }		$\Gamma$	<del>z: num</del> , y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1. num plus(num x, num y) {		T-Num	$\frac{}{\Gamma \vdash n : \text{num}}$
	2. num z = x + y;			
	3. return z;			
	4. }			
	5. }			
B	6. num absPlusOne(num x) {	C	T-App	$\frac{e_1 : t_1 \rightarrow t_2 \quad e_2 : t_1}{(e_1 e_2) : t_2}$
	7. num y = x;			
	8. if x < 0 {			
	9. num z = 0 - x;			
	10. y = z;			
	11. }			
	12. return plus(y, 1);			
	13. }			
A	14. }		check	(plus y 1) : num
	15. unit main() {			
	16. absPlusOne(-2);			
	17. }		$\Gamma$	y: num, x: num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1. num plus(num x, num y) {			
	2. num z = x + y;			
	3. return z;			
	4. }			
	5. }			
B	6. num absPlusOne(num x) {	C		
	7. num y = x;			
	8. if x < 0 {			
	9. num z = 0 - x;			
	10. y = z;			
	11. }			
	12. return plus(y, 1);			
	13. }			
A	14. }		check	none
	15. unit main() {			
	16. absPlusOne(-2);			
	17. }		$\Gamma$	y: num, x: num, absPlusOne: num $\rightarrow$ num, plus: num $\rightarrow$ num $\rightarrow$ num

## Procedural typechecking example

D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
A	15.	unit main() {	check	none
	16.	absPlusOne(-2);	$\Gamma$	absPlusOne : num $\rightarrow$ num, plus: num $\rightarrow$ num $\rightarrow$ num
	17.	}		

## Procedural typechecking example

T-Num $\frac{}{\Gamma \vdash n : \text{num}}$				
D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
A	15.	unit main() {	check	absPlusOne(-2) is well-typed
	16.	absPlusOne(-2);	$\Gamma$	absPlusOne : num $\rightarrow$ num, plus: num $\rightarrow$ num $\rightarrow$ num
	17.	}		

## Procedural typechecking example

T-Num $\frac{}{\Gamma \vdash n : \text{num}}$				
D	1.	num plus(num x, num y) {		
	2.	num z = x + y;		
	3.	return z;		
	4.	}		
	5.			
B	6.	num absPlusOne(num x) {		
	7.	num y = x;		
	8.	if x < 0 {		
	9.	num z = 0 - x;		
	10.	y = z;		C
	11.	}		
	12.	return plus(y, 1);		
	13.	}		
	14.			
A	15.	unit main() {	check	none
	16.	absPlusOne(-2);	$\Gamma$	main: unit $\rightarrow$ unit, absPlusOne : num $\rightarrow$ num, plus: num $\rightarrow$ num $\rightarrow$ num
	17.	}		

## Summary

- Imperative programs are characterized by commands with side effects
  - A referentially transparent expression has no side effects
- We can model imperative and procedural languages:
  - As extensions of functional languages
  - As state machines
- Scoping rules and lifetime rules define where variables are valid in a program and when they're "cleaned up" at runtime
  - Different languages have different scoping and lifetime rules
  - Different variables within the same language may have different scoping and lifetime rules
- Calling conventions define how arguments are passed in to a function
  - Different arguments in the same language may have different calling conventions
  - Language constructs can simulate calling conventions that aren't supported natively