

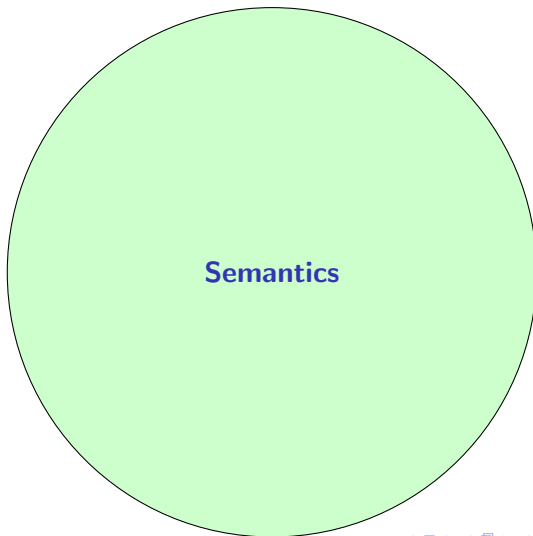


## Misc

- The midsession exam is on at **2pm, Thursday 23rd of August, 2018**.
- It should last for 50 minutes. **You must be seated in your assigned location by 2:05pm**. The exam will start right at that time.
- You may bring **one, single-sided** a4 sheet of paper with **hand-written** notes.
- A sample midsession paper will be provided very soon.
- There are **three venues** for the exam. You will be assigned a venue automatically, and can check which venue you have been assigned to on the course website.
- The venues are **OMB G31** (normal lecture theatre), **Blockhouse G6** and **Blockhouse G13**. Don't be late! Figure out where your venue is ahead of time!
- Marks will be returned to you before the census date.

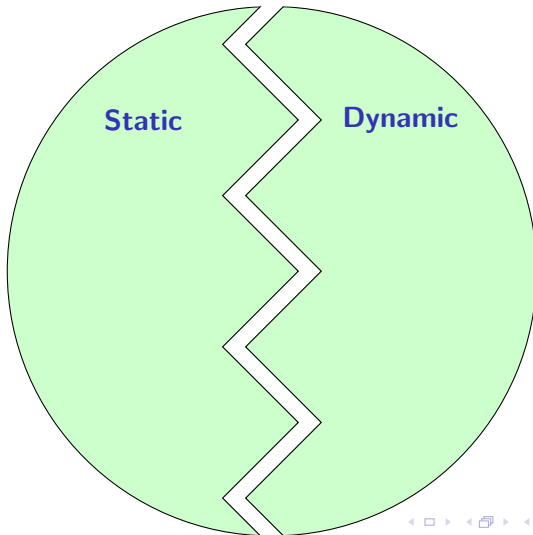
# Semantics

*σημαντικώς*



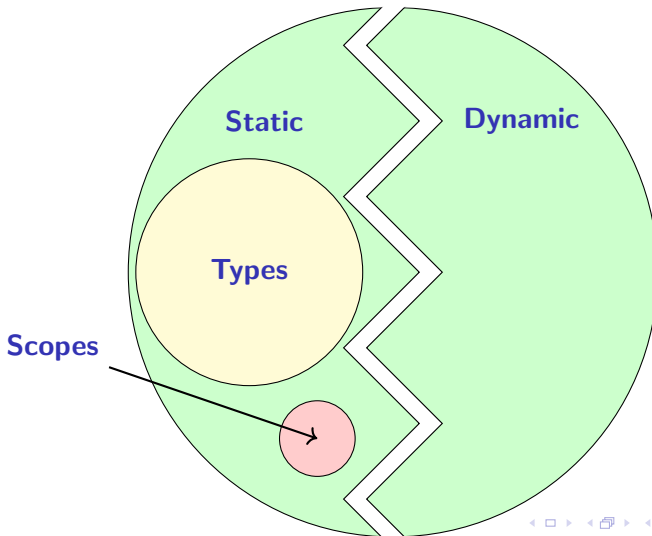
# Semantics

*σημαντικως*



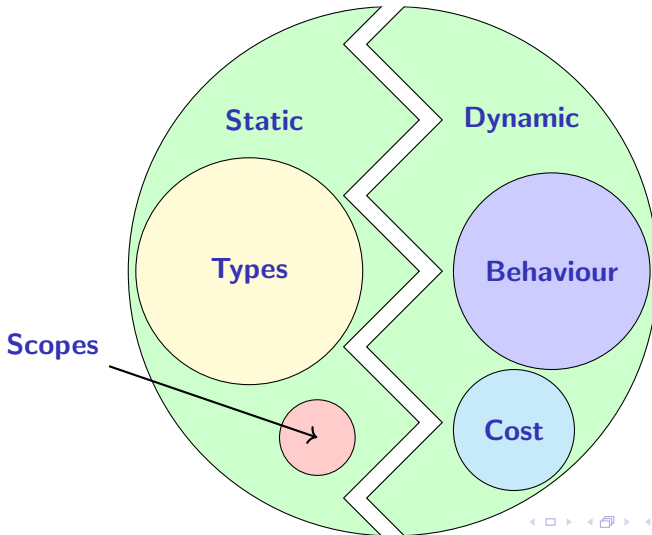
# Semantics

*σημαντικως*



# Semantics

*σημαντικως*



# Static Semantics

## Definition

The *static semantics* of a program is those significant aspects of the meaning of  $P$  that can be determined by the compiler (or an external lint tool) **without running the program**.

Recall our arithmetic expression language. What properties might we derive **statically** about those terms?

The only thing we can check is that the program is **well-scoped** (assuming FOAS).

# Static Semantics

## Definition

The *static semantics* of a program is those significant aspects of the meaning of  $P$  that can be determined by the compiler (or an external lint tool) **without running the program**.

Recall our arithmetic expression language. What properties might we derive **statically** about those terms?

The only thing we can check is that the program is **well-scoped** (assuming FOAS).



# Scope-Checking

$$\frac{}{(\text{Num } n) \text{ ok}} \quad \frac{e_1 \text{ ok} \quad e_2 \text{ ok}}{(\text{Times } e_1 \ e_2) \text{ ok}} \quad \frac{e_1 \text{ ok} \quad e_2 \text{ ok}}{(\text{Plus } e_1 \ e_2) \text{ ok}}$$

---

$$\Gamma \vdash (\text{Var } x) \text{ ok}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.

# Scope-Checking

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.

# Scope-Checking

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.

# Scope-Checking

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad x \text{ bound}, \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.

# Scope-Checking

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad x \text{ bound}, \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.

# Scope-Checking

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad x \text{ bound}, \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

## Key Idea

We keep a *context*  $\Gamma$ , a set of assumptions, on the left hand of our judgement  $e \text{ ok}$ , indicating what is required in order for  $e$  to be *well-scoped*.

This could be read as *hypothetical derivations* for the judgement  $e \text{ ok}$  or as a *binary judgement*  $\Gamma \vdash e \text{ ok}$ ; whichever you prefer.







## Scope-Checking Example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{Num } n) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Times } e_1 e_2) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Plus } e_1 e_2) \text{ ok}} \\
 \frac{(x \text{ bound}) \in \Gamma}{\Gamma \vdash (\text{Var } x) \text{ ok}} \quad \frac{\Gamma \vdash e_1 \text{ ok} \quad x \text{ bound}, \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash (\text{Let } x e_1 e_2) \text{ ok}}
 \end{array}$$

$$\frac{\frac{\frac{}{\vdash (\text{N } 3)}{\vdash (\text{N } 3)} \quad \frac{\frac{}{\text{"x"} \vdash (\text{N } 4)}{\text{"x"} \vdash (\text{N } 4)} \quad \frac{}{\text{"y"}, \text{"x"} \vdash (\text{Plus } (\text{V } \text{"x"}) (\text{V } \text{"y"}))}}{\text{"x"} \vdash (\text{Let } \text{"y"} (\text{N } 4) (\text{Plus } (\text{V } \text{"x"}) (\text{V } \text{"y"})))}}{\vdash (\text{Let } \text{"x"} (\text{N } 3) (\text{Let } \text{"y"} (\text{N } 4) (\text{Plus } (\text{V } \text{"x"}) (\text{V } \text{"y"}))))}$$





# Dynamic Semantics

**Dynamic Semantics** can be specified in many ways:

- 1 **Denotational Semantics** is the *compositional* construction of a *mathematical object* for each form of *syntax*. [COMP6752](#)
- 2 **Axiomatic Semantics** is the construction of a *proof calculus* to allow correctness of a program to be verified. [COMP2111](#)
- 3 **Operational Semantics** is the construction of a program-evaluating *state machine* or *transition system*.

## In this course

We focus mostly on *operational semantics*. We will use *axiomatic semantics* (Hoare Logic) on Thursday in the imperative programming topic. *Denotational semantics* are mostly an extension topic, except for the very next slide.

# Dynamic Semantics

**Dynamic Semantics** can be specified in many ways:

- 1 **Denotational Semantics** is the *compositional* construction of a *mathematical object* for each form of *syntax*. [COMP6752](#)
- 2 **Axiomatic Semantics** is the construction of a *proof calculus* to allow correctness of a program to be verified. [COMP2111](#)
- 3 **Operational Semantics** is the construction of a program-evaluating *state machine* or *transition system*.

## In this course

We focus mostly on *operational semantics*. We will use *axiomatic semantics* (Hoare Logic) on Thursday in the imperative programming topic. *Denotational semantics* are mostly an extension topic, except for the very next slide.

# Dynamic Semantics

**Dynamic Semantics** can be specified in many ways:

- 1 **Denotational Semantics** is the *compositional* construction of a *mathematical object* for each form of *syntax*. [COMP6752](#)
- 2 **Axiomatic Semantics** is the construction of a *proof calculus* to allow correctness of a program to be verified. [COMP2111](#)
- 3 **Operational Semantics** is the construction of a program-evaluating *state machine* or *transition system*.

## In this course

We focus mostly on *operational semantics*. We will use *axiomatic semantics* (Hoare Logic) on Thursday in the imperative programming topic. *Denotational semantics* are mostly an extension topic, except for the very next slide.

# Dynamic Semantics

**Dynamic Semantics** can be specified in many ways:

- 1 **Denotational Semantics** is the *compositional* construction of a *mathematical object* for each form of *syntax*. [COMP6752](#)
- 2 **Axiomatic Semantics** is the construction of a *proof calculus* to allow correctness of a program to be verified. [COMP2111](#)
- 3 **Operational Semantics** is the construction of a program-evaluating *state machine* or *transition system*.

## In this course

We focus mostly on **operational semantics**. We will use **axiomatic semantics** (Hoare Logic) on Thursday in the imperative programming topic. **Denotational semantics** are mostly an extension topic, except for the very next slide.

# Denotational Semantics

$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow$

$\llbracket \text{Num } n \rrbracket =$

$\llbracket \text{Var } x \rrbracket =$

$\llbracket \text{Plus } e_1 e_2 \rrbracket =$

$\llbracket \text{Times } e_1 e_2 \rrbracket =$

$\llbracket \text{Let } x e_1 e_2 \rrbracket =$



# Denotational Semantics

$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow \mathbb{Z}$

$\llbracket \text{Num } n \rrbracket = n$

$\llbracket \text{Var } x \rrbracket =$

$\llbracket \text{Plus } e_1 e_2 \rrbracket =$

$\llbracket \text{Times } e_1 e_2 \rrbracket =$

$\llbracket \text{Let } x e_1 e_2 \rrbracket =$

# Denotational Semantics

$$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow (\mathbf{Var} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z}$$

Our **denotation** for arithmetic expressions is functions from **environments** (mapping from variables to their values) to values.

$$\begin{aligned} \llbracket \mathbf{Num} \ n \rrbracket &= \lambda E. \ n \\ \llbracket \mathbf{Var} \ x \rrbracket &= \lambda E. \ E(x) \\ \llbracket \mathbf{Plus} \ e_1 \ e_2 \rrbracket &= \lambda E. \\ \llbracket \mathbf{Times} \ e_1 \ e_2 \rrbracket &= \lambda E. \\ \llbracket \mathbf{Let} \ x \ e_1 \ e_2 \rrbracket &= \lambda E. \end{aligned}$$

# Denotational Semantics

$$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow (\mathbf{Var} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z}$$

Our **denotation** for arithmetic expressions is functions from **environments** (mapping from variables to their values) to values.

$$\begin{aligned} \llbracket \text{Num } n \rrbracket &= \lambda E. n \\ \llbracket \text{Var } x \rrbracket &= \lambda E. E(x) \\ \llbracket \text{Plus } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E + \llbracket e_2 \rrbracket E \\ \llbracket \text{Times } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E \times \llbracket e_2 \rrbracket E \\ \llbracket \text{Let } x \ e_1 \ e_2 \rrbracket &= \lambda E. \end{aligned}$$

# Denotational Semantics

$$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow (\mathbf{Var} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z}$$

Our **denotation** for arithmetic expressions is functions from **environments** (mapping from variables to their values) to values.

$$\begin{aligned} \llbracket \text{Num } n \rrbracket &= \lambda E. n \\ \llbracket \text{Var } x \rrbracket &= \lambda E. E(x) \\ \llbracket \text{Plus } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E + \llbracket e_2 \rrbracket E \\ \llbracket \text{Times } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E \times \llbracket e_2 \rrbracket E \\ \llbracket \text{Let } x \ e_1 \ e_2 \rrbracket &= \lambda E. \end{aligned}$$

# Denotational Semantics

$$\llbracket \cdot \rrbracket : \mathbf{AST} \rightarrow (\mathbf{Var} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z}$$

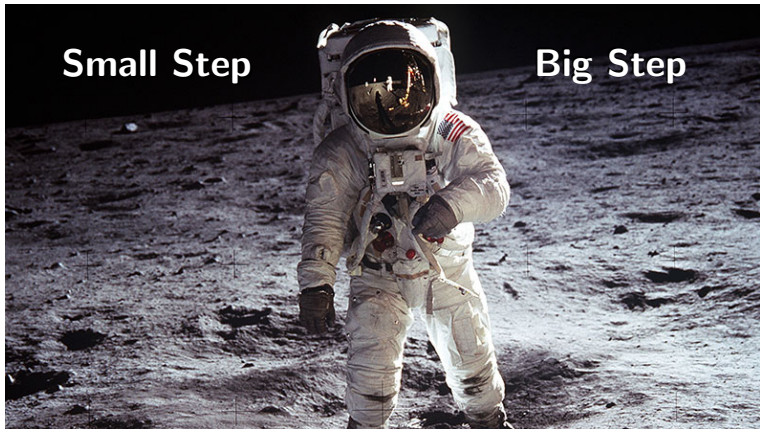
Our **denotation** for arithmetic expressions is functions from **environments** (mapping from variables to their values) to values.

$$\begin{aligned} \llbracket \text{Num } n \rrbracket &= \lambda E. n \\ \llbracket \text{Var } x \rrbracket &= \lambda E. E(x) \\ \llbracket \text{Plus } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E + \llbracket e_2 \rrbracket E \\ \llbracket \text{Times } e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_1 \rrbracket E \times \llbracket e_2 \rrbracket E \\ \llbracket \text{Let } x \ e_1 \ e_2 \rrbracket &= \lambda E. \llbracket e_2 \rrbracket (E[x := \llbracket e_1 \rrbracket E]) \end{aligned}$$

Where  $E[x := n]$  is a new environment just like  $E$  except the variable  $x$  now maps to  $n$ .

# Operational Semantics

There are two main kinds of operational semantics.



# Operational Semantics

There are two main kinds of operational semantics.



**Small Step**

**Big Step**

- Also called *natural* or *evaluation* semantics.
- One big judgement relating expressions to their values:

$$e \Downarrow v$$

# Operational Semantics

There are two main kinds of operational semantics.

## Small Step

- Also called *structural operational semantics (SOS)*.
- A judgement that specifies transitions between *states*:

$$e \mapsto e'$$



## Big Step

- Also called *natural* or *evaluation* semantics.
- One big judgement relating expressions to their values:

$$e \Downarrow v$$



# Big-Step Semantics

We need:

- A set of **evaluable expressions**  $E$
- A set of **values**  $V$
- A relation  $\Downarrow \subseteq E \times V$

## Example (Arithmetic Expressions)

$E$  is the set of all closed expressions  $\{e \mid e \text{ ok}\}$ .  $V$  is the set of integers  $\mathbb{Z}$ .

$$\frac{}{(\text{Num } n) \Downarrow n}$$
$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Plus } e_1 \ e_2) \Downarrow (v_1 + v_2)} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Times } e_1 \ e_2) \Downarrow (v_1 \times v_2)}$$

To Code Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of **evaluable expressions**  $E$
- A set of **values**  $V$
- A relation  $\Downarrow \subseteq E \times V$

## Example (Arithmetic Expressions)

$E$  is the set of all closed expressions  $\{e \mid e \text{ ok}\}$ .  $V$  is the set of integers  $\mathbb{Z}$ .

$$\frac{}{(\text{Num } n) \Downarrow n} \quad \frac{}{(\text{Let } e_1 (x. e_2)) \Downarrow}$$
$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Plus } e_1 e_2) \Downarrow (v_1 + v_2)} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Times } e_1 e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of **evaluable expressions**  $E$
- A set of **values**  $V$
- A relation  $\Downarrow \subseteq E \times V$

## Example (Arithmetic Expressions)

$E$  is the set of all closed expressions  $\{e \mid e \text{ ok}\}$ .  $V$  is the set of integers  $\mathbb{Z}$ .

$$\begin{array}{c}
 \frac{}{(\text{Num } n) \Downarrow n} \quad \frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2} \\
 \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Plus } e_1 e_2) \Downarrow (v_1 + v_2)} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Times } e_1 e_2) \Downarrow (v_1 \times v_2)}
 \end{array}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of **evaluable expressions**  $E$
- A set of **values**  $V$
- A relation  $\Downarrow \subseteq E \times V$

## Example (Arithmetic Expressions)

$E$  is the set of all closed expressions  $\{e \mid e \text{ ok}\}$ .  $V$  is the set of integers  $\mathbb{Z}$ .

$$\begin{array}{c}
 \frac{}{(\text{Num } n) \Downarrow n} \quad \frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2} \\
 \\
 \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Plus } e_1 e_2) \Downarrow (v_1 + v_2)} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Times } e_1 e_2) \Downarrow (v_1 \times v_2)}
 \end{array}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of **evaluable expressions**  $E$
- A set of **values**  $V$
- A relation  $\Downarrow \subseteq E \times V$

## Example (Arithmetic Expressions)

$E$  is the set of all closed expressions  $\{e \mid e \text{ ok}\}$ .  $V$  is the set of integers  $\mathbb{Z}$ .

$$\frac{}{(\text{Num } n) \Downarrow n} \quad \frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Plus } e_1 e_2) \Downarrow (v_1 + v_2)} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{Times } e_1 e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

## Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

Any other ways to evaluate Let?

The above is called *call-by-value* or *strict* evaluation. Below we have *call-by-name*:

$$\frac{e_2[x := e_1] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

This can be computationally very expensive, for example:

let  $x = \langle \text{very expensive computation} \rangle$  in  $x + x + x + x$

Haskell uses *call-by-need* or *lazy* evaluation, which optimises cases like this.

## Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

Any other ways to evaluate Let?

The above is called *call-by-value* or *strict* evaluation. Below we have *call-by-name*:

$$\frac{e_2[x := e_1] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

This can be computationally very expensive, for example:

`let x = (very expensive computation) in x + x + x + x`

Haskell uses *call-by-need* or *lazy* evaluation, which optimises cases like this.

## Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \quad e_2[x := (\text{Num } v_1)] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

Any other ways to evaluate Let?

The above is called *call-by-value* or *strict* evaluation. Below we have *call-by-name*:

$$\frac{e_2[x := e_1] \Downarrow v_2}{(\text{Let } e_1 (x. e_2)) \Downarrow v_2}$$

This can be computationally very expensive, for example:

`let x = <very expensive computation> in x + x + x + x`

Haskell uses *call-by-need* or *lazy* evaluation, which optimises cases like this.



# Small Step Semantics

For small step semantics, we need:

- A set of **states**  $\Sigma$
- A set of **initial states**  $I \subseteq \Sigma$
- A set of **final states**  $F \subseteq \Sigma$
- A relation  $\mapsto \subseteq \Sigma \times \Sigma$ , which specifies only “one step” of the execution.

An **execution** or **trace**  $\sigma_1 \mapsto \sigma_2 \mapsto \sigma_3 \mapsto \dots \mapsto \sigma_n$  is called **maximal** if there exists no  $\sigma_{n+1}$  such that  $\sigma_n \mapsto \sigma_{n+1}$ ; and is called **complete** if it is maximal and  $\sigma_n \in F$ .

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ (\text{Num } m))} \mapsto$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto}$$

---

$$(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{e_1 \mapsto e'_1}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto (\text{Let } e'_1 \ (x. \ e_2))}$$

$$(\text{Let } (\text{Num } n) \ (x. \ e_2)) \mapsto$$

**To Code** Let's do it in Haskell!



# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{e_1 \mapsto e'_1}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto (\text{Let } e'_1 \ (x. \ e_2))}$$

$$\frac{}{(\text{Let } (\text{Num } n) \ (x. \ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

## Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{e_1 \mapsto e'_1}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto (\text{Let } e'_1 \ (x. \ e_2))}$$

$$\frac{}{(\text{Let } (\text{Num } n) \ (x. \ e_2)) \mapsto e_2[x := \text{Num } n]}$$

**To Code** Let's do it in Haskell!

## Example

### Example (Arithmetic Expressions)

$\Sigma$  and  $I$  are the set of all closed expressions  $\{e \mid e \text{ ok}\}$ ,  $F$  is the set of evaluated expressions  $\{(\text{Num } n) \mid n \in \mathbb{Z}\}$ .

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)} \quad \frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) \ e_2) \mapsto (\text{Plus } (\text{Num } n) \ e'_2)}$$

$$\frac{}{(\text{Plus } (\text{Num } n) \ (\text{Num } m)) \mapsto (\text{Num } (n + m))}$$

(Similarly for Times)

$$\frac{e_1 \mapsto e'_1}{(\text{Let } e_1 \ (x. \ e_2)) \mapsto (\text{Let } e'_1 \ (x. \ e_2))}$$

$$\frac{}{(\text{Let } (\text{Num } n) \ (x. \ e_2)) \mapsto e_2[x := \text{Num } n]}$$

**To Code** Let's do it in Haskell!

# Equivalence

## Comparing small step and big step

Small step semantics are **lower-level**, they clearly specify the **order of evaluation**. Big step semantics give us a **result** without telling us explicitly **how it was computed**.

Having specified the dynamic semantics in these two ways, it becomes desirable to show they are **equivalent**, that is:

*If there exists a trace  $e \mapsto \cdots \mapsto (\text{Num } n)$ , then  $e \Downarrow n$ , and vice versa.*

We will need to define some notation to remove those blasted **magic dots**.

# Notation

Let  $\mapsto^*$  be the *reflexive, transitive closure* of  $\mapsto$ .

$$\frac{}{e \mapsto^* e} \quad \frac{e_1 \mapsto e_2 \quad e_2 \mapsto^* e_n}{e_1 \mapsto^* e_n}$$

We can now state our property formally as:

$$e \mapsto^* (\text{Num } n) \iff e \Downarrow n$$

## Doing the Proof

The proof will be done on the iPad, with typeset versions being uploaded as usual.

The big-step to small-step direction can be proven by reasonably straightforward rule induction:

$$\frac{e \Downarrow n}{e \xrightarrow{*} n}$$

The other direction requires the lemma:

$$\frac{e \mapsto e' \quad e' \Downarrow n}{e \Downarrow n}$$

Which is a painful case distinction, that, depending on time and my mood, I may skip presenting in the lecture.