# The Core of an Imperative Programming Language

## Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

# Outline and Road-map

1. Syntax for an Imperative Language (today)
2. Semantics for an Imperative Language (next lecture)

   $\Rightarrow$ an *executive overview* only—we return to this later
3. Syntactic sugar (next 3-4 weeks)

   $\Rightarrow$ from 1 page definition to a full-blown language

# Abstract Syntax

- Syntax $\Leftrightarrow$ context-free grammar

$$E ::= T + E \mid T \qquad\qquad T ::= N \mid (E)$$

$\Rightarrow$ too much detail (superfluous non-terminals, parentheses,...)

- Abstract Syntax gets rid of "details":

$$C ::= L := E \mid C; C \mid \textbf{if } E \textbf{ then } C \textbf{ else } C \textbf{ fi} \mid \textbf{while } E \textbf{ do } C \textbf{ od} \mid \textbf{skip}$$

$$E ::= N \mid @L \mid E + E \mid \neg E \mid E = E$$

$$L ::= loc_i \ \ (i > 0) \qquad\qquad N ::= n \ \ (n \text{ an integer numeral})$$

$\Rightarrow$ syntactic objects = trees!

# Abstract Syntax

- Syntax $\Leftrightarrow$ context-free grammar

$$E ::= T + E \mid T \qquad\qquad T ::= N \mid (E)$$

$\Rightarrow$ too much detail (superfluous non-terminals, parentheses,...)

- Abstract Syntax gets rid of "details":

$$C ::= L := E \mid C; C \mid \textbf{if } E \textbf{ then } C \textbf{ else } C \textbf{ fi} \mid \textbf{while } E \textbf{ do } C \textbf{ od} \mid \textbf{skip}$$

$$E ::= N \mid @L \mid E + E \mid \neg E \mid E = E$$

$$L ::= loc_i \ \ (i > 0) \qquad\qquad N ::= n \ \ (n \textit{ an integer numeral})$$

$\Rightarrow$ syntactic objects = trees!

# Valid Programs and Type Annotations

Typing for expressions:

$$\frac{N : int}{N : intexp} \qquad \frac{L : intloc}{@L : intexp} \qquad \frac{E_1 : intexp \quad E_2 : intexp}{E_1 + E_2 : intexp}$$

$$\frac{E : boolexp}{\neg E : boolexp} \qquad \frac{E_1 : \tau exp \quad E_2 : \tau exp}{E_1 = E_2 : boolexp} \; \tau \in \{int, bool\}$$

$\Rightarrow$ distinguishes *intexp*s from *boolexp*s.

# Valid Programs and Type Annotations

Typing for expressions:

$$\frac{N : int}{N : intexp} \qquad \frac{L : intloc}{@L : intexp} \qquad \frac{E_1 : intexp \quad E_2 : intexp}{E_1 + E_2 : intexp}$$

$$\frac{E : boolexp}{\neg E : boolexp} \qquad \frac{E_1 : \tau exp \quad E_2 : \tau exp}{E_1 = E_2 : boolexp} \; \tau \in \{int, bool\}$$

$\Rightarrow$ distinguishes *intexp*s from *boolexp*s.

# Typing Rules for Commands of Core

Imperative languages are structured around *commands*:

$$\frac{L : \mathit{intloc} \qquad E : \mathit{intexp}}{L := E : \mathit{comm}} \qquad \frac{C_1 : \mathit{comm} \qquad C_2 : \mathit{comm}}{C_1 ; C_2 : \mathit{comm}}$$

$$\frac{E : \mathit{boolexp} \qquad C : \mathit{comm}}{\textbf{while } E \textbf{ do } C \textbf{ od} : \mathit{comm}} \qquad \textbf{skip} : \mathit{comm}$$

$$\frac{E : \mathit{boolexp} \qquad C_1 : \mathit{comm} \qquad C_2 : \mathit{comm}}{\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} : \mathit{comm}}$$

# Issues with Types

1. Typing Rules vs. complex CFG
   $\Rightarrow$ are CFGs powerful enough?
2. Type annotations (tags) vs. Types (sets of values)
   $\Rightarrow$ *strong typing* and *run-time errors*?
   $\Rightarrow$ dynamic typing vs. **if**-statements?
3. Are type annotations *unique*?
   $\Rightarrow$ *unicity of typing* and type inference
   $\Rightarrow$ and if not? (extra annotations vs. union types)

# Issues with Types

1. Typing Rules vs. complex CFG
   $\Rightarrow$ are CFGs powerful enough?
2. Type annotations (tags) vs. Types (sets of values)
   $\Rightarrow$ *strong typing* and *run-time errors*?
   $\Rightarrow$ dynamic typing vs. **if**-statements?
3. Are type annotations *unique*?
   $\Rightarrow$ *unicity of typing* and type inference
   $\Rightarrow$ and if not? (extra annotations vs. union types)

# Issues with Types

1. Typing Rules vs. complex CFG
   $\Rightarrow$ are CFGs powerful enough?
2. Type annotations (tags) vs. Types (sets of values)
   $\Rightarrow$ *strong typing* and *run-time errors*?
   $\Rightarrow$ dynamic typing vs. **if**-statements?
3. Are type annotations *unique*?
   $\Rightarrow$ *unicity of typing* and type inference
   $\Rightarrow$ and if not? (extra annotations vs. union types)

# Issues with Types

1. Typing Rules vs. complex CFG
   $\Rightarrow$ are CFGs powerful enough?
2. Type annotations (tags) vs. Types (sets of values)
   $\Rightarrow$ *strong typing* and *run-time errors*?
   $\Rightarrow$ dynamic typing vs. **if**-statements?
3. Are type annotations *unique*?
   $\Rightarrow$ *unicity of typing* and type inference
   $\Rightarrow$ and if not? (extra annotations vs. union types)

# Summary

1. *Typing Rules* define (the syntax of) valid programs

   $\Rightarrow$ programs = *trees* = valid *derivations*

   $\Rightarrow$ proofs: induction on the structure of derivations

2. *Type Annotations* give *hints* about the rôle of a construct

   $\Rightarrow$ this impacts how *semantic definitions* are structured

3. *Unicity of Typing*

   $\Rightarrow$ programs have unique annotation

   $\Rightarrow$ it is sufficient to know the annotation of the root

# Semantics

a.k.a what do programs do?

Operational understanding

a “machine” model (TM, Rewriting system, . . . )

Programs as functions

programs = names of (mathematical) meanings

# Semantics

a.k.a what do programs do?

## Operational understanding

a “machine” model (TM, Rewriting system, . . . )

## Programs as functions

programs = names of (mathematical) meanings

# Semantics

a.k.a what do programs do?

## Operational understanding

a “machine” model (TM, Rewriting system, . . . )

## Programs as functions

programs = names of (mathematical) meanings

# Semantic Domains and Basic Operations

We need to understand what objects do we manipulate.

## Example

### Booleans:

$$Bool = \{true, false\}$$

Operations:

$$\begin{aligned} not &: Bool \to Bool \\ booleq &: Bool \times Bool \to Bool \end{aligned}$$

### Integers:

$$Int = \{\ldots, -1, 0, 1, \ldots\}$$

Operations:

$$\begin{aligned} plus &: Int \times Int \to Int \\ inteq &: Int \times Int \to Bool \end{aligned}$$

# What is an IMPERATIVE Language?

## Idea

*Programs manipulate a **storage vector** using assignments/lookups.*

Locations:

$$Location = \{loc_i \mid i > 0\}$$

Operations: (none)

Storage:

$$Store = \{\langle n_1, \ldots n_m \rangle \mid n_i \in Int, m > 0\}$$

Operations: $lookup$: $Location \times Store \to Int$

$update$: $Location \times Int \times Store \to Store$

# What is an IMPERATIVE Language?

## Idea

*Programs manipulate a **storage vector** using assignments/lookups.*

Locations:

$$Location = \{loc_i \mid i > 0\}$$

Operations: (none)

Storage:

$$Store = \{\langle n_1, \ldots n_m \rangle \mid n_i \in Int, m > 0\}$$

Operations: $lookup$: $Location \times Store \to Int$

$update$: $Location \times Int \times Store \to Store$

# Operational Semantics

## Idea

### Configurations

$\Rightarrow$ *descriptions of the state of the computation*

### Rewriting rules

$\Rightarrow$ *rules that "make progress" in the computation*

- finite number of rules $\Rightarrow$ local "application" $\Rightarrow$ notion of redex
  $\Rightarrow$ single step ($\rightarrow$) and multi-step reduction ($\rightarrow^*$)
- multiple redexes in a configuration
  $\Rightarrow$ nondeterminism and confluency
  $\Rightarrow$ evaluation ordering
- when are we done?
  $\Rightarrow$ notion of value (conf. w/o redexes) and computation ($\triangleright$)

# Operational Semantics

## Idea

### Configurations

$\Rightarrow$ *descriptions of the state of the computation*

### Rewriting rules

$\Rightarrow$ *rules that "make progress" in the computation*

- finite number of rules $\Rightarrow$ local "application" $\Rightarrow$ notion of redex
  $\Rightarrow$ single step ($\rightarrow$) and multi-step reduction ($\rightarrow^*$)
- multiple redexes in a configuration
  $\Rightarrow$ nondeterminism and confluency
  $\Rightarrow$ evaluation ordering
- when are we done?
  $\Rightarrow$ notion of value (conf. w/o redexes) and computation ($\triangleright$)

# Operational Semantics

## Idea

### Configurations

$\Rightarrow$ *descriptions of the state of the computation*

### Rewriting rules

$\Rightarrow$ *rules that "make progress" in the computation*

- finite number of rules $\Rightarrow$ local "application" $\Rightarrow$ notion of redex
  $\Rightarrow$ single step ($\rightarrow$) and multi-step reduction ($\rightarrow^*$)
- multiple redexes in a configuration
  $\Rightarrow$ nondeterminism and confluency
  $\Rightarrow$ evaluation ordering
- when are we done?
  $\Rightarrow$ notion of value (conf. w/o redexes) and computation ($\triangleright$)

# Operational Semantics

## Idea

### Configurations

$\Rightarrow$ *descriptions of the state of the computation*

### Rewriting rules

$\Rightarrow$ *rules that "make progress" in the computation*

- finite number of rules $\Rightarrow$ local "application" $\Rightarrow$ notion of redex
  $\Rightarrow$ single step ($\rightarrow$) and multi-step reduction ($\rightarrow^*$)
- multiple redexes in a configuration
  $\Rightarrow$ nondeterminism and confluency
  $\Rightarrow$ evaluation ordering
- when are we done?
  $\Rightarrow$ notion of value (conf. w/o redexes) and computation ($\triangleright$)

# Natural Semantics (for commands)

Computation = derivation tree:

$$\frac{[\![L]\!] \triangleright l \qquad [\![E]\!]s \triangleright n}{[\![L := E]\!]s \triangleright s'} \quad s' = update(l, n, s)$$

$$\frac{[\![C_1]\!]s \triangleright s' \qquad [\![C_2]\!]s' \triangleright s''}{[\![C_1; C_2]\!]s \triangleright s''} \qquad [\![\textbf{skip}]\!]s \triangleright s$$

$$\frac{[\![E]\!]s \triangleright true \qquad [\![C_1]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'} \qquad \frac{[\![E]\!]s \triangleright false \qquad [\![C_2]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'}$$

$$\frac{[\![E]\!]s \triangleright true \qquad [\![C]\!]s \triangleright s' \qquad [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s' \triangleright s''}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s''}$$

$$\frac{[\![E]\!]s \triangleright false}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s}$$

# Natural Semantics (for commands)

Computation = derivation tree:

$$\frac{[\![L]\!] \triangleright l \qquad [\![E]\!]s \triangleright n}{[\![L := E]\!]s \triangleright s'} \quad s' = update(l, n, s)$$

$$\frac{[\![C_1]\!]s \triangleright s' \qquad [\![C_2]\!]s' \triangleright s''}{[\![C_1; C_2]\!]s \triangleright s''} \qquad [\![\textbf{skip}]\!]s \triangleright s$$

$$\frac{[\![E]\!]s \triangleright true \qquad [\![C_1]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'} \qquad \frac{[\![E]\!]s \triangleright false \qquad [\![C_2]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'}$$

$$\frac{[\![E]\!]s \triangleright true \qquad [\![C]\!]s \triangleright s' \qquad [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s' \triangleright s''}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s''}$$

$$\frac{[\![E]\!]s \triangleright false}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s}$$

# Natural Semantics (for commands)

Computation = derivation tree:

$$\frac{[\![L]\!] \triangleright l \qquad [\![E]\!]s \triangleright n}{[\![L := E]\!]s \triangleright s'} \quad s' = update(l, n, s)$$

$$\frac{[\![C_1]\!]s \triangleright s' \qquad [\![C_2]\!]s' \triangleright s''}{[\![C_1 ; C_2]\!]s \triangleright s''} \qquad [\![\textbf{skip}]\!]s \triangleright s$$

$$\frac{[\![E]\!]s \triangleright \mathit{true} \qquad [\![C_1]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'} \qquad \frac{[\![E]\!]s \triangleright \mathit{false} \qquad [\![C_2]\!]s \triangleright s'}{[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s \triangleright s'}$$

$$\frac{[\![E]\!]s \triangleright \mathit{true} \qquad [\![C]\!]s \triangleright s' \qquad [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s' \triangleright s''}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s''}$$

$$\frac{[\![E]\!]s \triangleright \mathit{false}}{[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s \triangleright s}$$

# Properties

1. subject reduction

   $\Rightarrow$ evaluation does not change type annotation
2. soundness

   $\Rightarrow$ evaluation does not change meaning
3. computational adequacy

   $\Rightarrow$ program $p$ has a *proper meaning* $m$ then
   there is a *value* $v$ that means $m$ s.t. $p \triangleright v$

# Denotational Semantics

**Idea**

*Each type annotation is associated with a family of total functions*

$\Rightarrow$ *programs = names of functions.*

- type annotations determine the family of functions
- inductive definitions for functions (mostly)

# Expressions

*Semantic functions*:

- $[\![intexp]\!] : Store \to Int$:

$$[\![N : intexp]\!]s = [\![N : int]\!]$$
$$[\![@L : intexp]\!]s = lookup([\![L : intloc]\!], s)$$
$$[\![E_1 + E_2 : intexp]\!]s = plus([\![E_1 : intexp]\!]s, [\![E_1 : intexp]\!]s)$$

- $[\![boolexp]\!] : Store \to Bool$:

$$[\![\neg E : boolexp]\!]s = not([\![E : boolexp]\!]s)$$
$$[\![E_1 = E_2 : boolexp]\!]s = inteq([\![E_1 : intexp]\!]s, [\![E_1 : intexp]\!]s)$$
$$[\![E_1 = E_2 : boolexp]\!]s = booleq([\![E_1 : boolexp]\!]s, [\![E_1 : boolexp]\!]s)$$

# Expressions

*Semantic functions*:

- $[\![intexp]\!] : Store \to Int$:

$$[\![N : intexp]\!]s = [\![N : int]\!]$$
$$[\![@L : intexp]\!]s = lookup([\![L : intloc]\!], s)$$
$$[\![E_1 + E_2 : intexp]\!]s = plus([\![E_1 : intexp]\!]s, [\![E_1 : intexp]\!]s)$$

- $[\![boolexp]\!] : Store \to Bool$:

$$[\![\neg E : boolexp]\!]s = not([\![E : boolexp]\!]s)$$
$$[\![E_1 = E_2 : boolexp]\!]s = inteq([\![E_1 : intexp]\!]s, [\![E_1 : intexp]\!]s)$$
$$[\![E_1 = E_2 : boolexp]\!]s = booleq([\![E_1 : boolexp]\!]s, [\![E_1 : boolexp]\!]s)$$

# Semantics for Commands

*Semantic function* $[\![comm]\!] : Store \to Store$:

$$[\![L := E]\!]s = update([\![L]\!], [\![E]\!]s, s)$$

$$[\![C_1 ; C_2]\!]s = [\![C_2]\!]([\![C_1]\!]s)$$

$$[\![\textbf{skip}]\!]s = s$$

$$[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s = \begin{cases} [\![C_1]\!]s \text{ if } [\![E]\!]s = true \\ [\![C_2]\!]s \text{ if } [\![E]\!]s = false \end{cases}$$

$$[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = \begin{cases} [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]([\![C]\!]s) \\ \quad \text{if} \qquad\qquad [\![E]\!]s = true \\ s \text{ if } [\![E]\!]s = false \end{cases}$$

. . . this doesn't quite work

# Semantics for Commands

*Semantic function* $[\![comm]\!] : Store \to Store$:

$$[\![L := E]\!]s = update([\![L]\!], [\![E]\!]s, s)$$
$$[\![C_1 ; C_2]\!]s = [\![C_2]\!]([\![C_1]\!]s)$$
$$[\![\textbf{skip}]\!]s = s$$
$$[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s = \begin{cases} [\![C_1]\!]s \text{ if } [\![E]\!]s = true \\ [\![C_2]\!]s \text{ if } [\![E]\!]s = false \end{cases}$$
$$[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = \begin{cases} [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]([\![C]\!]s) \\ \quad \text{if} \qquad\qquad [\![E]\!]s = true \\ s \text{ if } [\![E]\!]s = false \end{cases}$$

. . . this doesn't quite work

# Semantics for Commands

*Semantic function* $[\![comm]\!] : Store \to Store$:

$$[\![L := E]\!]s = update([\![L]\!], [\![E]\!]s, s)$$
$$[\![C_1 ; C_2]\!]s = [\![C_2]\!]([\![C_1]\!]s)$$
$$[\![\textbf{skip}]\!]s = s$$
$$[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s = \begin{cases} [\![C_1]\!]s \text{ if } [\![E]\!]s = true \\ [\![C_2]\!]s \text{ if } [\![E]\!]s = false \end{cases}$$
$$[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = \begin{cases} [\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]([\![C]\!]s) \\ \quad \text{if} \qquad\qquad [\![E]\!]s = true \\ s \text{ if } [\![E]\!]s = false \end{cases}$$

. . . this doesn't quite work

# Semantics for Commands (corrected.)

How do we deal with looping programs?

**Idea**

*Add a **looping** value $\perp$ to the Store set.*

**Definition**

Let

$$f_0 s = \perp \qquad f_i s = \begin{cases} f_{i-1}([\![C]\!]s) & \text{if } [\![E]\!]s = true \\ s & \text{if } [\![E]\!]s = false \end{cases}$$

Define $f : Store_\perp \to Store_\perp$ as $fs = f_i s$ if there is $i > 0$ s.t. $f_i s \neq \perp$.

Set $[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = f$.

$\Rightarrow$ fix rest $[\![.]\!]s$ to be (mostly) strict!

# Semantics for Commands (corrected.)

How do we deal with looping programs?

## Idea

*Add a **looping** value $\perp$ to the Store set.*

## Definition

Let

$$f_0 s = \perp \qquad f_i s = \begin{cases} f_{i-1}([\![C]\!]s) & \text{if } [\![E]\!]s = true \\ s & \text{if } [\![E]\!]s = false \end{cases}$$

Define $f : Store_\perp \to Store_\perp$ as $fs = f_i s$ if there is $i > 0$ s.t. $f_i s \neq \perp$.

Set $[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = f$.

$\Rightarrow$ fix rest $[\![.]\!]s$ to be (mostly) strict!

# Semantics for Commands (corrected.)

How do we deal with looping programs?

## Idea

*Add a **looping** value $\perp$ to the Store set.*

## Definition

Let

$$f_0 s = \perp \qquad f_i s = \begin{cases} f_{i-1}([\![C]\!]s) & \text{if } [\![E]\!]s = \mathit{true} \\ s & \text{if } [\![E]\!]s = \mathit{false} \end{cases}$$

Define $f : \mathit{Store}_\perp \to \mathit{Store}_\perp$ as $fs = f_i s$ if there is $i > 0$ s.t. $f_i s \neq \perp$.

Set $[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = f$.

$\Rightarrow$ fix rest $[\![.]\!]s$ to be (mostly) strict!

# Semantics for Commands (corrected.)

How do we deal with looping programs?

## Idea

*Add a **looping** value $\perp$ to the Store set.*

## Definition

Let

$$f_0 s = \perp \qquad f_i s = \begin{cases} f_{i-1}([\![C]\!]s) & \text{if } [\![E]\!]s = \mathit{true} \\ s & \text{if } [\![E]\!]s = \mathit{false} \end{cases}$$

Define $f : \mathit{Store}_\perp \to \mathit{Store}_\perp$ as $fs = f_i s$ if there is $i > 0$ s.t. $f_i s \neq \perp$.

Set $[\![\textbf{while } E \textbf{ do } C \textbf{ od}]\!]s = f$.

$\Rightarrow$ fix rest $[\![.]\!]s$ to be (mostly) strict!

# Summary

1. Semantics assigns *meanings* to programs

   $\Rightarrow$ programs are names for functions
2. Two principal ways of assigning meaning
   1. operational (step-by-step)
   2. denotational
3. Type annotations "match" denotational semantics:

**Theorem**

*Let* $P : \theta$. *Then* $[\![P]\!] \in [\![\theta]\!]$.