Program Testing and Analysis: Operational Semantics

Dr. Michael Pradel

Software Lab, TU Darmstadt

Warm-up Quiz

What does the following code print?

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

Options: 5, 7, or something else

Warm-up Quiz

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

Correct answer: 5

Warm-up Quiz

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

`eval()` evaluates JavaScript code given as a string

Correct answer: 5

Warm-up Quiz

Store function into variable (functions are first-class objects)

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

Correct answer: 5

Warm-up Quiz

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

Define a function and call it immediately

Correct answer: 5

Warm-up Quiz

```
var e = eval;

(function f() {
  var x = 5;
  e("x=7")
  console.log(x);
})();
```

Indirect `eval()`: Works in global scope rather than local scope

Correct answer: 5

Follow Up on Last Week

(Hand-written notes)

Big Picture

Last lecture:

- Syntax of languages
- Representations of programs

This lecture:

- Assign meaning (= semantics) to programs
- Focus: Operational semantics of imperative languages
- Formal foundation for specifying languages and for describing dynamic analyses

Plan for Today

- **Motivation & preliminaries**
- **Abstract syntax of SIMP**
- **An abstract machine for SIMP**
- **Structural operation semantics for SIMP**
  - Small-step semantics
  - Big-step semantics

Why Do We Need Operational Semantic?

Example (C code):

```
int i = 5;
f(i++, --i);
```

← What are the actual parameters passed to f?

Option 1: 5, 5 (left-to-right)

Option 2: 4, 4 (right-to-left)

Both options are possible in C

→ Unspecified semantics

→ Compiler decides

Want: (Almost) all behavior should be clearly specified

Specifying the Semantics of Programs

- Static semantics : E.g., types
- Dynamic semantics :
  - Denotational
  - Axiomatic
  - Operational (focus)

↳ Useful for :
- Lang. design
- Lang. implementation
- Programming
- Program analysis

Preliminaries

a) Transition system

- set Config of configurations or states
- binary relation $\rightarrow \subseteq Config \times Config$
  ("transition relation")

↳ $c \rightarrow c'$ .. transition, or change of state

$\Longrightarrow$ step of computation

$\rightarrow^*$ .. reflexive-transitive closure of $\rightarrow$

$\forall c.\ c \rightarrow^* c$ $\qquad$ $\forall c, c', c''.\ c \rightarrow^* c' \wedge c' \rightarrow^* c'' \Rightarrow c \rightarrow^* c''$

deterministic : $c \rightarrow c_1 \wedge c \rightarrow c_2 \Rightarrow c_1 = c_2$

b) Rule induction

↳ define a set ("inductive set") with:

- a finite set of basic elements (axioms): $\dfrac{}{t}$
- a finite set of rules that specify how to generate elements of the set

$$\frac{h_1 \quad \dots \quad h_n}{c}$$

← hypotheses (or premises)

← conclusion

↳ <u>Example 1</u> set of natural numbers

axiom: $\dfrac{}{0}$

rule: $\dfrac{n}{n+1}$

c) <u>Proof tree</u>

↳ show that an element is in an inductive set

<u>Example 1:</u>

$$\dfrac{\dfrac{\dfrac{}{0}}{1}}{2}$$

→ 2 is a natural number

<u>Example 2:</u> show that

$$-(+(3,4),1) \Downarrow 6$$

Quiz: # axioms ? 3

rules ? 2

$$\dfrac{\dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{4 \Downarrow 4}}{+(3,4) \Downarrow 7} \quad \dfrac{}{1 \Downarrow 1}}{-(+(3,4),1) \Downarrow 6}$$

<u>Example 2</u>: Evaluation of expressions, e.g. $+(3,4)$

↳ Set = pairs of AST & value

Notation: $E \Downarrow n$ .. expr. E evaluates to number n

Axioms: $\underbrace{1 \Downarrow 1, \quad 2 \Downarrow 2, \quad \text{etc.}}_{\text{axiom scheme: } n \Downarrow n}$.

Rules: $\underbrace{\dfrac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{+(E_1, E_2) \Downarrow n}}_{\text{rule scheme:}}$ if $n = n_1 + n_2$, etc.

$$\dfrac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{Op(E_1, E_2) \Downarrow n} \quad \text{if } n = n_1 \, Op \, n_2$$

Plan for Today

- **Motivation & preliminaries**
- **Abstract syntax of SIMP** ←
- **An abstract machine for SIMP**
- **Structural operation semantics for SIMP**
  - Small-step semantics
  - Big-step semantics

Operational Semantics of an Imperative Language

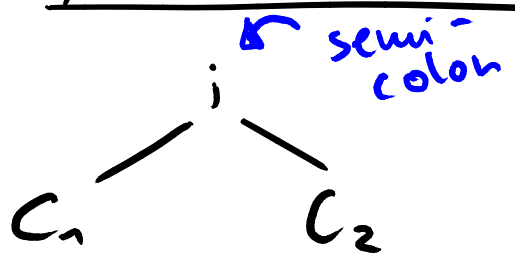
1) Abstract syntax of SIMP

- SIMP = simple imperative PL
- features: assignments, sequencing, conditionals, loops, integer variables

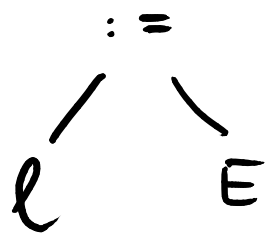
- abstract syntax:

$$P ::= C \mid E \mid B$$

a) Commands:

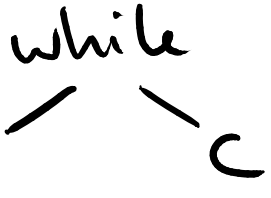


(sequence)



(assignment of expression to label)

(two-way selector)



(while loop)

skip

(do nothing)

Textual notation: $C ::= C ; C \mid \ell := E \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid \text{skip}$

b) <u>Integer expressions</u>

$E ::= \,!\ell \mid n \mid E \text{ op } E$

$\text{op} ::= + \mid - \mid * \mid /$

where n ... integer

$\ell \in L = \{\ell_0, \ell_1, \dots\}$ .. memory locations

$!\ell$ .. value stored at location ℓ

c) <u>Boolean expressions</u>

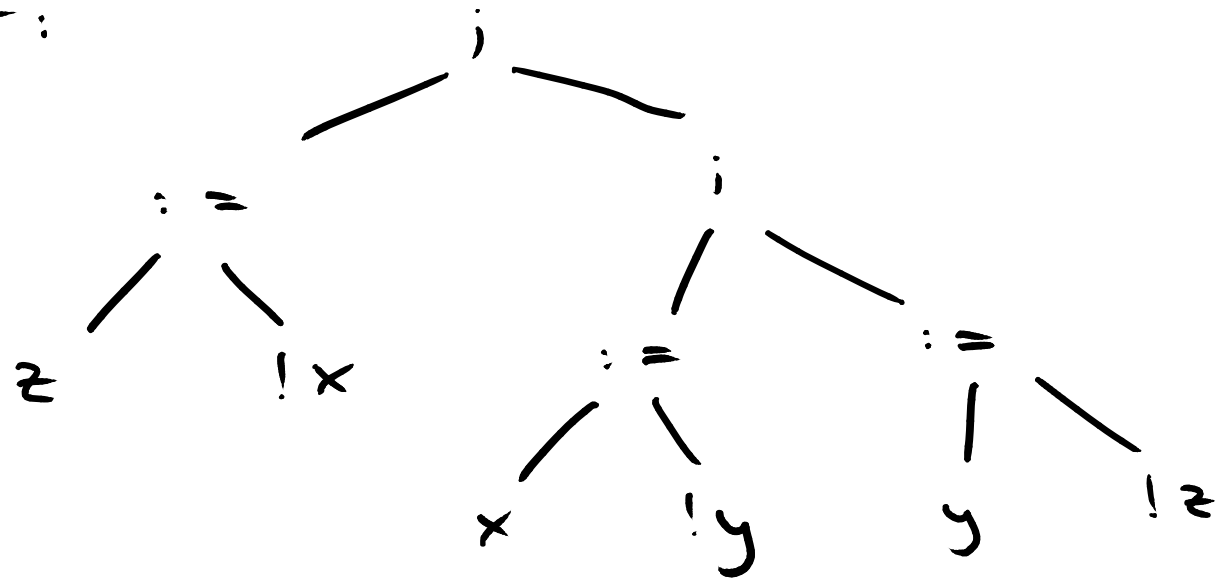
$B ::= \text{True} \mid \text{False} \mid E \text{ bop } E \mid \neg B \mid B \wedge B$

$\text{bop} ::= > \mid < \mid =$

<u>Example 1</u> $z := !x \; ; \; (x := !y \; ; \; y := !z)$

... swap values in x and y

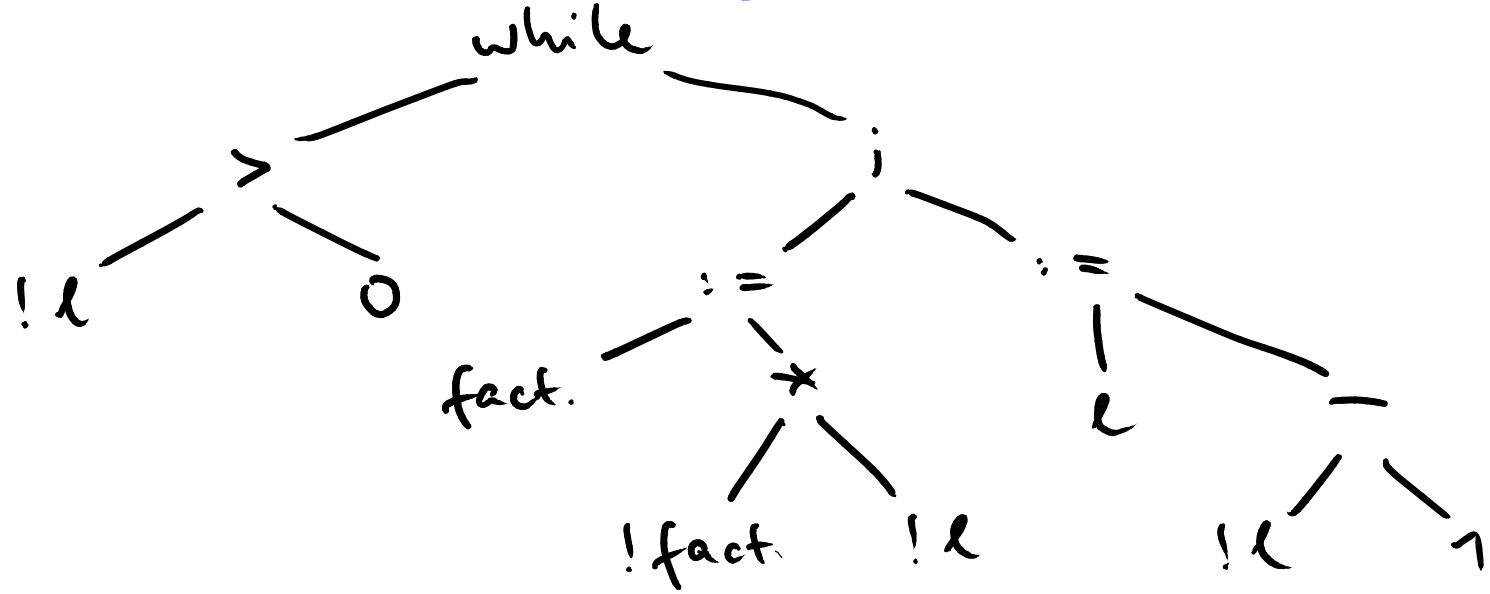
AST:



Example 2

```
while ( !l > 0 ) do (
    factorial := !factorial * !l ;
    l := !l - 1 )
```

AST: Quiz: # nodes? 15 # edges? 14



Plan for Today

- **Motivation & preliminaries**
- **Abstract syntax of SIMP**
- **An abstract machine for SIMP** ←
- **Structural operation semantics for SIMP**
  - Small-step semantics
  - Big-step semantics

2) An Abstract Machine for SIMP

4 main elements:

- control stack c : stores instructions to execute
- auxiliary / results stack r: stores intermediate results
- processor: performs arithmetic ops., comparisons, and boolean ops.
- memory / store m: partial function mapping locations to integers

↳ Notation: $m[l \mapsto n]$ .. update fct. m with new mapping $l \mapsto n$

↳ i.e. $m[l \mapsto n](l) = n$

$m[l \mapsto n](l') = m(l') \quad \forall l'.\ l' \neq l$

Abstract machine = transition system

Configuration : $\langle c, r, m \rangle$

$c ::= nil \mid i \circ c$ ← instr. i pushed on top of c

↳ empty stack

$i ::= P \mid op \mid \neg \mid \wedge \mid bop \mid := \mid if \mid while$

$r ::= nil \mid P \circ r \mid l \circ r$

Modul execution of programs as sequences of transitions from initial to final configurations

$\langle C \circ nil, nil, m \rangle$ $\langle nil, nil, m \rangle$

Execute C in a given memory state

Stop when all stacks are empty

Transition rules: $\rightarrow$

$$\langle c, r, m \rangle \longrightarrow \langle c', r', m' \rangle$$

a) Evaluating expressions

$\langle n \circ c, r, m \rangle \longrightarrow \langle c, n \circ r, m \rangle$

↳ pop n from control stack & push it onto r

$\langle !\ell \circ c, r, m \rangle \longrightarrow \langle c, n \circ r, m \rangle$ if $m(\ell) = n$

↳ read memory at ℓ & push result on r

$\langle (E_1 \text{ op } E_2) \circ c, r, m \rangle \longrightarrow \langle E_1 \circ E_2 \circ \text{op} \circ c, r, m \rangle$

$\langle \text{op} \circ c, n_2 \circ n_1 \circ r, m \rangle \longrightarrow \langle c, n \circ r, m \rangle$

if $n_1 \text{ op } n_2 = n$

(similar for Boolean expr.)