

CS 252:

*Advanced Programming Language Principles*



# Operational Semantics, Continued

Prof. Tom Austin

San José State University

# Topics covered today

- Small-step operational semantics
- Representing mutable state
- Evaluation contexts

# Review: Bool\* Language

$e ::=$

true

| false

| if e

then e

else e

*expressions:*

constant true

constant false

conditional

# Big-step operational semantics evaluate every expression to a value.

The expression  $e$  ...

$e$



$v$

... evaluates to ...

... the value  $v$ .



# Problems with Big-Step Op. Sem.

Big-step operational semantics closely match the "natural" way to write an interpreter.

However, they can't reason about:

- Programs that crash
- Programs that run forever
- Certain properties of type systems
- Many other interesting properties

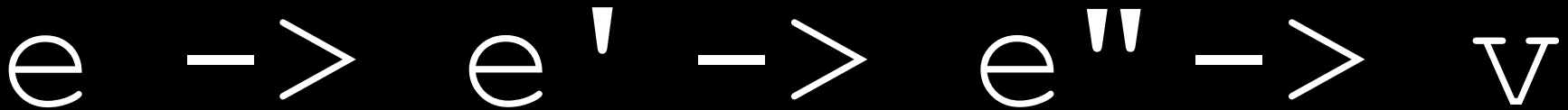
*Small-step* operational semantics  
evaluate an expression until it is in  
*normal form*.



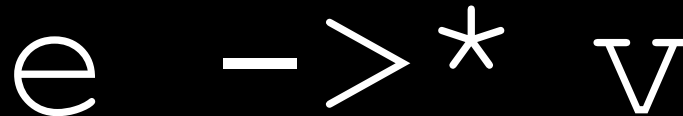
"normal form" – it  
cannot be evaluated  
further.

# Small-Step Evaluation Relation

One step in the  
evaluation



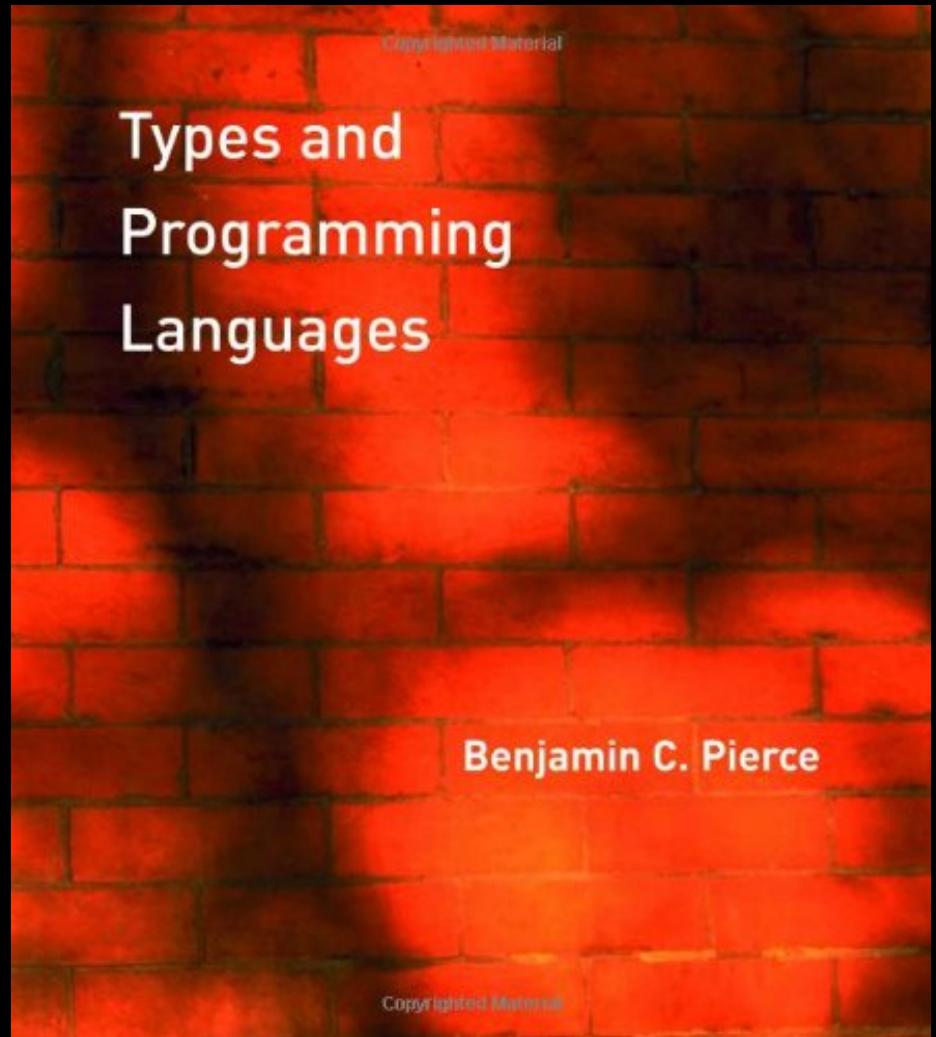
Many steps in  
the evaluation



# TAPL

*The top reference for  
more details on PL  
formalisms.*

*Available at library.*



# Review: Big-step semantics for Bool\*

**B-IfTrue**

$$\frac{e1 \Downarrow \text{true} \quad e2 \Downarrow v}{\text{if } e1 \text{ then } e2 \text{ else } e3 \Downarrow v}$$

**B-IfFalse**

$$\frac{e1 \Downarrow \text{false} \quad e3 \Downarrow v}{\text{if } e1 \text{ then } e2 \text{ else } e3 \Downarrow v}$$

**B-Value**

$$\frac{}{v \Downarrow v}$$

# Small-step semantics for Bool\*

(in-class)

# Bool\* Small-Step Semantics

**No rule  
needed for  
values!**

## E-IfTrue

```
if true then e2 else e3 -> e2
```

## E-IfFalse

```
if false then e2 else e3 -> e3
```

## E-If

$$\frac{e1 \rightarrow e1'}{\text{if } e1 \text{ then } e2 \text{ else } e3 \rightarrow \text{if } e1' \text{ then } e2 \text{ else } e3}$$

Let's develop operational semantics for the WHILE language.

Unlike Bool\*, WHILE supports *mutable references*.

# WHILE Language

<code>e ::= a</code>	variables/addresses
<code>v</code>	values
<code>a := e</code>	assignment
<code>e ; e</code>	sequence
<code>e op e</code>	binary operations
<code>if e then e</code>	conditionals
<code>    else e</code>	
<code>while (e) e</code>	while loops

# WHILE Language (continued)

$v ::= i$  integers  
|  $b$  booleans

$op ::= +$  |  $-$  binary operators  
|  $\backslash$  |  $*$   
|  $<$  |  $>$   
|  $<=$  |  $>=$

# Bool\* vs. WHILE evaluation relation

Bool\* relation:

$$e \rightarrow e'$$

WHILE relation:

$$e, \sigma \rightarrow e', \sigma'$$

A "store", represented by the Greek letter sigma

# Important Note on The Store

- We can represent state in big-step operational semantics with the same notation.
- Small-step relation for WHILE language:  
 $e, \sigma \rightarrow e', \sigma'$
- Big-step relation for WHILE language:  
 $e, \sigma \Downarrow v, \sigma'$

# The Store

- Maps *references* to values
- Some key operations:
  - $\sigma (a)$  : Get value at "address"  $a$
  - $\sigma [a := v]$  : New store identical to  $\sigma$ , except that the value at address  $a$  is  $v$ .

*In-class:* Specify semantics for the WHILE language  $(e, \sigma \rightarrow e', \sigma')$

$e ::= a$	variables/addresses
$v$	values
$a := e$	assignment
$e ; e$	sequence
$e \text{ op } e$	binary operations
$\text{if } e \text{ then } e$	conditionals
$\quad \text{else } e$	
$\text{while } (e) \ e$	while loops

*Evaluation order rules* specify an order for evaluating expressions.

*Reduction rules* rewrite the expression.

**E-IfFalse** (reduction)

```
if false
  then e2
  else e3 -> e3
```

**E-If** (evaluation order)

```


$$\frac{e1 \rightarrow e1'}{\text{if } e1 \text{ then } e2 \text{ else } e3 \rightarrow \text{if } e1' \text{ then } e2 \text{ else } e3}$$

```

## Concise representation of evaluation order rules

- Evaluation order rules tend to
  - be repetitive
  - clutter the semantics
- Evaluation contexts represent the same information concisely

*A redex* (reducible expression) is an expression that can be transformed in one step

# Which expression is a redex?

1. `if true  
then (if true then false else false)  
else true`

The whole expression is  
a redex: a rule  
transforms "if true ..."

2. `if (if true then false else false)  
then false else true`

Condition needs to be  
evaluated first: not a redex

# Evaluation Contexts

- Replace evaluation order rules
- Marker (•) or "hole" indicates the next place for evaluation.
  - $C = \text{if } \bullet \text{ then true else false}$
  - $r = \text{if true then true else false}$
  - $C[r] = \text{if (if true then true else false) then true else false}$

The original expression

## Rewriting our evaluation rules

The rules now apply to a redex  
*within the specified context.*

**EC-IfFalse**

Note the  
addition of the  
C[...] to the rule

```
C[if false
  then e2
  else e3] -> C[e3]
```

## E-If (evaluation order)

$$\frac{e1 \rightarrow e1'}{\text{if } e1 \text{ then } e2 \text{ else } e3 \rightarrow \text{if } e1' \text{ then } e2 \text{ else } e3}$$

Context:

**C** ::= •  
| if **C** then e  
| else e  
| ...

Rewrite

## E-IfFalse (reduction)

if false  
  then e2  
  else e3  $\rightarrow$  e3

## EC-IfFalse

**C**[if false  
  then e2  
  else e3]  $\rightarrow$  **C**[e3]

*In class:* let's rewrite  
our evaluation rules in  
the new format.

## Homework #2: WHILE Interpreter

- Part 1: Rewrite the semantics for WHILE to use big-step semantics.
- Part 2: Write an interpreter for WHILE.  
Starter code is available on the course website

Haskell does not have mutable state.  
How can we write a program that does?

Introducing `Data.Map`...

## Data.Map

- Maps are **immutable**.
- Useful methods:
  - `empty`: creates a new, empty map
  - `insert k v m`: returns a new, updated map
  - `lookup k m`: returns the value for key `k` stored in map `m`, wrapped in a `Maybe` type
- See "Learn You a Haskell", Chapter 7