For next class, install LaTeX
(pronounced "LAH-tech")

CS 252:

Advanced Programming Language Principles

Operational Semantics, Continued

Prof. Tom Austin

San José State University

Review: Higher order functions lab

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 …

$$e \Downarrow v$$

… evaluates to …

… the value v.

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

```
e -> e' -> e" -> v
```

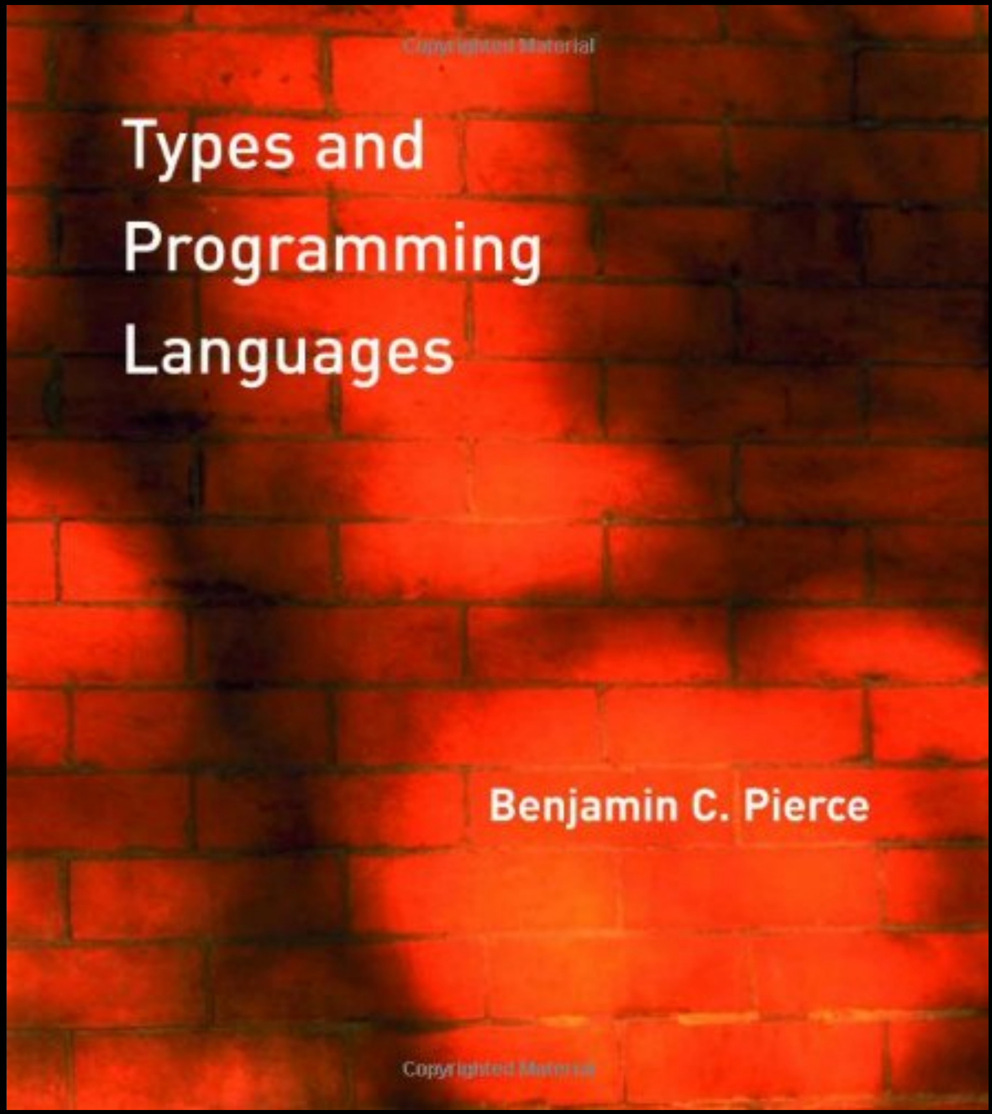
Many steps in the evaluation

```
e ->* v
```

TAPL

The top reference for more details on PL formalisms.

Available at library.



Review: Big-step semantics for Bool*

B-IfTrue

$$\frac{\texttt{e1} \Downarrow \texttt{true} \qquad \texttt{e2} \Downarrow \texttt{v}}{\texttt{if e1 then e2 else e3} \Downarrow \texttt{v}}$$

B-IfFalse

$$\frac{\texttt{e1} \Downarrow \texttt{false} \qquad \texttt{e3} \Downarrow \texttt{v}}{\texttt{if e1 then e2 else e3} \Downarrow \texttt{v}}$$

B-Value

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

Small-step semantics for Bool*

(in-class)

Bool* Small-Step Semantics

E-IfTrue

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

E-IfFalse

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

E-If

$$\frac{\texttt{e1 -> e1'}}{\texttt{if e1 then e2 else e3 -> if e1' then e2 else e3}}$$

Let's develop operational semantics for the WHILE language.

Unlike Bool*, WHILE supports *mutable references*.

WHILE Language

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

WHILE Language (continued)

`v ::= i`	integers
`    | b`	booleans
`op ::= +  | -`	binary operators
`     | \  | *`	
`     | <  | >`	
`     | <= | >=`	

Small-step semantics with state

Since Bool* did not have mutable references, our evaluation rules only handled expressions:

E-IfFalse

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

WHILE *does* allow for imperative updates, so we need to modify our semantics.

Bool* vs. WHILE evaluation relation

<u>Bool* relation:</u>

```
e -> e'
```

<u>WHILE relation:</u>

$$e,\sigma \;\texttt{->}\; e',\sigma'$$

A "store", represented by the Greek letter sigma

The Store

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

In-class: Specify semantics for the WHILE language (`e,σ -> e',σ'`)

```
e ::= a                    variables/addresses
    | v                    values
    | a:=e                 assignment
    | e;e                  sequence
    | e op e               binary operations
    | if e then e          conditionals
           else e
    | 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)

$$\frac{\texttt{e1 -> e1'}}{\texttt{if e1 then e2 else e3 -> if e1' then e2 else e3}}$$

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?

This is a redex: a rule transforms "if true …"

```
1. if true
   then(if true then false else
   false) else 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 = `if • then true else false`
  - r = `if true then true else false`
  - C[r] = `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*.

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

EC-IfFalse

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

E-If (evaluation order)

```
   e1 -> e1'
---------------
if e1 then e2
  else e3 ->
if e1' then
  e2 else e3
```

E-IfFalse (reduction)

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

Rewrite

Context:

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

EC-IfFalse

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

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

Homework #2: WHILE Interpreter

- http://www.cs.sjsu.edu/~austin/cs252-fall17/hw/hw2/while-semantics.pdf specifies details.
- Part 1: Rewrite the semantics for WHILE without contexts.
- Part 2: Write an interpreter for WHILE. Starter code is available at http://www.cs.sjsu.edu/~austin/cs252-fall17/hw/hw2/

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