Attendance question

Are these two the same functions?

```
fun x -> x
fun y -> y
```

A. Yes
B. No
Review

Previously in 3110: simple interpreter
• abstract syntax tree (AST)
• evaluation based on single steps

Today:
• Formal syntax: BNF
• Formal dynamic semantics:
  small-step, substitution model
• Formal static semantics
FORMAL SYNTAX
e ::= x
| i
| e1 + e2
| let x = e1 in e2

Backus-Naur Form (BNF)
John Backus (1924-2007)
ACM Turing Award Winner 1977
“For profound, influential, and lasting contributions to the design of practical high-level programming systems”

Peter Naur (1928-2016)
ACM Turing Award Winner 2005
“For fundamental contributions to programming language design”
BNF

Note resemblance:

e ::= x | i | e1 + e2
    | let x = e1 in e2

\textbf{type} expr =
\hspace{1em} Var of string
\hspace{1em} Int of int
\hspace{1em} Add of expr * expr
\hspace{1em} Let of string * expr * expr
e → e' 

single-step relation
values never step
multi-step relation
Question

Which of these is true?

A. \((5+2)+0\) \(\rightarrow\ast\) \((5+2)+0\)
B. \((5+2)+0\) \(\rightarrow\ast\) \(7+0\)
C. \((5+2)+0\) \(\rightarrow\ast\) \(7\)
D. All of the above
e1 + e2 \rightarrow e1' + e2
   if e1 \rightarrow e1'

v1 + e2 \rightarrow v1 + e2'
   if e2 \rightarrow e2'

v1 + v2 \rightarrow i
   if i is the result of primitive operation v1+v2
let x = e1 in e2

--> let x = e1' in e2

  if e1 --> e1'

let x = v1 in e2 --> e2{v1/x}
Booleans

e ::= x | i | b
   | e1 + e2 | e1 && e2
   | let x = e1 in e2
   | if e1 then e2 else e3

v ::= i | b
Evaluation models

Small-step substitution model:
• Substitute value for variable
• Good mental model for evaluation
• Inefficient: too much work at run time
• Not really what OCaml does

Big-step environment model:
• Maintain data structure binding variables to values
• At the heart of what OCaml really does
• (next lecture)
FORMAL STATIC SEMANTICS
Static semantics

We can have nonsensical expressions:

\[ 5 + \text{false} \]
\[ \text{if } 5 \text{ then } \text{true else } 0 \]

Need to rule those out...
if expressions [from lec 2]

Syntax:

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

Type checking:

if \( e_1 \) has type \texttt{bool} and \( e_2 \) has type \( t \) and \( e_3 \) has type \( t \)
then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) has type \( t \)
Static semantics

Defined as a ternary relation:

\[ \mathcal{T} \vdash e : t \]

- Read as in typing context \( \mathcal{T} \), expression \( e \) has type \( t \)
- Turnstile \( \vdash \) can be read as "proves" or "shows"
- You're already used to \( e : t \), because utop uses that notation
- *Typing context* is a dictionary mapping variable names to types
Static semantics

e.g.,

\[ x : \text{int} \vdash x + 2 : \text{int} \]

\[ x : \text{int}, y : \text{int} \vdash x < y : \text{bool} \]

\[ \vdash 5 + 2 : \text{int} \]
Purpose of type system

Ensure **type safety**: well-typed programs don't get *stuck*:
• haven't reached a value, and
• unable to evaluate further

Lemmas:
**Progress:** if $e : t$, then either $e$ is a value or $e$ can take a step.
**Preservation:** if $e : t$, and if $e$ takes a step to $e'$, then $e' : t$.

Type safety = progress + preservation

Proving type safety is a fun part of CS 4110
Upcoming events

• N/A

This is not a substitute.

THIS IS 3110