Review

Previously in 3110: simple interpreter for expression language
• abstract syntax tree (AST)
• evaluation based on single steps
• parser and lexer (in lab)

Today:
• Formal syntax: BNF
• Formal dynamic semantics: small-step, substitution model
• Formal static semantics
Substitution

(* [subst e v x] is e{v/x}, that is, * [e] with [v] substituted for [x]. *)

```
let rec subst e v x = match e with
| Var y -> if x=y then v else e
| Int n -> Int n
| Add(el,er) ->
  Add(subst el v x, subst er v x)
| Let(y,ebind,ebody) ->
  let ebind' = subst ebind v x in
  if x=y
  then Let(y, ebind', ebody)
  else Let(y, ebind', subst ebody v x)
```
let rec step = function
    | Int n -> failwith "Does not step"
    | Add(Int n1, Int n2) -> Int (n1 + n2)
    | Add(Int n1, e2) -> Add (Int n1, step e2)
    | Add(e1, e2) -> Add (step e1, e2)
    | Var _ -> failwith "Unbound variable"
    | Let(x, Int n, e2) -> subst e2 (Int n) x
    | Let(x, e1, e2) -> Let (x, step e1, e2)
FORMAL SYNTAX
Abstract syntax of expression lang.

\[
e ::= x \mid i \mid e1+e2 \\
    \mid \text{let } x = e1 \text{ in } e2
\]

e, x, i: \textit{meta-variables} that stand for pieces of syntax
- e: expressions
- x: program variables, aka identifiers
- i: integer constants, aka literals

\::=\text{ and } \mid \text{ are } \textit{meta-syntax}: \text{used to describe syntax of language}

notation is called \textit{Backus-Naur Form} (BNF) from its use by Backus and Naur in their definition of Algol-60
Backus and Naur

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”
Abstract syntax of expr. lang.

\[ e ::= x \mid i \mid e_1+e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]

Note resemblance of BNF to AST type:

```plaintext
type expr =
    | Var of string
    | Int of int
    | Add of expr * expr
    | Let of string * expr * expr
```
Extended with Booleans

\[ e ::= x \mid i \mid b \]
\[ \mid e_1 + e_2 \mid e_1 && e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]
\[ \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ v ::= i \mid b \]
FORMAL DYNAMIC SEMANTICS
Dynamic semantics

Defined as a binary relation:

\[ e \rightarrow e' \]

Read as \( e \) takes a single step to \( e' \)

E.g., \((5+2) + 0 \rightarrow 7+0\)

Expressions continue to step until they reach a value
E.g., \((5+2) + 0 \rightarrow 7+0 \rightarrow 7\)

Values are a syntactic subset of expressions:

\[ v ::= i \mid b \]
Dynamic semantics

Reflexive transitive closure of $\rightarrow$ is written $\rightarrow^*$

$e \rightarrow^* e'$ read as $e$ multisteps to $e'$

e.g.,

\[
(5+2) + 0 \rightarrow^* (5+2) + 0 \\
(5+2) + 0 \rightarrow^* 7 + 0 \\
(5+2) + 0 \rightarrow^* 7
\]

This style of definition is called a *small-step semantics*: based on taking single small steps
Dynamic semantics of expr. lang.

\[ e_1 + e_2 \rightarrow e_1' + e_2 \]
\[ \text{if } e_1 \rightarrow e_1' \]

\[ v_1 + e_2 \rightarrow v_1 + e_2' \]
\[ \text{if } e_2 \rightarrow e_2' \]

\[ v_1 + v_2 \rightarrow i \]
\[ \text{if } i \text{ is the result of primitive operation } v_1 + v_2 \]
Dynamic semantics of expr. lang.

`let x = e1 in e2 --> let x = e1' in e2`  
  `if e1 --> e1'`

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

recall: read `e2{v1/x}` as `e2` with `v1` substituted for `x`  
(as we defined last lecture and implemented in `subst`)

so we call this the substitution model of evaluation
Evaluation models

• Small-step substitution model:
  – Substitute value for variable in body of \texttt{let} expression
  – And in body of function, since \texttt{let x = e1 in e2}
    behaves the same as \texttt{(fun x -> e2) e1}
  – Inefficient implementation: have to do too much substitution
    at run time
  – Not really what OCaml does
  – Good mental model for evaluation

• Big-step environment model:
  – Keep a data structure that binds variables to values
  – At the heart of what OCaml really does
Dynamic semantics of expr. lang.

\[
\begin{align*}
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
\quad \rightarrow \ 	ext{if } e_1' \text{ then } e_2 \text{ else } e_3 \\
\quad \quad \ 	ext{if } e_1 \ ightarrow \ e_1'
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{true} \text{ then } e_2 \text{ else } e_3 \ ightarrow \ e_2
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{false} \text{ then } e_2 \text{ else } e_3 \ ightarrow \ e_3
\end{align*}
\]
Dynamic semantics of expr. lang.

Values and variables do not single step:
\[ v \rightarrow \rightarrow \] (values are already done evaluating)
\[ x \rightarrow \rightarrow \] (should have been substituted away)

But they do multistep (because they can take 0 steps):
\[ v \rightarrow \rightarrow^* v \]
\[ x \rightarrow \rightarrow^* x \]
Scaling up to OCaml

Read notes on website: full dynamic semantics for core OCaml:

\[
e ::= x \mid e_1 \ e_2 \mid \text{fun} \ x \to e \\
    \mid i \mid e_1 + e_2 \\
    \mid (e_1, \ e_2) \mid \text{fst} \ e_1 \mid \text{snd} \ e_2 \\
    \mid \text{Left} \ e \mid \text{Right} \ e \\
    \mid \text{match} \ e \ \text{with} \ \text{Left} \ x \to e_1 \mid \text{Right} \ y \to e_2 \\
    \mid \text{let} \ x = e_1 \ \text{in} \ e_2
\]

**Missing:** other built-in types, records, lists, options, declarations, patterns in function arguments and let bindings, if expressions, recursion
FORMAL STATIC SEMANTICS
Static semantics

We can have nonsensical expressions:

5 + false

if 5 then true else 0

Need *static semantics* (type checking) to rule those out...
if expressions [from lec 2]

Syntax:

    if e1 then e2 else e3

Type checking:

    if e1 has type bool and e2 has type t and e3 has type t
    then if e1 then e2 else e3 has type t
Static semantics

Defined as a ternary relation:

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

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

e.g.,

\[ x: \text{int} \mid - x + 2 : \text{int} \]
\[ x: \text{int}, y: \text{int} \mid - x < y : \text{bool} \]
\[ \mid - 5 + 2 : \text{int} \]
Static semantics of expr. lang.

\[ T \vdash i : \text{int} \]

\[ T \vdash b : \text{bool} \]
Static semantics of expr. lang.

\[ T \vdash e_1 + e_2 : \text{int} \]
\[ \text{if } T \vdash e_1 : \text{int} \]
\[ \text{and } T \vdash e_2 : \text{int} \]

\[ T \vdash e_1 && e_2 : \text{bool} \]
\[ \text{if } T \vdash e_1 : \text{bool} \]
\[ \text{and } T \vdash e_2 : \text{bool} \]
Static semantics of expr. lang.

\[ T \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]
\[ \text{if } T \vdash e_1 : \text{bool} \]
\[ \text{and } T \vdash e_2 : t \]
\[ \text{and } T \vdash e_3 : t \]

\[ T \vdash \text{let } x : t_1 = e_1 \text{ in } e_2 : t_2 \]
\[ \text{if } T \vdash e_1 : t_1 \]
\[ \text{and } T, x : t_1 \vdash e_2 : t_2 \]

\[ T, x : t \vdash x : t \]

To avoid need for type inference, require type annotation here.
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
Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.

—Anonymous
Interpreter for expr. lang.

See *interp3.ml* in code for this lecture

1. Type-checks expression, then
2. Evaluates expression
Upcoming events

• [Wednesday] A3 due
• [Thursday] A4 out