Today’s music: “Down to Earth” by Peter Gabriel from the WALL-E soundtrack
Review

Previously in 3110:
• simple interpreter for expression language:
  – abstract syntax tree (AST)
  – small-step, substitution model of evaluation
  – parser and lexer

Today:
• Formal syntax: BNF
• Formal dynamic semantics:
  – small-step, substitution model
  – large-step, environment model
• Formal static semantics
FORMAL SYNTAX
Notation

• The code we've written is one way of defining the syntax and semantics of a language

• Programming language designers have another more compact notation that's independent of the implementation language of interpreter...
Abstract syntax of expression lang.

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

\[ e, x, i: \text{meta-variables} \] that stand for pieces of syntax

- \( e \): expressions
- \( x \): program variables
- \( i \): integers

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

notation is called 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 | i | e + e
    | let x = e1 in e2

Note how closely the BNF resembles the OCaml variant we used to represent it!
FORMAL DYNAMIC SEMANTICS
Language we'll use for now

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

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

Defined by a judgement:
\[ e \rightarrow e' \]
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.

\[ e_1 + e_2 \rightarrow e_1' + e_2 \]
if \( e_1 \rightarrow e_1' \)

\[ v_1 + e_2 \rightarrow v_1 + e_2' \]
if \( e_2 \rightarrow e_2' \)

\[ v_1 + v_2 \rightarrow n \]
if \( n \) is the result of primitive operation \( v_1 + v_2 \)
Dynamic semantics of expr. lang.

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2 \\
\quad \text{if } e_1 \rightarrow e_1'
\]

\[
\text{let } x = v_1 \text{ in } e_2 \rightarrow e_2\{v_1/x\}
\]

recall: read \(e_2\{v_1/x\}\) as \(e_2\) with \(v_1\) substituted for \(x\) (as we defined last lecture and implemented in \text{subst})

so we call this the substitution model of evaluation
Dynamic semantics of expr. lang.

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

\[
\text{if true then } e_2 \text{ else } e_3 \implies e_2
\]

\[
\text{if false then } e_2 \text{ else } e_3 \implies e_3
\]
Dynamic semantics of expr. lang.

Values and variables do not single step:

\[ v \rightarrow v \]
\[ x \rightarrow x \]

- Values don't step because they're done computing
- Variables don't step we should never reach a variable; it should have already been substituted away

But they do multistep (because in 0 steps they are themselves):

\[ v \rightarrow^{*} v \]
\[ x \rightarrow^{*} x \]
Scaling up to OCaml

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

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

**Missing, unimportant:** other built-in types, records, lists, options, declarations, patterns in function arguments and let bindings, if

**Missing, important:** let rec
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 by a judgement:

\[ T \vdash e : t \]

• Read as in typing context \( T \), 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.,
\[ x: \text{int} \vdash x + 2 : \text{int} \]
\[ x: \text{int}, y: \text{int} \vdash x < y : \text{bool} \]
\[ \vdash 5 + 2 : \text{int} \]
Static semantics of ext. expr. lang.

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

\[ T \models b : \text{bool} \]

\[ T, x : t \models x : t \]
Static semantics of ext. expr. lang.

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

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

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

\[ T \vdash \text{let } x : t_1 = e_1 \text{ in } e_2 : t_2 \]
\[ \text{if } T \vdash e_1 : t_1 \]
\[ \text{and } T, x : t_1 \vdash e_2 : t_2 \]
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
Interpreter for ext. expr. lang.

See `interp3.ml` in code for this lecture

1. Type-checks expression, then
2. Evaluates expression
Preview of next lecture

Today we saw:

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

Next time we'll see:

• **Big-step environment model:** keep a data structure around that binds variables to values
  – Also a good mental model
  – At the heart of what OCaml really does
Upcoming events

• [ASAP] Prelim1 grades out
• [Friday] MS0 due, no late submissions

This is not just semantics.

THIS IS 3110