Computer Science 131
Programming Languages

September 12, 2000
Static and Dynamic Semantics
Syntax vs. Semantics

- **Syntax**
  - What phrases may occur where.

- **Semantics:**
  - What the phrases mean when put together.
  - What should the answer be?
  - How should execution proceed?
Purposes of a Language Definition

• For the programmer
  – Understanding the language
  – Reasoning about programs

• For the language implementor
  – Understanding what correct implementations must/may do
  – Deciding whether program transformations are correct
  – Facilitate multiple (compatible) implementations

• For the language designer
  – Recording design decisions
  – Understanding interaction between language features
  – Reasoning about the language
Formal Definitions?

• Why a formal semantics?
  - Informal definitions invariably contain ambiguities or errors.
  - Facilitates reasoning about the language
  - Facilitates reasoning about programs in the language
  - Facilitates reasoning about program transformations
  - May permit automatic generation of implementations

• Truth in advertising: still very hard to give a formal description of a full, real language
  - But can handle quite large subsets
  - Active research topic
Simple Arithmetic Expressions

• Abstract Syntax

\[
\begin{align*}
  v & ::= n \quad \text{(values)} \\
  e & ::= v \mid e + e \quad \text{(expressions)} \\
  p & ::= e \quad \text{(programs)}
\end{align*}
\]

• What does a program in this language mean?
Two Approaches to Formal Semantics

• Denotational semantics
  - The meaning of every expression is a mathematical object (a number, a function, etc.)
  - Compositionality: meaning of an expression is a function of the meanings of its sub-expressions.
    • E.g., the meaning of a while loop
      \[
      \text{while } b \text{ do } e
      \]
      is calculated from the meanings of the guard expression \( b \) and of the loop body \( e \)
  - Two expressions with the same meaning are interchangeable.
Two Approaches to Formal Semantics

• Operational semantics
  - Defines evaluation of (complete) programs
  - High-level specification of an interpreter
  - We can choose the level of abstraction
    • Which (if any) low-level machine details we want to describe
      - Data representations
      - Memory management
  • Which concepts considered primitive
Small-step Operational Semantics

- Defines a relation $\rightarrow$ between programs corresponding to one step of execution

$$(3 + 4) + (5 + 3) \rightarrow 7 + (5 + 3)$$

$$7 + (5 + 3) \rightarrow 7 + 8$$

$$7 + 8 \rightarrow 15$$
Small-step Operational Semantics

• We define the relation $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$.

  $e \rightarrow^* e'$ if $e \rightarrow e'$.

  $e \rightarrow^* e$ always.

  $e \rightarrow^* e''$ if $e \rightarrow^* e'$ and $e' \rightarrow^* e''$ for some $e'$. 
Small-step Operational Semantics

- We define the relation $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$.
  
  $e \rightarrow^* e'$ if $e \rightarrow e'$.
  $e \rightarrow^* e$ always.
  $e \rightarrow^* e''$ if $e \rightarrow^* e'$ and $e' \rightarrow^* e''$ for some $e'$.

- For example, $(3+4)-(5-3) \rightarrow^* 5$

- A program $p$ terminates with value $v$ if $p \rightarrow^* v$

- A program $p$ fails to terminate if $p \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow ...$
Small-step Operational Semantics

• Equivalent definition: inference rules (hopefully familiar from CS 80)

\[
\begin{align*}
\frac{e \rightarrow e'}{e \rightarrow^* e'} & \quad \frac{e \rightarrow^* e}{e \rightarrow^* e} \\
\frac{e \rightarrow^* e'}{e' \rightarrow^* e'} & \quad \frac{e' \rightarrow^* e''}{e \rightarrow^* e''}
\end{align*}
\]
Structured Operational Semantics

\[ n_1 + n_2 \rightarrow n_1 + n_2 \]

\[ e_1 \rightarrow e_1' \]
\[ e_1 + e_2 \rightarrow e_1' + e_2 \]

\[ e_2 \rightarrow e_2' \]
\[ e_1 + e_2 \rightarrow e_1 + e_2' \]
Concurrency and Non-determinism?

\[
\begin{align*}
&n_1 + n_2 \rightarrow n_1 + n_2 \\
&e_1 \rightarrow e_1' \\
&e_1 + e_2 \rightarrow e_1' + e_2 \\
&e_2 \rightarrow e_2' \\
&e_1 + e_2 \rightarrow e_1 + e_2'
\end{align*}
\]

\[
(3 + 4) + (5 + 3) \rightarrow 7 + (5 + 3) \\
(3 + 4) + (5 + 3) \rightarrow (3 + 4) + 8
\]
Left-to-Right Evaluation

\[ n_1 + n_2 \rightarrow n_1 + n_2 \]

\[ e_1 \rightarrow e_1' \]
\[ e_1 + e_2 \rightarrow e_1' + e_2 \]

\[ e_2 \rightarrow e_2' \]
\[ e_1 + e_2 \rightarrow e_1 + e_2' \]

\[ e_2 \rightarrow e_2' \]
\[ v_1 + e_2 \rightarrow v_1 + e_2' \]
Adding Booleans

• Abstract Syntax

\[\begin{align*}
v & ::= n \mid \text{tt} \mid \text{ff} & \quad \text{(values)} \\
e & ::= v \mid e + e \mid e < e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{if } e \text{ then } e \text{ else } e & \quad \text{(expressions)} \\
p & ::= e & \quad \text{(programs)}
\end{align*}\]

• For example,

\[\text{if (if 3 < 5 then ff else tt) then 1+2 else 7+8}\]
Dynamic Semantics

\[ n_1 + n_2 \rightarrow n_1 + n_2 \]

\[ e_1 \rightarrow e_1' \]
\[ e_1 + e_2 \rightarrow e_1' + e_2 \]

\[ e_2 \rightarrow e_2' \]
\[ v_1 + e_2 \rightarrow v_1 + e_2' \]

\[ n_1 < n_2 \rightarrow n_1 < n_2 \]

\[ e_1 \rightarrow e_1' \]
\[ e_1 < e_2 \rightarrow e_1' < e_2 \]

\[ e_2 \rightarrow e_2' \]
\[ v_1 < e_2 \rightarrow v_1 < e_2' \]
Dynamic Semantics

\[ e_1 \rightarrow e_1' \]

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

What other rules?
Dynamic Semantics

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

\[
\begin{align*}
  \text{if } \text{tt} \text{ then } e_2 \text{ else } e_3 \rightarrow e_2
\end{align*}
\]

\[
\begin{align*}
  \text{if } \text{ff} \text{ then } e_2 \text{ else } e_3 \rightarrow e_3
\end{align*}
\]
Stuck programs

• We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!

\[3 + \text{tt} \]
\[\text{if tt < ff then 3 else 5}\]
Stuck programs

- We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!
  
  \[ 3 + \texttt{tt} \]
  
  \[ \text{if } \texttt{tt} < \texttt{ff} \text{ then } 3 \text{ else } 5 \]

- Answer 1: who cares?
  - Implementation-dependent behavior
  - Or, program should stop with error.
Stuck programs

• We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!

  \[ 3 + \text{tt} \]
  \[ \text{if } \text{tt} < \text{ff} \text{ then } 3 \text{ else } 5 \]

• Answer 1: who cares?
  - Implementation-dependent behavior
  - Or, program should stop with error.

• Answer 2: a type system to prevent this
Typing

• We define a collection of types. For now, just
  \[ t ::= \text{Int} \mid \text{Bool} \]

• We define the relation \( e : t \)
  - A collection of inference rules is used to define when this relation holds.

• The type system is frequently called the "static semantics" of the language.
Static Semantic Rules

\( n : \text{Int} \quad tt : \text{Bool} \quad ff : \text{Bool} \)
Static Semantic Rules

\[ n : \text{Int} \quad \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \]

\[ \text{???
}\]

\[ e_1 + e_2 : \text{Int} \]
Static Semantic Rules

\[ n : \text{Int} \quad \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \]

\[ e_1 : \text{Int} \quad e_2 : \text{Int} \]

\[ e_1 + e_2 : \text{Int} \]
Static Semantic Rules

\[ n : \text{Int} \quad \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \]

\[ e_1 : \text{Int} \quad e_2 : \text{Int} \]

\[ e_1 + e_2 : \text{Int} \]

\[ e_1 : \text{Int} \quad e_2 : \text{Int} \]

\[ e_1 < e_2 : \text{Bool} \]
Static Semantic Rules

\[
\begin{align*}
\text{n : Int} & \quad \text{tt : Bool} & \quad \text{ff : Bool} \\
\hline
\text{e}_1 : \text{Int} & \quad \text{e}_2 : \text{Int} & \quad \text{e}_1 + e_2 : \text{Int} \\
\text{e}_1 : \text{Int} & \quad \text{e}_2 : \text{Int} & \quad \text{e}_1 < e_2 : \text{Bool} \\
\text{e}_1 : ? & \quad \text{e}_2 : ? & \quad \text{e}_3 : ? & \quad \text{if e}_1 \text{ then e}_2 \text{ else e}_3 : ?
\end{align*}
\]
Static Semantic Rules

\[
\begin{align*}
\text{n} & : \text{Int} \quad \text{tt} & : \text{Bool} \quad \text{ff} & : \text{Bool} \\
\hline
\text{e}_1 & : \text{Int} \quad \text{e}_2 & : \text{Int} \\
\end{align*}
\]

\[
\text{e}_1 + \text{e}_2 : \text{Int}
\]

\[
\text{e}_1 < \text{e}_2 : \text{Bool}
\]

\[
\begin{align*}
\text{e}_1 & : \text{Bool} \quad \text{e}_2 & : ? \quad \text{e}_3 & : ? \\
\text{if e}_1 \text{ then e}_2 \text{ else e}_3 & : ?
\end{align*}
\]
Static Semantic Rules

\( n : \text{Int} \quad \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \)

\( e_1 : \text{Int} \quad e_2 : \text{Int} \)

\[ e_1 + e_2 : \text{Int} \]

\( e_1 : \text{Int} \quad e_2 : \text{Int} \)

\[ e_1 < e_2 : \text{Bool} \]

\( e_1 : \text{Bool} \quad e_2 : t \quad e_3 : t \)

\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]
Claim: Type Soundness

- Well-typed programs (i.e., programs that have some type) can't get stuck
  - That is, must either eventually reach a value or fail to terminate

- Why is this important? How to prove this?
  - See next lecture
But...

• Some programs wouldn't get stuck but still don't typecheck
  \[(\text{if } \text{ff then } \text{tt} \text{ else } 4) + 1\]

• For any interesting language, a type system preventing all bad programs also rejects programs that would run without problems.

• Research topic: type systems that catch as many errors as possible, but don't reject useful programs
Adding Local Definitions

• Abstract Syntax

$$v ::= \text{n} \mid \text{tt} \mid \text{ff}$$  \hspace{1cm} (values)

$$e ::= v \mid e + e \mid e < e$$  \hspace{1cm} (expressions)

$$\mid \text{if e then e else e}$$

$$\mid x$$

$$\mid \text{let x be e in e}$$

$$p ::= e$$  \hspace{1cm} (programs)
Scoping For This Language

• In the expression `let x be e₁ in e₂`
  - The variable x is bound
  - The scope of x (where it can be referenced) is \( e₂ \)
Scoping For This Language

• All $\alpha$-equivalent expressions are identified
  - E.g., the following are the same expressions
    
    \[
    \text{let } x \text{ be } 3 \text{ in } x + y \\
    \text{let } z \text{ be } 3 \text{ in } z + y \\
    \]
  - But not
    
    \[
    \text{let } y \text{ be } 3 \text{ in } y + y \\
    \]
Changes to Dynamic Semantics

• Add the two rules:

\[
\begin{align*}
\text{let } x \text{ be } e_1 \text{ in } e_2 & \rightarrow \text{ let } x \text{ be } e_1' \text{ in } e_2 \\
\text{let } x \text{ be } v_1 \text{ in } e_2 & \rightarrow e_2[x \rightarrow v_1]
\end{align*}
\]
Changes to Dynamic Semantics

• What if we had just this single rule instead?

\[
\text{let } x \text{ be } e_1 \text{ in } e_2 \rightarrow e_2[x\rightarrow e_1]
\]

• Consider \(\text{let } x \text{ be } 1+2 \text{ in } x+x\)
Changes to the Static Semantics

• Typing is now context-sensitive
  - What is the type of \( x \)?
  - Is \( x+3 \) well-typed?
Changes to the Static Semantics

• Typing is now context-sensitive
  - What is the type of \( x \) ?
  - Is \( x + 3 \) well-typed?
    • Yes, in the program \( \text{let } x \text{ be } 4 \text{ in } x + 3 \)
    • No, in the program \( \text{let } x \text{ be } \texttt{tt} \text{ in } x + 3 \)

• Need to know the types of free variables
Changes to the Static Semantics

• 3-place typing relation  $\Gamma \vdash e : t$
  - Here $\Gamma$ is a type environment
  • Mapping from variables to their types
  - Type environments will be written as a list
  • e.g., $x: \text{Int}, y: \text{Bool}, z: \text{Int}$
  - The notation $\Gamma(x)$ gives the type of $x$ (lookup)
  - The notation $\Gamma, x: \text{Int}$
    is the extension of $\Gamma$ that maps $x$ to $\text{Int}$ (insert)
  - We write $\vdash e : t$ when type env. is empty
Static Semantics

\[ \Gamma \vdash n : \text{Int} \quad \Gamma \vdash \texttt{tt} : \text{Bool} \quad \Gamma \vdash \texttt{ff} : \text{Bool} \]
Static Semantics

\[
\begin{align*}
\Gamma \vdash n : \text{Int} & \quad \Gamma \vdash \text{tt} : \text{Bool} & \quad \Gamma \vdash \text{ff} : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
& \quad \Gamma \vdash e_1 + e_2 : \text{Int} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
& \quad \Gamma \vdash e_1 < e_2 : \text{Bool} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : t & \quad \Gamma \vdash e_3 : t \\
& \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]
Static Semantics:
The Interesting Rules

\[ \Gamma \vdash x : ? \]
Static Semantics:
The Interesting Rules

\[ \Gamma \vdash x : \Gamma(x) \]
Static Semantics:
The Interesting Rules

\[
\Gamma \vdash x : \Gamma(x)
\]

\[
???
\]

\[
???
\]

\[
\Gamma \vdash \text{let } x \text{ be } e_1 \text{ in } e_2 : t_2
\]
Static Semantics:
The Interesting Rules

\[ \Gamma \vdash x : \Gamma(x) \]

\[ \Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2 \]
\[ \Gamma \vdash \text{let } x \text{ be } e_1 \text{ in } e_2 : t_2 \]
Claim: Type Soundness

• If a program is well-typed in the empty type environment (i.e., \( \vdash p : t \) for some \( t \)) then it cannot get stuck

  - Note that such programs must have no free variables.
Adding Recursive Functions

• Abstract Syntax

\[ \begin{align*}
    v & ::= \mathit{n} \mid \mathit{tt} \mid \mathit{ff} \\ 
    & \mid \text{fun } x(x:t):t \text{ is } e \\
    e & ::= v \mid e + e \mid e < e \\ 
    & \mid \text{if } e \text{ then } e \text{ else } e \\ 
    & \mid x \mid \text{let } x \text{ be } e \text{ in } e \\ 
    & \mid e \ e \\
    p & ::= e
\end{align*} \]

(values) (expressions) (programs)
WARNING

• This is not SML!
• The expression \( \text{fun } f(x:t_1):t_2 = e \)
  - is a function value, like SML's \( \text{fn x => e} \) except that it can be recursive.
  - The equivalent SML code would be
    
    ```sml
    let
      fun f(x:t_1):t_2 = e
    in
      f
    end
    ```
Scoping

• In the expression \( \text{fun } f(x:t_1):t_2 \text{ is } e \)
  - Both \( f \) and \( x \) as bound variables
    • The name of the argument is \( x \)
    • The local name of the function is \( f \)
  - Both \( f \) and \( x \) have \( e \) as their scope
\[\alpha\text{-conversion}\]

• The following are the exact same programs:
  \[
  \text{fun } f(x:\text{Int}):\text{Int is } f(h(x+y)) \\
  \text{fun } g(z:\text{Int}):\text{Int is } f(h(x+y))
  \]

• But not:
  \[
  \text{fun } h(y:\text{Int}):\text{Int is } h(h(y+y))
  \]
Example Code

(fun f(x:Int):Int is x+1) 3
let
  fact be (fun g(y:Int):Int is
      if (y=0) then 1
      else y*(g(y-1)))
in
  (fact 3) + (fact 4)
Dynamic Semantics

• Add the rules

\[
\begin{align*}
\frac{e_1 \rightarrow e_1'}{e_1 \ e_2 \rightarrow e_1' \ e_2} & \quad \frac{e_2 \rightarrow e_2'}{v_1 \ e_2 \rightarrow v_1 \ e_2'}
\end{align*}
\]
Static Semantics

\[ t ::= \text{Int} \mid \text{Bool} \mid t \rightarrow t \]
Static Semantics

\[ t ::= \text{Int} \mid \text{Bool} \mid t \rightarrow t \]

\[
\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : ? \\
\hline
\Gamma \vdash e_1 \; e_2 : t
\]
Static Semantics

$$t ::= \text{Int} \mid \text{Bool} \mid t \rightarrow t$$

\[
\Gamma \vdash e_1 : t_2 \rightarrow t \quad \Gamma \vdash e_2 : t_2 \\
\Gamma \vdash e_1 \; e_2 : t
\]
Static Semantics

\[ t ::= \text{Int} | \text{Bool} | t \rightarrow t \]

\[
\begin{array}{c}
\Gamma \vdash e_1 : t_2 \rightarrow t \quad \Gamma \vdash e_2 : t_2 \\
\hline
\Gamma \vdash e_1 \ e_2 : t
\end{array}
\]

???

\[
\begin{array}{c}
???
\hline
\Gamma \vdash \text{fun} \ f(x:t_1):t_2 \text{ is } e : ???
\end{array}
\]
Static Semantics

\[ t ::= \text{Int} \mid \text{Bool} \mid t \rightarrow t \]

\[ \Gamma \vdash e_1 : t_2 \rightarrow t \quad \Gamma \vdash e_2 : t_2 \]
\[ \Gamma \vdash e_1 \ e_2 : t \]

\[ \text{fun} \ f(x : t_1) : t_2 \text{ is } e : t_1 \rightarrow t_2 \]
Static Semantics

\[ t ::= \text{Int} \mid \text{Bool} \mid t \rightarrow t \]

\[
\begin{align*}
\Gamma &\vdash e_1 : t_2 \rightarrow t \\
\Gamma &\vdash e_2 : t_2 \\
\Gamma &\vdash e_1 \; e_2 : t
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : t_1, f : t_1 \rightarrow t_2 &\vdash e : t_2 \\
\Gamma &\vdash \text{fun } f(x : t_1) : t_2 \text{ is } e : t_1 \rightarrow t_2
\end{align*}
\]
Dynamic Semantics

• Add the rules

\[
\begin{align*}
  e_1 &\rightarrow e_1' \\
  e_1 &\rightarrow e_1' \\
  e_2 &\rightarrow e_2' \\
  v_1 &\rightarrow v_1 \\
  e_2 &\rightarrow e_2'
\end{align*}
\]
Dynamic Semantics

- Add the rules

\[
\begin{align*}
  &e_1 \rightarrow e_1' \\
\hline
  &e_1 e_2 \rightarrow e_1' e_2 \\
\end{align*}
\]

\[
\begin{align*}
  &e_2 \rightarrow e_2' \\
\hline
  &v_1 e_2 \rightarrow v_1 e_2' \\
\end{align*}
\]

\[
(f\text{un } f(x:t_1):t_2 \text{ is } e_2) \ v \rightarrow \ ???
\]
Dynamic Semantics

• Add the rules

\[
\begin{align*}
    e_1 & \rightarrow e_1' \\
    e_1 \ e_2 & \rightarrow e_1' \ e_2 \\
    e_2 & \rightarrow e_2' \\
    v_1 \ e_2 & \rightarrow v_1 \ e_2'
\end{align*}
\]

\[
\frac{(\text{fun } f(x:t_1):t_2 \text{ is } e_2) \ v \rightarrow e_2[x\rightarrow?, f\rightarrow?]}{}
\]
Dynamic Semantics

• Add the rules

\[
\begin{align*}
  e_1 & \rightarrow e_1' \\
  e_1 \ e_2 & \rightarrow e_1' \ e_2 \\
  e_2 & \rightarrow e_2' \\
  v_1 \ e_2 & \rightarrow v_1 \ e_2'
\end{align*}
\]

\[
(fun \ f(x:t_1):t_2 \ is \ e_2) \ v \rightarrow \\
(e_2[x\rightarrow v])\ [f\rightarrow?] \]

Dynamic Semantics

• Add the rules

\[
\begin{align*}
\text{e}_1 & \rightarrow \text{e}_1' \\
\text{e}_1 \text{ e}_2 & \rightarrow \text{e}_1' \text{ e}_2 \\
\text{e}_2 & \rightarrow \text{e}_2' \\
\text{v}_1 \text{ e}_2 & \rightarrow \text{v}_1 \text{ e}_2'
\end{align*}
\]

\[
\text{(fun } f(\text{x}:\text{t}_1):\text{t}_2 \text{ is } \text{e}) \text{ v} \rightarrow (\text{e}_2[\text{x} \rightarrow \text{v}])[f \rightarrow (\text{fun } f(\text{x}:\text{t}_1):\text{t}_2 \text{ is } \text{e}_2)]
\]