Introduction to Semantics

February 5, 2001
CS 131: Programming Languages

Review: Syntax

- The syntax of a language determines what language constructs can/must occur where.

\[
\begin{align*}
\text{<comm>} & ::= \text{<var>} ::\text{=} \text{<exp>} \\
\text{while} \text{<exp>} \text{do} \text{<comm>} \\
\text{if} \text{<exp>} \text{then} \text{<comm>} \text{else} \text{<comm>} \\
\text{<comm>} ; \text{<comm>} \\
\{ \text{<comm>} \} & \text{...}
\end{align*}
\]

\[
\begin{align*}
\text{<exp>} & ::= \text{<var>} | \text{<int>} | \text{<real>} \\
& | (\text{<exp>} + \text{<exp>}) \\
& | (\text{<exp>} - \text{<exp>}) \\
& | (\text{<exp>} \geq \text{<exp>}) & \text{...}
\end{align*}
\]

Review: Concrete & Abstract Syntax

Review: Concrete Syntax

- Concrete syntax is "arbitrary"

\[
\begin{align*}
\text{<num>} & ::= \text{one} | \text{two} | \text{three} & \text{...} \\
\text{<exp>} & ::= \text{<num>} \\
& | \text{add} \text{<exp>} \text{and} \text{<exp>} \\
& | \text{subtract} \text{<exp>} \text{from} \text{<exp>} \\
& | \text{multiply} \text{<exp>} \text{by} \text{<exp>} \\
& | \{ \text{<exp>} \} \\
\end{align*}
\]

\[
\text{subtract} \text{(add three plus five)} \text{from two}
\]

\[
\begin{align*}
2 - (3+5) & = 2 - 8 \\
& = -6
\end{align*}
\]
Review: Concrete Syntax

- Concrete syntax is “arbitrary”

```
<num> ::= 1 | 2 | 3 | ...
<exp> ::= <num>
| <exp> <exp> *
| <exp> <exp> +
| <exp> <exp> -
```

```
2 3 5 + -
```

Review: Abstract Syntax

- Recall: starting today we will write abstract syntax trees in non-tree form.
  - We always have a specific tree in mind, though.
  - Free to use parentheses, conventions to hint at which tree we have in mind.
  - Can specify abstract syntax with grammar as well

```
n ::= 1 | 2 | 3 | ...
e ::= n | e + e | e - e | e * e
```

```
2 - (3+5)
```

Semantics

- To understand a programming language, not enough to know its syntax.
- The semantics of a language specifies the meaning of a program
  - What program phrases mean when put together.
  - What answer does each program produce?
  - How should execution proceed?

- The large majority of the work in defining a language is specifying the semantics.

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: very hard to give a formal description of a full, real language
  - But can handle quite large subsets
  - Active research topic

Two Approaches to Formal Semantics

- Denotational semantics
  - The meaning of every program phrase is a mathematical object (a number, a function, a pair, a sequence, etc.)
  - Compositionality: meaning of an expression is a function of the meanings of its sub-expressions.
    - E.g., the meaning of the loop `while b do c` is calculated from the meanings of the guard expression `b` and of the loop body `c`
  - 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

Semantics for Machine Language

- Idea: A computer is just a big state machine
  - In principle, could give a precise specification for how a particular computer behaves
    - Cycle-by-cycle description of how the machine state changes
  - This would provides a semantics for a particular machine language

- What does a given machine language program mean?
  - Just run it and see what happens
Leveraging Machine Language?

- By specifying, for example, a particular C++ compiler, could get a semantics for C++
  - What's the meaning of a C++ program? Compile it and run it to see what happens.
- Problems
  - Assumes we can formalize the compilation process
    - Also, C++ compilers are generally large and complex
  - As are modern CPUs (pipelines, interrupts, ...)
  - Specifies too many irrelevant details
    - Order of evaluation, memory layout, etc.

Abstract Machines

- Idea:
  - Don't specify a real CPU
  - Specify any "machine" that is convenient
- Advantages
  - Can ignore any issues that aren't relevant
  - Can tailor the machine to a specific language
    - e.g., JVM specification for Java bytecode

Specifying an Abstract Machine

- Ingredients
  1. Description of the possible machine states
  2. Which of these are final states
    - Mark the end of execution
  3. Description of the execution process (also known as the dynamic semantics)
    - Small-step: Define relation \( s_1 \rightarrow s_2 \)
      "If the machine starts in state \( s_1 \) then after one step (or cycle) the machine can be in state \( s_2 \)."
    - Big-step: Define relation \( s \downarrow o \)
      "If the machine starts in state \( s \), then after many steps it can yield output \( o \)."

An AM for Simple Arithmetic

- Abstract syntax
  \[
  \begin{align*}
  v & ::= n \\
  e & ::= v \mid e + e
  \end{align*}
  \]  
  (values)  
  (expressions)

- Abstract Machine states
  \[
  \begin{align*}
  \text{states} & ::= e \\
  \text{final states} & ::= v
  \end{align*}
  \]  
  (eventually the machine states will be more complex)
An AM for Arithmetic

- We will define the (small-step) transition relation by specifying a collection of inference rules.
  - Provides an inductive definition of the transition relation.
- We say that \( s_1 \to s_2 \) holds (or is true) iff there is a proof of this using the given rules!

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

Examples

- \( 3 + 5 \to \)
- \( (3+4) + 5 \to \)
- \( 7 + (5+3) \to \)
- \( (2+5) + (5+3) \to \)

Multi-Step Evaluation

- We define the relation \( \to^* \) to be the reflexive, transitive closure of \( \to \).
  - That is,
    \[
    \begin{align*}
    s_1 & \to^* s_2 \quad \text{if } s_1 \to s_2 \\
    s & \to^* s \quad \text{always.}
    \end{align*}
    \]
    - \( s_1 \to^* s_3 \) if \( s_1 \to^* s_2 \) and \( s_2 \to^* s_3 \) for some \( s_2 \)

Multi-Step Evaluation

- Equivalent definition using inference rules

\[
\begin{align*}
s_1 & \to s_2 \\
s_1 & \to^* s_2 \\
\vdots & \\
s & \to^* s
\end{align*}
\]

\[
\begin{align*}
s_1 & \to^* s_2 \\
s_2 & \to^* s_3 \\
s_1 & \to^* s_3
\end{align*}
\]
Multi-Step Termination

• A program $p$ may terminate with value $v$ if $p \rightarrow^* v$

  For example,

  $$(3+4)+(5+3) \rightarrow^* 15$$

• A program $p$ may fail to terminate (or may diverge) if $p \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow ...$

Determinacy

• The evaluation rules as given are non-deterministic.
  - A single state may step to several different states.
  - No specified evaluation order

• For this language, it doesn’t make much difference.
• But, for larger languages we might care.
  - How can we specify order of evaluation?

Left-to-Right Evaluation

$\begin{array}{c}
  n_1 + n_2 \rightarrow n_1 \oplus n_2 \\
  e_1 \rightarrow e_1' \\
  e_1 + e_2 \rightarrow e_1' + e_2 \\
  v + e_2 \rightarrow v + e_2'
\end{array}$

$\begin{array}{c}
  e_2 \rightarrow e_2' \\
  e_1' + e_2 \rightarrow e_1' + e_2' \\
  e_1 + e_2 \rightarrow e_1 + e_2'
\end{array}$

Now $(2+5)+(5+3) \rightarrow ?$

How would we specify right-to-left evaluation?

Adding Booleans

• Abstract Syntax
  
  $v ::= n \mid \text{tt} \mid \text{ff}$ (values)
  
  $e ::= v \mid e + e \mid e < e \mid \text{if } e \text{ then } e \text{ else } e$ (expressions)

  For example,

  $\text{if } (\text{if } 3 \leq 5 \text{ then } \text{ff} \text{ else } \text{tt}) \text{ then } 1 + 2 \text{ else } 7 + 8$

  • States are still expressions, terminal states are values.
Dynamic Semantics

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

Stuck Programs

- Now have non-final states that can't make progress:
  \[
  3 + tt
  \]
  \[
  \text{if tt \leq ff then 3 else 5}
  \]
  \[
  \text{if 4 then tt else ff}
  \]
- Answer 1: who cares?
  - Implementation-dependent behavior
  - Or, program should yield run-time error.
- Answer 2: use a type system
  - "Prove" that such bad cases won't arise at run-time
  - We'll look at this much more, later.

Adding Local Definitions

- Abstract Syntax
  \[
  \begin{align*}
  v ::= n \mid \text{tt} \mid \text{ff} & \quad \text{(values)} \\
  e ::= v \mid e + e \mid e < e \mid \text{if } e \text{ then } e \text{ else } e \mid x \mid \text{let } x \text{ be } e \text{ in } e & \quad \text{(expressions)}
  \end{align*}
  \]
Scoping of Variables

• In

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

the variable \( x \) is bound, and its scope is \( e_2 \).

• All \( \alpha \)-equivalent expressions are identified
  – E.g., the following are the same expression
    \[ \text{let } x \text{ be } 3 \text{ in } x + y \]
    \[ \text{let } z \text{ be } 3 \text{ in } z + y \]
  But not
    \[ \text{let } y \text{ be } 3 \text{ in } y + y \]

Changes to Dynamic Semantics

• Add two rules:

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

Examples

• \( \text{let } x \text{ be } 3+4 \text{ in } (1+2)+x \rightarrow \)

• \( \text{let } x \text{ be } 1+1 \text{ in } x+x \rightarrow \)

Alternative Dynamic Semantics

• What if we had just this single rule instead?

\[
\begin{align*}
\text{let } x \text{ be } e_1 \text{ in } e_2 & \rightarrow e_2[x \rightarrow e_1]
\end{align*}
\]

• Consider

\( \text{let } x \text{ be } 1+2 \text{ in } x+x \)