Introduction to Semantics

February 5, 2001

CS 131: Programming Languages

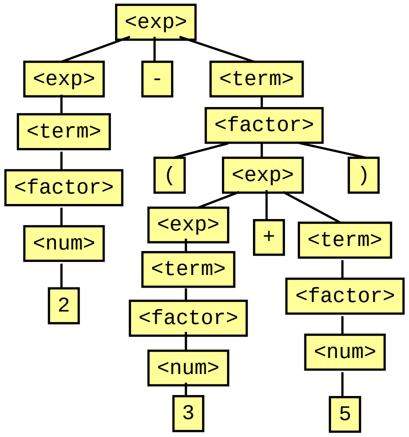
Review: Syntax

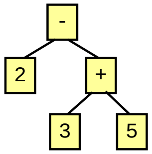
- The *syntax* of a language determines what language constructs can/must occur where.

```
<comm> ::=  <var> := <exp>
         |  while <exp> do comm
         |  if <exp> then <comm> else <comm>
         |  <comm> ; <comm>
         |  { <comm> } | ...

<exp>  ::=  <var> | <int> | <real>
         |  ( <exp> + <exp> )
         |  ( <exp> - <exp> )
         |  ( <exp> >= <exp> ) | ...
```

Review: Concrete & Abstract Syntax





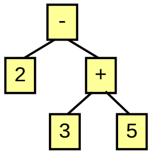
2-(3+5)

Review: Concrete Syntax

- Concrete syntax is "arbitrary"

```
<num> ::= one | two | three | ...
<exp> ::= <num>
        | add <exp> and <exp>
        | subtract <exp> from <exp>
        | multiply <exp> by <exp>
        | ( exp )
```

```
subtract (add three plus five)
from two
```

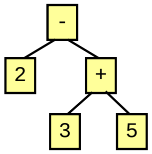


Review: Concrete Syntax

- Concrete syntax is "arbitrary"

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

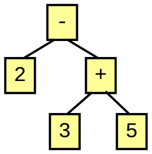
`2 3 5 + -`



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 CPU's (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

  v ::= n (values)

  e ::= v | e + e (expressions)

- Abstract Machine states

  states ::= e

  final states ::= v

(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 $s_1 \rightarrow s_2$ holds (or is true) iff there is a proof of this using the given rules!

$$\frac{}{n_1+n_2 \rightarrow n_1 \oplus n_2}$$

$$\frac{e_1 \rightarrow e_1'}{e_1+e_2 \rightarrow e_1'+e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1+e_2 \rightarrow e_1+e_2'}$$

Examples

- `3+5` →
- `(3+4)+5` →
- `7+(5+3)` →
- `(2+5)+(5+3)` →

Multi-Step Evaluation

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

$s_1 \rightarrow^* s_2$ if $s_1 \rightarrow s_2$

$s \rightarrow^* s$ always.

$s_1 \rightarrow^* s_3$ if $s_1 \rightarrow^* s_2$ and $s_2 \rightarrow^* s_3$ for some s_2

Multi-Step Evaluation

- Equivalent definition using inference rules

$$\frac{s_1 \rightarrow s_2}{s_1 \rightarrow^* s_2}$$

$$\frac{}{s \rightarrow^* s}$$

$$\frac{s_1 \rightarrow^* s_2 \qquad s_2 \rightarrow^* s_3}{s_1 \rightarrow^* s_3}$$

Multi-Step Termination

- A program p *may terminate with value* v if

$$p \to^* v$$

For example,

$$\texttt{(3+4)+(5+3)} \quad \to^* \quad \texttt{15}$$

- A program p *may fail to terminate* (or may *diverge*) if

$$p \to p_1 \to p_2 \to p_3 \to \ldots$$

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?

$$\frac{}{n_1 + n_2 \to n_1 \oplus n_2}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}$$

Left-to-Right Evaluation

$$\frac{}{n_1 + n_2 \to n_1 \oplus n_2}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \cancel{\frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}}$$

$$\frac{e_2 \to e_2'}{v + e_2 \to v + e_2'}$$

Now `(2+5)+(5+3)` → ?

How would we specify right-to-left evaluation?

Adding Booleans

- Abstract Syntax

$$
\begin{array}{lll}
v & ::= n \mid \texttt{tt} \mid \texttt{ff} & \text{(values)} \\
e & ::= v \mid e + e \mid e < e & \text{(expressions)} \\
 & \quad \mid \textbf{if } e \textbf{ then } e \textbf{ else } e &
\end{array}
$$

For example,

if (**if** 3 ≤ 5 **then ff else tt**) **then** 1+2 **else** 7+8

- States are still expressions, terminal states are values.

Dynamic Semantics

$$\frac{}{n_1 + n_2 \rightarrow n_1 \oplus n_2}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \frac{e_2 \rightarrow e_2'}{v + e_2 \rightarrow v + e_2'}$$

$$\frac{}{n_1 \leq n_2 \rightarrow n_1 \textcircled{$\leq$} n_2}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 \leq e_2 \rightarrow e_1' \leq e_2} \qquad \frac{e_2 \rightarrow e_2'}{v \leq e_2 \rightarrow v \leq e_2'}$$

Dynamic Semantics

$$\frac{e_1 \rightarrow e_1'}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightarrow \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3}$$

$$\frac{}{\mathbf{if\ tt\ then}\ e_2\ \mathbf{else}\ e_3 \rightarrow ?}$$

$$\frac{}{\mathbf{if\ ff\ then}\ e_2\ \mathbf{else}\ e_3 \rightarrow ?}$$

Stuck Programs

- Now have non-final states that can't make progress:

```
3 + tt
if tt ≤ ff then 3 else 5
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

v ::= n | **tt** | **ff** (values)

e ::= v | e + e | e < e (expressions)

| **if** e **then** e **else** e

| x

| **let** x **be** e **in** e

Scoping of Variables

- In
  $\mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2$
  the variable x is bound, and its scope is e_2.

- All α-equivalent expressions are identified
  - E.g., the following are the same expression
    `let x be 3 in x + y`
    `let z be 3 in z + y`
    But not
    `let y be 3 in y + y`

Changes to Dynamic Semantics

- Add two rules:

$$\frac{e_1 \to e_1'}{\mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \to \mathbf{let}\ x\ \mathbf{be}\ e_1'\ \mathbf{in}\ e_2}$$

$$\frac{}{\mathbf{let}\ x\ \mathbf{be}\ v_1\ \mathbf{in}\ e_2 \to e_2[x \to v_1]}$$

Examples

- `let x be 3+4 in (1+2)+x` →

- `let x be 1+1 in x+x` →

Alternative Dynamic Semantics

- What if we had just this single rule instead?

$$\frac{}{\mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \to e_2[x \to e_1]}$$

- Consider
  `let x be 1+2 in x+x`