CPSC 311: Definition of Programming Languages: Specifying dynamic semantics ("04-operational")

Joshua Dunfield
University of British Columbia
(portions based on notes by Brigitte Pientka, McGill University)

September 19, 2015

What do programs mean? They mean whatever the[1] language definition says they do. So the real question is: How do we specify, in a language definition, the meaning of the language's programs?

1 Why dynamic semantics?

Unlike syntax, where practically all language designers[2] uses some variation of BNF grammars, specifying which syntactically well-formed programs actually mean something, and what they mean, is less settled.

Various methods have been used, with names like "axiomatic semantics", "operational semantics", "natural semantics", and "denotational semantics". Within the programming languages research community, there is lively competition amongst these methods. To most of the world, though, this competition is off the radar: most languages' semantics are specified informally. (Standard ML is probably the most popular formally defined language—and Standard ML is even less "mainstream" than Scheme/Racket.)

In 311, we will focus on one method, *operational semantics*. Given a specification in operational semantics, it is relatively easy (compared to specifications using other methods) to write an interpreter. Operational semantics has a rich mathematical foundation, which you would want to understand to do research in programming languages, but you don't need to understand that foundation to turn operational semantics into interpreters. In lecture, I turned my mathematical definition of *subst* into a Racket function, without worrying about whether my definition had good mathematical properties.

The idea of trying to specify the meaning of a mathematical object (a program) through natural language alone, rather than more "formally" (through logic and mathematics), calls to mind a quotation:

> "About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead." —Edsger Dijkstra

Formal mathematical language is not an absolute guarantee against mistakes or oversights in defining the semantics (a serious mistake in SML's formal definition went unnoticed for years), but it, at least, gives us a point of reference. Rules in natural language are for people, not computers; understanding a programming language shouldn't require one to be a "language lawyer".

2 Evaluation semantics

As I mentioned, operational semantics is closer to an interpreter than other methods for specifying what a program does. There are different flavours of operational semantics; we'll start with the one that's usually easier to understand, called *evaluation semantics*.

(That is, evaluation semantics is one kind of operational semantics, which is one method for specifying dynamic semantics. But for now, just remember: what we're going to do, right now, is called evaluation semantics.)

[1] We'll assume that we know *which* language the program is written in, despite programs such as http://ideology.com.au/polyglot/polyglot.txt.

[2] One exception: the designers of Algol 68, who tried to innovate in this area; it didn't end well.

The idea of evaluation semantics is that the dynamic behaviour—the dynamic "meaning" of a program—is *the value it computes*, or equivalently, *what it evaluates to*. We expect that {+ 2 2} will compute 4, or equivalently, will evaluate to 4.

For our very first example of evaluation semantics, we'll follow the language "AE" from Chapter 2 of PLAI. Its concrete syntax, or EBNF—as given on p. 7 of PLAI—is

$$\begin{aligned}\langle \texttt{AE} \rangle \ ::= \ & \langle \texttt{num} \rangle \\ & \mid \{+ \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\} \\ & \mid \{- \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\}\end{aligned}$$

Also, recall its abstract syntax (PLAI, p. 6):

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])
```

However, we'll use the concrete syntax to write the evaluation semantics. This allows programmers to use our semantics to understand the language, provided they can read evaluation semantics rules. The only people who should have to know what the abstract syntax looks like are the people writing the interpreter (or reading the interpreter's source code in the textbook).

Now, let's write down a specification of the dynamic behaviour of this language, using the method of evaluation semantics. First, we should ask: what is our goal? How will we know when we have a complete (not necessarily *good*, but complete) specification? One answer (which is not always a good answer, but will work just fine for this language) is: if we can specify the meaning of all expressions that are syntactically well-formed (according to the EBNF for $\langle \texttt{AE} \rangle$), then we have a complete specification.

Thus, we need to specify the meaning of each of the three syntactic cases ($\langle \texttt{num} \rangle$, $+$ and $-$) in the EBNF. Since we're going to use evaluation semantics, we need to specify what a $\langle \texttt{num} \rangle$ evaluates to, what a $+$ evaluates to, and what a $-$ evaluates to.

This language is so tiny, and there's only one reasonable way it can work: $+$ should add, and $-$ should subtract. So we can focus on *how* to write down an evaluation semantics, rather than spend time wondering if we're making good design decisions.

We want to be *really* precise, so let's try to be a little more precise than just saying "$+$ should add, and $-$ should subtract". Just as CPSC 110 shows how to follow a data definition (for example, if you need to write a function that takes a BST (binary search tree), you need to write a case for 'false' and a case for '(make-node . . .)'), let's try to follow the BNF:

- we need to say what a number evaluates to,
- we need to say what a $+$ evaluates to, and
- we need to say what a $-$ evaluates to.

This is a little vague, though, because we didn't mention the subexpressions of $+$ and $-$. Let's fix that, and also (in the first case) mention the specific number!

- we need to say what a number $\texttt{n}$ evaluates to,
- we need to say what $\{+ \ \texttt{AE1} \ \texttt{AE2}\}$ evaluates to, and
- we need to say what $\{- \ \texttt{AE1} \ \texttt{AE2}\}$ evaluates to.

Here, $\texttt{AE1}$ stands for the first subexpression, and $\texttt{AE2}$ stands for the second subexpression.

Now let's actually say (in English) what these things evaluate to. A number shouldn't *do* anything, so we'll say that it evaluates to itself:

- A number n evaluates to n.

What should {+ AE1 AE2} evaluate to? Well, that depends on what AE1 and AE2 are. Or rather, what they evaluate to. So let's start there.

- If AE1 evaluates to n_1, and AE2 evaluates to n_2, then {+ AE1 AE2} evaluates to ...

We want + to *add*, so it needs to add n_1 to n_2.

- If AE1 evaluates to n_1, and AE2 evaluates to n_2, then {+ AE1 AE2} evaluates to $n_1 + n_2$.

Now we can give meaning to − in the same way, resulting in something reasonably precise (it's still in English):

- A number n evaluates to n.
- If AE1 evaluates to n_1, and AE2 evaluates to n_2, then {+ AE1 AE2} evaluates to $n_1 + n_2$.
- If AE1 evaluates to n_1, and AE2 evaluates to n_2, then {− AE1 AE2} evaluates to $n_1 - n_2$.

2.1 Rules

We're now very close to an evaluation semantics! In fact, all we have to do is rewrite the above using some funny notation: Instead of "AE evaluates to n", we'll write "AE $\Downarrow n$". And instead of "If... then ...", we'll use a horizontal line, like this:

$$\frac{\text{AE1} \Downarrow n_1 \qquad \text{AE2} \Downarrow n_2}{\{- \text{ AE1 AE2}\} \Downarrow n_1 - n_2} \text{ Eval-sub}$$

This notation was invented by the logician Gerhard Gentzen.

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \qquad \ldots \qquad \text{premise}_m}{\text{conclusion}} \text{ rule name}$$

The part below the line is called the *conclusion*, and the parts above the line are called the *premises*. To the right of the line, we often write the name of the rule. We can read a rule as follows: To derive the conclusion, we must satisfy each of the premises. In other words, if the premises are satisfied, we have shown the conclusion. Or, very briefly, "if premises, then conclusion".

Rules always have a conclusion, but they don't have to have premises. In fact, to write down the rule for our first case ("A number n evaluates to n"), we don't need any premises:

$$\frac{}{n \Downarrow n} \text{ Eval-num}$$

Often, the rule for a syntactic form will have exactly one premise for each smaller expression it contains. A number doesn't contain any subexpressions, so the evaluation rule for numbers doesn't have any premises. On the other hand, {− AE1 AE2} has two subexpressions so its rule has two premises.

What can you do with a rule? You can *apply* it, by filling in its "meta-variables". Here, our "meta-variables" are n (in Eval-num), and AE1, AE2, n_1, and n_2 in Eval-add and Eval-sub. A meta-variable is a placeholder: we can fill in AE1 and AE2 with ⟨AE⟩'s, and we can fill in n, n_1 and n_2 with numbers.

This is easier to see with an example. Given the rule

$$\frac{}{n \Downarrow n} \text{ Eval-num}$$

we can apply it by plugging in an actual number for the meta-variable n:

$$\frac{}{7 \Downarrow 7}$$

Once we've applied Eval-num, we have an *evaluation derivation* of $7 \Downarrow 7$, and say that we have *derived* $7 \Downarrow 7$.

Note that, unlike our EBNF grammar—where we wrote $\langle \texttt{AE} \rangle$ twice in the production for $+$ to refer to (possibly) *different* expressions—writing n twice in the rule Eval-num means that we have to substitute the same number.

We can similarly derive $6 \Downarrow 6$:

$$\frac{}{6 \Downarrow 6}$$

This gives us two derivations, one of $7 \Downarrow 7$ and one of $6 \Downarrow 6$, so we have enough derivations to apply Eval-sub:

$$\frac{\dfrac{}{7 \Downarrow 7} \qquad \dfrac{}{6 \Downarrow 6}}{\{- \; 7 \; 6\} \Downarrow 1}$$

We got this by looking at the rule Eval-sub, plugging in 7 for AE1, plugging in 6 for AE2, plugging in 7 for n_1, and 6 for n_2. The conclusion of Eval-sub says "$\ldots \Downarrow n_1 - n_2$", which—after plugging in for n_1 and n_2—is $\ldots \Downarrow 7 - 6$, which is $\ldots \Downarrow 1$.

Notice that this derivation of $\{- \; 7 \; 6\} \Downarrow 1$ looks like a tree (oriented the natural way, with the root at the bottom, rather than the usual computer science way). And in fact, derivations are also called derivation trees. This is a nice feature of Gentzen's notation: derivations "fit together" visually.

Here's a slightly larger example:

$$\frac{\dfrac{\dfrac{}{20 \Downarrow 20} \qquad \dfrac{}{2 \Downarrow 2}}{\{+ \; 20 \; 2\} \Downarrow 22} \qquad \dfrac{\dfrac{}{7 \Downarrow 7} \qquad \dfrac{}{6 \Downarrow 6}}{\{- \; 7 \; 6\} \Downarrow 1}}{\{- \; \{+ \; 20 \; 2\} \; \{- \; 7 \; 6\}\} \Downarrow 21}$$

It's often useful to write the names of the rules being applied (later languages will have more than just three rules!):

$$\dfrac{\dfrac{\dfrac{}{20 \Downarrow 20}\,\text{Eval-num} \qquad \dfrac{}{2 \Downarrow 2}\,\text{Eval-num}}{\{+ \; 20 \; 2\} \Downarrow 22}\,\text{Eval-add} \qquad \dfrac{\dfrac{}{7 \Downarrow 7}\,\text{Eval-num} \qquad \dfrac{}{6 \Downarrow 6}\,\text{Eval-num}}{\{- \; 7 \; 6\} \Downarrow 1}\,\text{Eval-sub}}{\{- \; \{+ \; 20 \; 2\} \; \{- \; 7 \; 6\}\} \Downarrow 21}\,\text{Eval-sub}$$

2.2 Evaluation rules for AEs

In PL research papers, it's customary to collect all the evaluation rules together, and throw one giant figure at the reader. Fortunately, we only have three rules.

$$\frac{}{n \Downarrow n}\,\text{Eval-num} \qquad \frac{\text{AE1} \Downarrow n_1 \qquad \text{AE2} \Downarrow n_2}{\{+ \; \text{AE1} \; \text{AE2}\} \Downarrow n_1 + n_2}\,\text{Eval-add} \qquad \frac{\text{AE1} \Downarrow n_1 \qquad \text{AE2} \Downarrow n_2}{\{- \; \text{AE1} \; \text{AE2}\} \Downarrow n_1 - n_2}\,\text{Eval-sub}$$

3 From the rules to an interpreter

Now we'll write an interpreter that follows our evaluation rules. This interpreter will turn out to do the same thing as PLAI's interpreter in Chapter 2. The difference is how we got there. Once you understand how to

write interpreters based on evaluation rules, you can take evaluation rules you've never seen before—and that may define a language with features you've never heard of—and write an interpreter that follows those rules.

You won't get that skill instantly just from this one tiny language, but you have to start somewhere!

3.1 Restating the rules in abstract syntax

It's easier to work with abstract syntax—the "AE" defined with define-type—than concrete syntax, so our interpreter will accept programs in abstract syntax. You can learn to mentally translate between concrete and abstract syntax, but for now, let's explicitly translate the rules to abstract syntax. We just have to change all the AEs, inserting the constructors num, add and sub.

(I'm also going to write AE1 and AE2 in lowercase. I apologize for the extra confusion now; it will save us some annoyance later.)

$$\frac{}{(\mathsf{num}\ n) \Downarrow n}\ \text{Eval-num} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\mathsf{add}\ ae1\ ae2) \Downarrow n_1 + n_2}\ \text{Eval-add} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\mathsf{sub}\ ae1\ ae2) \Downarrow n_1 - n_2}\ \text{Eval-sub}$$

This shows something interesting, though: the animals on each side of the "evaluates to" arrow (⇓) are not the same kind of animal.[3] In Eval-num, we have an AE, (num n), on the left of ⇓, but a plain number n on the right. In the concrete syntax, we didn't write num explicitly, so we couldn't see this difference. We could have chosen, instead, to "evaluate cats to cats" and produce an AE on the right, but it's a little more convenient to produce a number. (Later in 311, we'll define other flavours of operational semantics that don't work this way.)

The job of writing an interpreter for AEs boils down to writing a function that answers this question:

"Given an ae, find a number n such that $ae \Downarrow n$."

During lecture, we wrote the following function:

```
(define (interp ae)
  (type-case AE ae
    [num (n) n]
    [add (ae1 ae2)
        (let ([n1 (interp ae1)]
              [n2 (interp ae2)])
          (+ n1 n2))]
    [sub (ae1 ae2)
        (let ([n1 (interp ae1)]
              [n2 (interp ae2)])
          (- n1 n2))]
    ))
```

Our `interp` function behaves the same as the `calc` function in PLAI, but our function has more let-bindings. This is more verbose, but strengthens the connection between our interpreter and the rules. For example, the expression `(+ n1 n2)` is a direct Racket translation (parentheses and a prefix operator +) of the $n_1 + n_2$ that appears in the conclusion of Eval-add.

4 The WAE language

Let's extend the evaluation semantics to a slightly bigger language: WAE, which adds the "with" construct. Here's the concrete syntax (PLAI, p. 16):

[3] In honour of my undergrad discrete math professor's advice: "You must always ask yourself: what kind of an animal is it?"

$$\begin{array}{rcl}\langle \text{WAE} \rangle & ::= & \langle \text{num} \rangle \\ & | & \{+ \langle \text{WAE} \rangle \langle \text{WAE} \rangle\} \\ & | & \{- \langle \text{WAE} \rangle \langle \text{WAE} \rangle\} \\ & | & \{\text{with } \{\langle \text{id} \rangle \langle \text{WAE} \rangle\} \langle \text{WAE} \rangle\} \\ & | & \langle \text{id} \rangle\end{array}$$

And here's the abstract syntax (PLAI, p. 16):

```
(define-type WAE
 [num (n number?)]
 [add (lhs WAE?) (rhs WAE?)]
 [sub (lhs WAE?) (rhs WAE?)])
 [with (name symbol?) (named-expr WAE?) (body WAE?)]
 [id (name symbol?)])
```

4.1 Rules

We just added two new constructors (variants) to the **define-type** declaration, so we need to say what they mean. For convenience, I'll go straight to abstract syntax this time.

Let's bring in all the evaluation rules from the AE language, but we'll write $e1$ and $e2$ instead of $ae1$ and $ae2$.

$$\frac{}{(\text{num } n) \Downarrow n}\ \text{Eval-num} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\text{add } e1\ e2) \Downarrow n_1 + n_2}\ \text{Eval-add} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\text{sub } e1\ e2) \Downarrow n_1 - n_2}\ \text{Eval-sub}$$

The rule for with says that, if $e1$ (called *named-expr* in WAE's **define-type**) evaluates to a value $v1$, and substituting that value for x in $e2$ (called *body* in WAE's **define-type**) gives $v2$, then the entire with evaluates to $v2$.

$$\frac{e1 \Downarrow v1 \qquad \mathit{subst}(e2, x, (\text{num } v1)) \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Eval-with}$$

Remark. Using $v1$ and $v2$ in Eval-with isn't consistent with n, n_1 and n_2 in the other rules. But it's not quite wrong: by convention, v stands for any *value*; in this simple WAE language, the only values we have are numbers.

Exercise 1.
Write the above rule using *concrete* syntax, as defined by the grammar for $\langle \text{WAE} \rangle$, instead of abstract syntax. (Assume that $\mathit{subst}(e2, x, v1)$ works on concrete syntax.)

The above rule uses the (mathematical) function *subst* that was defined in lecture. Let's see that definition again. Actually, to develop the connection between concrete and abstract syntax, let's see versions for both concrete and abstract syntax, side-by-side in Figure 1.

One thing to notice about this definition of substitution is that it does *not* refer to evaluation: we didn't use the "evaluates to" symbol ($\Downarrow$). Rather, our new evaluation rule (Eval-with) "calls" substitution. Effectively, substitution is a (mathematical) "helper function" for the evaluation semantics.

Example

Let's try to write an evaluation derivation for $\big(\text{with } x\ (\text{num } 1)\ (\text{add } (\text{id } x)\ (\text{id } x))\big)$. We have a with, so we need to apply Eval-with:

$$\frac{(\text{num } 1) \Downarrow \_\_\_ \qquad \mathit{subst}((\text{add } (\text{id } x)\ (\text{id } x)), x, (\text{num } \_\_\_)) \Downarrow \_\_\_}{\big(\text{with } x\ (\text{num } 1)\ (\text{add } (\text{id } x)\ (\text{id } x))\big) \Downarrow \_\_\_}\ \text{Eval-with}$$

Substitution, for WAE concrete syntax

$$
\begin{aligned}
\mathit{subst}(n, x, v) &= n \\
\mathit{subst}(x, x, v) &= v \\
\mathit{subst}(y, x, v) &= y \quad \text{if } x \neq y \\
\mathit{subst}(\{+\ eL\ eR\}, x, v) &= \{+\ \mathit{subst}(eL, x, v)\ \mathit{subst}(eR, x, v)\} \\
\mathit{subst}(\{-\ eL\ eR\}, x, v) &= \{-\ \mathit{subst}(eL, x, v)\ \mathit{subst}(eR, x, v)\} \\
\mathit{subst}(\{\texttt{with}\ \{x\ e\}\ eB\}, x, v) &= \{\texttt{with}\ \{x\ \mathit{subst}(e, x, v)\}\ eB\} \\
\mathit{subst}(\{\texttt{with}\ \{y\ e\}\ eB\}, x, v) &= \{\texttt{with}\ \{y\ \mathit{subst}(e, x, v)\}\ \mathit{subst}(eB, x, v)\} \\
&\qquad \text{if } x \neq y
\end{aligned}
$$

Substitution, for WAE abstract syntax

$$
\begin{aligned}
\mathit{subst}((\mathsf{num}\ n), x, v) &= (\mathsf{num}\ n) \\
\mathit{subst}((\mathsf{id}\ x), x, v) &= v \\
\mathit{subst}((\mathsf{id}\ y), x, v) &= (\mathsf{id}\ y) \quad \text{if } x \neq y \\
\mathit{subst}((\mathsf{add}\ eL\ eR), x, v) &= (\mathsf{add}\ \mathit{subst}(eL, x, v)\ \mathit{subst}(eR, x, v)) \\
\mathit{subst}((\mathsf{sub}\ eL\ eR), x, v) &= (\mathsf{sub}\ \mathit{subst}(eL, x, v)\ \mathit{subst}(eR, x, v)) \\
\mathit{subst}((\mathsf{with}\ x\ e\ eB), x, v) &= (\mathsf{with}\ x\ \mathit{subst}(e, x, v)\ eB) \\
\mathit{subst}((\mathsf{with}\ y\ e\ eB), x, v) &= (\mathsf{with}\ y\ \mathit{subst}(e, x, v)\ \mathit{subst}(eB, x, v)) \\
&\qquad \text{if } x \neq y
\end{aligned}
$$

Figure 1 Substitution for the WAE language.

I'm leaving blanks for things I don't know yet, because I'm writing the derivation tree, starting from the root.

Let's derive the first premise, $(\mathsf{num}\ 1) \Downarrow \cdots$. Looking at our rules, we need to apply Eval-num.

$$\dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad \mathit{subst}((\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x)), x, (\mathsf{num}\ 1)) \Downarrow \_\_\_}{(\mathsf{with}\ x\ (\mathsf{num}\ 1)\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))) \Downarrow \_\_\_}\ \text{Eval-with}$$

That 1 gave us the missing argument to *subst*, so we can use the definition of *subst*:

$$\dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad (\mathsf{add}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 1)) \Downarrow \_\_\_}{(\mathsf{with}\ x\ (\mathsf{num}\ 1)\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))) \Downarrow \_\_\_}\ \text{Eval-with}$$

For the remaining premise, we need to apply the same old rules from the AE language:

$$\dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad \dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad \dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num}}{(\mathsf{add}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 1)) \Downarrow 1 + 1}\ \text{Eval-add}}{(\mathsf{with}\ x\ (\mathsf{num}\ 1)\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))) \Downarrow \_\_\_}\ \text{Eval-with}$$

By arithmetic,

$$\dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad \dfrac{\dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num} \qquad \dfrac{}{(\mathsf{num}\ 1) \Downarrow 1}\ \text{Eval-num}}{(\mathsf{add}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 1)) \Downarrow 2}\ \text{Eval-add}}{(\mathsf{with}\ x\ (\mathsf{num}\ 1)\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))) \Downarrow 2}\ \text{Eval-with}$$

So, we have successfully showed that the meaning of adding x and x with x being 1 is 2!

Exercise 2.
Have we defined enough new rules, or did we miss something? Give it some thought, then turn the page...

4.2 Open expressions not welcome

We added two new constructs (with and id) to our syntax, but only one rule to our evaluation semantics! That doesn't seem right.

In fact, it *is* right (or at least reasonable), because some expressions "pass" the EBNF (for concrete syntax, or the **define-type** for abstract syntax), but still don't really make sense. The WAE

{+ y 3} (abstract syntax: (plus (id y) (num 3)))

has an identifier y that isn't inside a with, and is a "free identifier" (or "free variable"). We don't know what y is supposed to be, so we can't evaluate y, and therefore can't evaluate {+ y 3}.

Definition 3. An expression is **open** if it has free identifiers. An expression is **closed** if it has no free identifiers (equivalently, if all identifier instances are bound).

As long as all instances are bound (by with), our rule Eval-with always substitutes for them; we don't introduce free identifiers as we evaluate. (In a different kind of course, like CPSC 509, we would *prove* that these evaluation rules don't introduce free identifiers.)

In other words, there is a gap between WAEs that pass the EBNF, and WAEs that mean something (evaluate to something). This gap exists in many real programming languages; it didn't exist in the AE language because it was so simple. For example, in a statically-typed language like Java, there are plenty of programs that pass the Java EBNF, but can't be compiled because of type errors. In DrRacket with the PLAI language, if you click "Check Syntax", it will complain about a **type-case** with a missing branch, even though the **type-case** matches the EBNF. (It should probably be called "Check Syntax, And Some Other Stuff".)

We could still give a rule for id, but it's going to have to look a little different from our other rules.

$$\frac{}{\text{(id x)} \Downarrow \text{free-variable-error}}\ \text{Eval-free-identifier}$$

This rule does *not* say that (id x) evaluates to an error: we didn't write the "evaluates to" symbol (⇓). It says that (id x) generates a free variable error. In our interpreter, we can implement this rule with the (error ...) function.

Question: Could we say that (id x) evaluates to itself?

$$\frac{}{\text{(id x)} \Downarrow \text{(id x)}}$$

This would say that (id x) evaluates, but it doesn't evaluate to a number, which isn't consistent with our other rules. Is it *wrong*? That depends on what kind of language you want. If we wanted a language in which free identifiers stood for unknown quantities, then it could make sense to say that (id x) evaluates to itself.

For the moment, the languages we're building will see a free identifier as a mistake (perhaps a misspelling of a with-bound identifier).