Formal Semantics

CAS CS 320: Concepts of Programming Languages

October 20, 2026

Semantics refers generally to the study of meaning; the "formal" part of *formal semantics* refers to the use of mathematics as a framework for studying meaning. In linguistics and philosophy, meaning is delicate business. We have to consider questions like: *How does a word or phrase get its meaning? What is the nature of reference, i.e., how does a word refer to an object? What makes a sentence true? What is truth?* In PL theory, semantics is the fun part; it's how we say what our programs "do."

There are two kinds of formal semantics that come up in PL theory: ***denotational semantics*** and ***operational semantics***. We'll be focused primarily on the latter, but we should say something briefly about the former. Denotational semantics is about interpreting programs as mathematical objects, i.e., letting a program to "denote" a mathematical object. We use the notation $[\![p]\!]$ to refer to the denotation of the program p.

Denotational semantics for arithmetic expressions. As a basic case, we can take arithemtic expressions to denote integers. That is, we can define denotations inductively on the structure of arithmetic expressions:

$$\begin{aligned}
[\![e_1 \texttt{ + } e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![e_1 \texttt{ - } e_2]\!] &= [\![e_1]\!] - [\![e_2]\!] \\
[\![e_1 \texttt{ * } e_2]\!] &= [\![e_1]\!] \times [\![e_2]\!] \\
[\![n]\!] &= n
\end{aligned}$$

where $n \in \mathbb{Z}$, i.e., n is an integer literal. So, for example,

$$\begin{aligned}
[\![\texttt{1 + 2 * 3}]\!] &= [\![\texttt{1}]\!] + [\![\texttt{2 * 3}]\!] \\
&= 1 + [\![\texttt{2}]\!] \times [\![\texttt{3}]\!] \\
&= 1 + 2 \times 3 \\
&= 7
\end{aligned}$$

i.e., the expression `1 + 2 * 3` denotes the integer 7.[1]

This is all fine and good in this basic case, but things get complicated fast, e.g., what would be a reasonable denotation for `fun x -> x`? Our denotations

[1] We'll generally blur the distinction between a value and its corresponding literal, e.g., here we take 7 and `7` to be the same thing.

would need to include some notion of functions, but then what would be the domain of the function $[\![\texttt{fun x -> x}]\!]$? We won't concern ourselves with this, but hopefully this is enough food for thought for the intellectually hungry.

Having a denotational semantics for a PL is important foundationally because it ensures that the programs we write reasonably refer to things (even if those things are abstract objects). But PL folks are often interested not only in what programs mean, but also in what they "do." It's more common to specify a PL with an operational semantics, i.e., a description of how programs behave computationally. There are two forms of operational semantics we'll consider: ***small-step semantics*** and ***big-step semantics***. These are essentially as they sound: a small-step semantics describes the step-by-step computational behavior of a program whereas a big-step semantics describes how to compute the value of a program, given the values of its constituent parts.[2] In either case, operational semantics are presented as inference systems over semantic judgments.

Before moving on, I'd like to note that it's also common to see the distinction between ***static semantics*** and ***dynamic semantics***. We should read "static" as "at compile time" or "before evaluation." Said another way, static semantic information is what we can learn about a program "without running it." There are many forms of static semantic information—the study of which falls under the umbrella of *static program analysis*—but for all intents and purposes, we will take "static" to refer to typing and "dynamic" to refer to operational semantics.

Small-step operational semantics. This kind of semantics is presented as an inference system over judgments of the form

$$\mathcal{C}_1 \longrightarrow \mathcal{C}_2$$

where $\mathcal{C}_1$ and $\mathcal{C}_2$ are called ***configurations***. We read this judgment as "the configuration $\mathcal{C}_1$ evaluates to the configuration $\mathcal{C}_2$ in a single step." We'll also occasionally say "$\mathcal{C}_1$ single-step evaluates to $\mathcal{C}_2$." A configuration consists of two parts: a program and auxiliary data we call ***state***. We write a configuration as

$$\langle\ \mathcal{S}\ ,\ p\ \rangle$$

where $\mathcal{S}$ is our state and p is our program. Exactly of what our state consists depends on the PL we're specifying (and how we're choosing to specify it). Part of presenting a semantics is specifying what state is required in a configuration. It's often unnecessary to have any state at all (as with functional PLs under substitution semantics (more on that later)) so we'll sometimes just write p instead of $\langle \varnothing, p \rangle$ for configurations without state.

A subtlety about abstract syntax. Evaluation happens after parsing (and type checking) so by the time we're evaluating a program, we already know it's well-formed. This means when we specify a semantics—and, in particular, reason about programs in a configuration—we're really reasoning about abstract

[2] We can think of computing values as taking "big steps" to final outputs, without worrying as much about the individual steps that got us there.

```
<expr> ::= <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <int>
```

Figure 1: Abstract syntax for arithmetic expressions

syntax trees (ASTs). That said, it's easier to write down inference rules using ***concrete syntax*** (i.e., the actual syntax we'd use when programming) so we don't have to draw ASTs in our inference rules (that'd become notationally clunky). This is just to say that, even though we write semantic rules using concrete syntax, we should really think about this as ***abstract syntax***, standing for an AST. This is why we don't deal with parentheses or precedence when writing semantic rules (though we'll still *use* parentheses and precedence rules to distinguish between ASTs in our abstract syntax).

Small-step semantics for arithmetic expressions. We fix a grammar for the abstract syntax of arithmetic expressions in Figure 1.[3] Figure 2 contains a small-step semantics for expressions in this grammar. Since we don't need any stateful data, we omit it from our configurations. Pay close attention to rules like (addLits); on the left-hand side of the arrow in the conclusion we have "`+`" whereas on the right-hand side we have "+." These rules are basically the "junctions" between the syntax and its meaning, where we "transfer" from plus-sign the symbol to plus-sign the mathematical operation.

With this semantics, we can now formally state and prove that one configuration evaluates to another in a single step. We do this by writing derivations in the above inference system.

Exercise 1. *Give a derivation for the judgment* `(2 + 3) * 4` $\longrightarrow$ `5 * 4` *using the rules in Figure 2.*

Exercise 2. *Determine an expression e such that* `2 * 3 * 4` $\longrightarrow e$*, and provide a derivation of this judgment (using the rules in Figure 2).*

This semantics also has a not-so-nice property: it's *nondeterministic*. We say that a small-step semantics is ***deterministic*** if for every configuration $\mathcal{C}$, there is a unique configuration $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow \mathcal{C}'$. Having a deterministic semantics amounts to fixing an ***evaluation strategy***, i.e., a choice of which step to take, in the case there are multiple choices.

Exercise 3. *Demonstrate that the semantics in Figure 2 is nondeterministic.*

The issue is that the semantics in Figure 2, given an expression of the form e_1 `+` e_2 or e_1 `-` e_2 or e_1 `*` e_2, has no opinion about *which* operand (e_1 or e_2)

[3] Again, because this is abstract syntax, we don't include parentheses or deal with precedence in the grammar. In particular, this grammar is ambiguous.

$$\frac{e_1 \longrightarrow e_2}{e_1 \texttt{ + } e \longrightarrow e_2 \texttt{ + } e}\ \textsc{addRedLeft} \qquad \frac{e_1 \longrightarrow e_2}{e \texttt{ + } e_1 \longrightarrow e \texttt{ + } e_2}\ \textsc{addRedRight}$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ + } n \longrightarrow m + n}\ \textsc{addLits} \qquad \frac{e_1 \longrightarrow e_2}{e_1 \texttt{ - } e \longrightarrow e_2 \texttt{ - } e}\ \textsc{subRedLeft}$$

$$\frac{e_1 \longrightarrow e_2}{e \texttt{ - } e_1 \longrightarrow e \texttt{ - } e_2}\ \textsc{subRedRight} \qquad \frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ - } n \longrightarrow m - n}\ \textsc{subLits}$$

$$\frac{e_1 \longrightarrow e_2}{e_1 \texttt{ * } e \longrightarrow e_2 \texttt{ * } e}\ \textsc{mulRedLeft} \qquad \frac{e_1 \longrightarrow e_2}{e \texttt{ * } e_1 \longrightarrow e \texttt{ * } e_2}\ \textsc{mulRedRight}$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ * } n \longrightarrow m \times n}\ \textsc{mulLits}$$

Figure 2: A small-step semantics for arithmetic expressions

to evaluate first. We can address this by "breaking the symmetry" in the rules. That is, we can enforce that we're only allowed to evaluated the right operand once the left operand is completely evaluated. This in essence requires us to include side-conditions that state as much (see Figure 3). The choice to evaluate the left operand before the right is arbitrary, but it aligns with standard practice.

Reverse Polish Notation. One more example, this time with state. We fix a grammar for arithmetic expressions in reverse Polish notation in Figure 4. As a reminder, this grammar says that a program (<prog>) is a sequence of zero or more commands (<com>) and a command is either an operator (<op>) or an integer literal (<int>). We give a small-step semantics for this grammar that maintains a *stack* (i.e., a list) of integers as state in Figure 5. Note that this semantics is deterministic; there is one rule per command, and commands are processed one at a time. One notable feature of this example is that all single-step derivations are trivial since all inference rules are axioms. Another notable feature as compared to non-RPN arithmetic is that the notion of a well-formed RPN arithmetic expression is far weaker; there are many configurations with well-formed programs (w.r.t. the grammar in Figure 4) that cannot be single-step evaluated.

Exercise 4. *Determine a configuration for the RPN arithmetic semantics in Figure 5 that has a nonemtpy stack, a nonempty program, and cannot be single-step evaluated (i.e., there is no configuration it evaluates to).*

Multi-step evaluation. In the context of (deterministic) small-step semantics, computation is repeated single-step evaluation. To model this, every small-step semantics comes with an additional form of judgment "$\mathcal{C}_1 \longrightarrow^\star \mathcal{C}_2$" which we

$$\frac{e_1 \longrightarrow e_2}{e_1 \texttt{ + } e \longrightarrow e_2 \texttt{ + } e}\ \textsc{addRedLeft} \qquad \frac{m \in \mathbb{Z} \qquad e_1 \longrightarrow e_2}{m \texttt{ + } e_1 \longrightarrow m \texttt{ + } e_2}\ \textsc{addRedRight}$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ + } n \longrightarrow m + n}\ \textsc{addLits} \qquad \frac{e_1 \longrightarrow e_2}{e_1 \texttt{ - } e \longrightarrow e_2 \texttt{ - } e}\ \textsc{subRedLeft}$$

$$\frac{m \in \mathbb{Z} \qquad e_1 \longrightarrow e_2}{m \texttt{ - } e_1 \longrightarrow m \texttt{ - } e_2}\ \textsc{subRedRight} \qquad \frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ - } n \longrightarrow m - n}\ \textsc{subLits}$$

$$\frac{e_1 \longrightarrow e_2}{e_1 \texttt{ * } e \longrightarrow e_2 \texttt{ * } e}\ \textsc{mulRedLeft} \qquad \frac{m \in \mathbb{Z} \qquad e_1 \longrightarrow e_2}{m \texttt{ * } e_1 \longrightarrow m \texttt{ * } e_2}\ \textsc{mulRedRight}$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{m \texttt{ * } n \longrightarrow m \times n}\ \textsc{mulLits}$$

Figure 3: A deterministic small-step semantics for arithmetic

$$\begin{aligned} \texttt{<prog>} &= \{\texttt{<com>}\} \\ \texttt{<com>} &= \texttt{<op>} \mid \texttt{<int>} \\ \texttt{<op>} &= \texttt{+} \mid \texttt{-} \mid \texttt{*} \end{aligned}$$

Figure 4: Abstract syntax for arithemtic expressions in reverse Polish notation

$$\frac{n \in \mathbb{Z}}{\langle\ \mathcal{S}\ ,\ n\ p\ \rangle \longrightarrow \langle\ n :: \mathcal{S}\ ,\ p\ \rangle}\ \textsc{num}$$

$$\frac{}{\langle\ n :: m :: \mathcal{S}\ ,\ \texttt{+}\ p\ \rangle \longrightarrow \langle\ (m+n) :: \mathcal{S}\ ,\ p\ \rangle}\ \textsc{add}$$

$$\frac{}{\langle\ n :: m :: \mathcal{S}\ ,\ \texttt{-}\ p\ \rangle \longrightarrow \langle\ (m-n) :: \mathcal{S}\ ,\ p\ \rangle}\ \textsc{sub}$$

$$\frac{}{\langle\ n :: m :: \mathcal{S}\ ,\ \texttt{*}\ p\ \rangle \longrightarrow \langle\ (m \times n) :: \mathcal{S}\ ,\ p\ \rangle}\ \textsc{mul}$$

Figure 5: A small-step semantics for arithemtic in reverse Polish notation

$$\frac{}{\mathcal{C} \longrightarrow^\star \mathcal{C}}\ \text{REFL} \qquad\qquad \frac{\mathcal{C}_1 \longrightarrow \mathcal{C}_2 \qquad \mathcal{C}_2 \longrightarrow^\star \mathcal{C}_3}{\mathcal{C}_1 \longrightarrow^\star \mathcal{C}_3}\ \text{STEP}$$

Figure 6: Multi-step semantic rules

read as "the configuration $\mathcal{C}_1$ evaluates to the configuration $\mathcal{C}_2$ in zero or more steps." This judgment comes with two inference rules (Figure 6), which we also take to be implicitly included in any small-step semantics. The first rule (refl)—short for "reflexivity"—says that a configuration evaluates to itself in zero ~~or more~~ steps.

Exercise 5. *State in English what the (step) rule says.*

Exercise 6. *Give a derivation of the judgment* `(2 + 3) * 4` $\longrightarrow^*$ `20` *using the rules in Figures 3 and 6.*

Exercise 7. *Determine an integer literal e such that* `2 * 3 * 4` $\longrightarrow^*$ e*, and provide a derivation of this judgment (using the rules in Figures 3 and 6).*

Exercise 8. *Determine the stack $\mathcal{S}$ such that*

$$\langle\ \varnothing\ ,\ \texttt{1 2 3 + +}\ \rangle \longrightarrow^\star \langle\ \mathcal{S}\ ,\ \epsilon\ \rangle$$

where $\varnothing$ is the empty stack and ϵ is the empty program. Additionally, give a derivation using the rules in Figures 5 and 6.

Exercise 9. *Give two distinct derivation for the judgment*

$$\texttt{(1 + 1) * (1 + 1)} \longrightarrow^\star \texttt{4}$$

using the nondeterministic rules in Figure 2.

Traces. The sequence of single-step evaluations required to get from one configuration to another may be quite long. And often it's not difficult to check by eye that a given single-step semantic judgment holds (e.g., in our RPN arithemtic semantics all single-step judgments are axioms). In this case, we may write a *compact* multi-step derivation, which is just the sequence of configurations such that each step in the sequence is a valid single-step evaluation. We call such a sequence a ***trace***. For example, the following is trace for the judgment

$$\texttt{(1 + 2) * 3 * (4 - 5 * 6)} \longrightarrow^\star \texttt{-54}$$

using the deterministic rules in Figure 3.

$$\begin{aligned}\texttt{(1 + 2) * 3 * (4 - 5 * 6)} &\longrightarrow \texttt{3 * 3 * (4 - 5 * 2)}\\ &\longrightarrow \texttt{9 * (4 - 5 * 2)}\\ &\longrightarrow \texttt{9 * (4 - 10)}\\ &\longrightarrow \texttt{9 * (-6)}\\ &\longrightarrow \texttt{-54}\end{aligned}$$

Exercise 10. *Rewrite both of your derivations from Exercise 9 as traces.*

Final and Stuck Configurations. Since computation is repeated single-step evaluation, we're most interested in the problem: *Given a configuration $\mathcal{C}$, determine a configuration $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow^\star \mathcal{C}'$ and $\mathcal{C}'$ cannot be further evaluated.* We call $\mathcal{C}'$ an ***irreducible configuration***. But, as noted above, it may be that not all irreducible configurations are considered "successes." When specifying a small-step semantics, in addition to the inference rules we also have to provide what we consider to be a successful configurations, which we call ***final configurations***. All other irreducible configurations we call ***stuck configurations***, the idea being that we got stuck during evaluation, and can't make further progress, but didn't "finish" evaluating. Final configurations are usually specified by fixing a subset of the abstract syntax for the programs that we consider to be fully evaluated. In the case of our RPN arithemtic semantics, the final configurations are those whose programs are empty. We can specify this using a BNF grammar:

<final> ::= ϵ

where ϵ is the empty program. It's also common to restrict the state in both starting and final configurations, e.g., in the RPN arithmetic semantics we might require the starting configurations to have empty stacks and final configurations to have singleton stacks.[4]

Exercise 11. *Demonstrate that `2 3 4 + 5 + + * 6` multi-step evaluates to a stuck configuration starting at the empty stack by giving a trace.*

Exercise 12. *Demonstrate that `2 3 4 + 5 + + * 6` multi-step evaluates to a final configuration starting at the stack* $1 :: \varnothing$ *by giving a trace.*

Things are simpler in the case of expression-based semantics with no state (as in our initial two small-step semantics) but the terminology changes. We call irreducible expressions ***normal forms***, i.e., an expression e is a normal form if there is no expression e' such that $e \longrightarrow e'$. For example, you'll notice that the small-step semantics in Figures 2 and 3 have no rules for integer literals. This is on purpose; integer literals are normal forms, they cannot be further reduced. We call our "successful" normal forms ***values*** and our "unsuccessful" ones ***stuck terms***. As above, we specify values by designating a subset of the abstract syntax. For example, the semantics for arithmetic expressions includes the following BNF production rule for values:

<val> ::= <int>

That is, the values of the semantics are exactly the integer literals.

Exercise 13. *Demonstrate that the semantics specified in Figure 2 and 3 have no stuck terms.*

[4] That said, we will tend to not do this in this course.

All of the semantics we've considered thus far has a nice property: they're ***normalizing***. This is to say that every configuration multi-step evaluates to an irreducible configuration. Not all systems we consider will have this property; in fact it's often *undesirable* to have this property because this implies the PL is not Turing-complete (i.e., it can't "do" all possible computation). Regardless, this means three things can happen when you evaluate a program:

1. evaluation terminates successfully, i.e., multi-step evaluation ends in a final configuration;
2. evaluation terminates unsuccessfully, i.e., multi-step evaluation ends in a stuck configuration;
3. evaluation runs forever, i.e., multi-step evaluation never ends; we sometimes say here that evaluation ***diverges***.

Exercise 14. ***(Challenge)*** *Prove all the above semantics are normalizing. Hint: Use induction. Determine a measure on expressions/configurations that strictly decreases with every single-step evaluation.*

Small-Step Semantics in OCaml. It's very simple to translate all the theory we've discussed above into OCaml code. We'll start with the deterministic arithmetic semantics. The abstract syntax corresponds to an ADT for expressions:

```
type expr =
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Int of int
```

Single-step evaluation then becomes pattern matching on expressions, turning each rule into a match case:

```
let single_step_eval e =
  match e with
  | Add(Int m, Int n) -> Int (m + n)
  | Add(Int e, e1) ->
    let e2 = single_step_eval e1 in
    Add (Int e, e2)
  | Add (e1, e) ->
    let e2 = single_step_eval e1 in
    Add (e2, e)
  | Sub(Int m, Int n) -> Int (m - n)
  | Sub(Int e, e1) ->
```

```
    let e2 = single_step_eval e1 in
    Sub (Int e, e2)
  | Sub (e1, e) ->
    let e2 = single_step_eval e1 in
    Sub (e2, e)
  | Mul (Int m, Int n) -> Int (m * n)
  | Mul (Int e, e1) ->
    let e2 = single_step_eval e1 in
    Mul (Int e, e2)
  | Mul (e1, e) ->
    let e2 = single_step_eval e1 in
    Mul (e2, e)
  | Int n -> failwith "cannot single-step evaluate an int"
```

Note the order in which the rules are included; this is because of the fall-through behavior of pattern matching.

Exercise 15. *Make sure you understand this point.*

Finally, we can distinguish our values via a Boolean predicate and use this as the termination condition for a recursive multi-step evaluator.

```
let is_value e = match e with | Int _ -> true | _ -> false

let rec multi_step_eval e =
  if is_value e
  then e
  else multi_step_eval (single_step eval e)
```

The multi-step evaluator basically implements the (refl) and (step) rules from Figure 6.

Exercise 16. *Suppose we want to introduce if-expressions into our arithmetic language so we could write something like* `if 2 then 0 else 1 + 1`*, where the condition fails if it evaluates to* `0`*, and otherwise succeeds. Extend the abstract syntax and the small-step evaluation rules to account for this. The final system you present should be deterministic and normalizing.*

Exercise 17. *Extend the above OCaml implementation to include if-expressions as in the previous problem.*

The translation is similarly simple for RPN arithmetic.[5] The abstract syntax:

[5]This is, in a sense, one of the pitches of functional PLs: they're very well suited for designing and implementing PLs.

```
type op = Add | Sub | Mul
type com = Op of op | Int of int
type prog = com list
```

Again, note how similar this is to the BNF grammar given in Figure 4. We can also represent our configurations in types if we want to:

```
type stack = int list
type config = stack * prog
```

Next, the single-step evaluator:

```
let single_step_eval (s, p) =
  match s, p with
  | s, Int n :: p -> n :: s, p
  | n :: m :: s, Op Add :: p -> (m + n) :: s, p
  | n :: m :: s, Op Sub :: p -> (m - n) :: s, p
  | n :: m :: s, Op Mul :: p -> (m * n) :: s, p
  | _, Op _ :: p ->
    failwith "cannot apply operator to fewer than two arguments"
  | s, [] ->
    failwith "cannot single-step evaluate the empty program"
```

And the multi-step evaluator:

```
let is_final (s, p) = p = []

let rec multi_step_eval c =
  if is_final c
  then c
  else multi_step_eval (single_step_eval c)
```

Big-step operational semantics. This kind of semantics is presented as an inference system over judgments of the form

$$\mathcal{C}_1 \Downarrow \mathcal{C}_2$$

where $\mathcal{C}_1$ and $\mathcal{C}_2$ are configuration and $\mathcal{C}_2$ is final. We read this judgment simply as "the configuration $\mathcal{C}_1$ (big-step) evaluates to $\mathcal{C}_2$." In the case of expressions, "$e \Downarrow v$" is read "the expression e evaluates to the value v".[6] Figure 7 has a big-step semantics for arithmetic. The rules are meant to capture a "natural" idea: to evaluate an expression, we evaluate it's sub-expressions, and then somehow put those values together.

[6] We present 320Caml at the beginning of the semester using big-step semantics.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \texttt{ + } e_2 \Downarrow v_1 + v_2}\ \textsc{add} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \texttt{ - } e_2 \Downarrow v_1 - v_2}\ \textsc{sub}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \texttt{ * } e_2 \Downarrow v_1 \times v_2}\ \textsc{mul} \qquad \frac{n \in \mathbb{Z}}{n \Downarrow n}\ \textsc{intLit}$$

Figure 7: A big-step semantics for arithmetic

Exercise 18. *Give a derivation of* `(1 + 2) * (3 - 4)` $\Downarrow$ `-3` *using the big-step rules in Figure 7.*

There is a degree of freedom in the statement of these rules w.r.t. ***order of evaluation***. In the small-step semantics above, we fixed the order of evaluation in the rules themselves, e.g., in the determintic arithmetic rules (Figure 3) we decided the left operand is evaluated before the right, and then designed the rules to reflect this. We can't do this in big-step semantics; instead, the order of evaluation is determined by *the order (from left to right) of the premises of the rule*. So, for example, the following rule is different from the (add) rule given in Figure 7.

$$\frac{e_2 \Downarrow v_2 \qquad e_1 \Downarrow v_1}{e_1 \texttt{ + } e_2 \Downarrow v_1 + v_2}\ \textsc{add}'$$

This rule states that the right operand should be evaluated before the left.

Big-step semantics in OCaml. Big-step semantics are sometimes called "natural" semantics because they translate very easily to recursive evaluator implementations. We use the same `expr` type as above.

```
let rec big_step_eval (e : expr) : int =
  match e with
  | Add (e1, e2) ->
    let v1 = big_step_eval e1 in
    let v2 = big_step_eval e2 in
    v1 + v2
  | Sub (e1, e2) ->
    let v1 = big_step_eval e1 in
    let v2 = big_step_eval e2 in
    v1 - v2
  | Mul (e1, e2) ->
    let v1 = big_step_eval e1 in
    let v2 = big_step_eval e2 in
```

```
    v1 * v2
  | Int n -> n
```

Every match-case corresponds almost one-to-one to a rule in the semantics. One final subtlety: it's often easier in a big-step semantics evaluator to make a distinction between values and expressions. Here, rather than evaluating to an integer literal, we evaluate directly to an integer. We will often introduce a separate `value` type in the our evaluators moving forward.

Exercise 19. *Suppose we wanted to introduce if-expressions* ***and Booleans*** *into our arithmetic language. Extend the abstract syntax and the big-step evaluation rules to account for this.*

Exercise 20. *Extend the above OCaml implementation to include if-expressions as in the previous problem.*

The relationship between small-step and big-step semantics. Not all PLs can have big-step semantics. Sometimes this is by virtue of the way the PL is designed, e.g., there is no natural way to give our RPN arithmetic a big step semantics.[7] Other times this is a theoretical limitation; we likely won't cover this, but designing PLs for concurrency often require more fine-grained semantics that cannot be captured accurately with a big-step semantics.

That said, we can often provide both a big-step and small-step semantics for a PL. When this is possible, we need to demonstrate that they're in some sense equivalent. Formally we say that a big-step semantics is ***adequate*** if $\mathcal{C}_1 \Downarrow \mathcal{C}_2$ implies $\mathcal{C}_1 \longrightarrow^\star \mathcal{C}_2$ and is ***(small-step) sound*** if $\mathcal{C}_1 \longrightarrow^\star \mathcal{C}_2$ and $\mathcal{C}_2$ is final implies $\mathcal{C}_1 \Downarrow \mathcal{C}_2$.[8]

Exercise 21. ***(Challenge)*** *Demonstrate that the big-step arithmetic rules in Figure 7 are adequate and sound w.r.t. the small-step rules in Figure 2.*

So if we can have both, *why choose one over the other?* Big step semantics tend to be easier to understand and reason about, but small-step semantics offer more fine-grained reasoning. This is particularly notable in the case of non-normalization; *we can't reason about stuck or divergent terms in a big-step semantics.* If evaluation of expression e is stuck or divergent, all we can say w.r.t. big-step semantics is that there is no value v such that $e \Downarrow v$. With small step semantics, we can analyze partial evaluations, i.e., we can look at intermediate expression e' such that $e \longrightarrow^\star e'$, which can often help us better understand the computational behavior of our PL. In particular, when we talk about type safety, we'll use small-step semantics for exactly this reason.

[7] It's possible, but it wouldn't be useful or enlightening.

[8] We'll come across another notion of soundness when we discuss *type soundness*. Unfortunately "soundness" is an overloaded term in PL theory and logic.