3 *Untyped Arithmetic Expressions*

To talk rigorously about type systems and their properties, we need to start by dealing formally with some more basic aspects of programming languages. In particular, we need clear, precise, and mathematically tractable tools for expressing and reasoning about the syntax and semantics of programs.

This chapter and the next develop the required tools for a small language of numbers and booleans. This language is so trivial as to be almost beneath consideration, but it serves as a straightforward vehicle for the introduction of several fundamental concepts—abstract syntax, inductive definitions and proofs, evaluation, and the modeling of run-time errors. Chapters 5 through 7 elaborate the same story for a much more powerful language, the untyped lambda-calculus, where we must also deal with name binding and substitution. Looking further ahead, Chapter 8 commences the study of type systems proper, returning to the simple language of the present chapter and using it to introduce basic concepts of static typing. Chapter 9 extends these concepts to the lambda-calculus.

3.1 Introduction

The language used in this chapter contains just a handful of syntactic forms: the boolean constants `true` and `false`, conditional expressions, the numeric constant `0`, the arithmetic operators `succ` (successor) and `pred` (predecessor), and a testing operation `iszero` that returns `true` when it is applied to `0` and `false` when it is applied to some other number. These forms can be summarized compactly by the following grammar.

The system studied in this chapter is the untyped calculus of booleans and numbers (Figure 3-2, on page 41). The associated OCaml implementation, called `arith` in the web repository, is described in Chapter 4. Instructions for downloading and building this checker can be found at `http://www.cis.upenn.edu/~bcpierce/tapl`.

t ::=		*terms:*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*

The conventions used in this grammar (and throughout the book) are close to those of standard BNF (cf. Aho, Sethi, and Ullman, 1986). The first line (`t ::=`) declares that we are defining the set of *terms,* and that we are going to use the letter `t` to range over terms. Each line that follows gives one alternative syntactic form for terms. At every point where the symbol `t` appears, we may substitute any term. The italicized phrases on the right are just comments.

The symbol `t` in the right-hand sides of the rules of this grammar is called a *metavariable*. It is a variable in the sense that it is a place-holder for some particular term, and "meta" in the sense that it is not a variable of the *object language*—the simple programming language whose syntax we are currently describing—but rather of the *metalanguage*—the notation in which the description is given. (In fact, the present object language doesn't even have variables; we'll introduce them in Chapter 5.) The prefix *meta-* comes from *meta-mathematics*, the subfield of logic whose subject matter is the mathematical properties of systems for mathematical and logical reasoning (which includes programming languages). This field also gives us the term *metatheory*, meaning the collection of true statements that we can make about some particular logical system (or programming language)—and, by extension, the study of such statements. Thus, phrases like "metatheory of subtyping" in this book can be understood as, "the formal study of the properties of systems with subtyping."

Throughout the book, we use the metavariable `t`, as well as nearby letters such as `s`, `u`, and `r` and variants such as $\mathtt{t}_1$ and $\mathtt{s}'$, to stand for terms of whatever object language we are discussing at the moment; other letters will be introduced as we go along, standing for expressions drawn from other syntactic categories. A complete summary of metavariable conventions can be found in Appendix B.

For the moment, the words *term* and *expression* are used interchangeably. Starting in Chapter 8, when we begin discussing calculi with additional syntactic categories such as *types*, we will use *expression* for all sorts of syntactic phrases (including term expressions, type expressions, kind expressions, etc.), reserving *term* for the more specialized sense of phrases representing

computations (i.e., phrases that can be substituted for the metavariable `t`).

A program in the present language is just a term built from the forms given by the grammar above. Here are some examples of programs, along with the results of evaluating them:

```
if false then 0 else 1;
```

```
▸ 1
```

```
iszero (pred (succ 0));
```

```
▸ true
```

Throughout the book, the symbol ▸ is used to display the results of evaluating examples. (For brevity, results will be elided when they are obvious or unimportant.) During typesetting, examples are automatically processed by the implementation corresponding to the formal system in under discussion (`arith` here); the displayed responses are the implementation's actual output.

In examples, compound arguments to `succ`, `pred`, and `iszero` are enclosed in parentheses for readability.[1] Parentheses are not mentioned in the grammar of terms, which defines only their *abstract syntax*. Of course, the presence or absence of parentheses makes little difference in the extremely simple language that we are dealing with at the moment: parentheses are usually used to resolve ambiguities in the grammar, but this grammar is does not have any ambiguities—each sequence of tokens can be parsed as a term in at most one way. We will return to the discussion of parentheses and abstract syntax in Chapter 5 (p. 52).

For brevity in examples, we use standard arabic numerals for numbers, which are represented formally as nested applications of `succ` to `0`. For example, `succ(succ(succ(0)))` is written as `3`.

The results of evaluation are terms of a particularly simple form: they will always be either boolean constants or numbers (nested applications of zero or more instances of `succ` to `0`). Such terms are called *values,* and they will play a special role in our formalization of the evaluation order of terms.

Notice that the syntax of terms permits the formation of some dubious-looking terms like `succ true` and `if 0 then 0 else 0`. We shall have more to say about such terms later—indeed, in a sense they are precisely what makes this tiny language interesting for our purposes, since they are examples of the sorts of nonsensical programs we will want a type system to exclude.

1. In fact, the implementation used to process the examples in this chapter (called `arith` on the book's web site) actually *requires* parentheses around compound arguments to `succ`, `pred`, and `iszero`, even though they can be parsed unambiguously without parentheses. This is for consistency with later calculi, which use similar-looking syntax for function application.

3.2 Syntax

There are several equivalent ways of defining the syntax of our language. We have already seen one in the grammar on page 24. This grammar is actually just a compact notation for the following inductive definition:

3.2.1 DEFINITION [TERMS, INDUCTIVELY]: The set of *terms* is the smallest set $\mathcal{T}$ such that

1. $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$;
2. if $\texttt{t}_1 \in \mathcal{T}$, then $\{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1\} \subseteq \mathcal{T}$;
3. if $\texttt{t}_1 \in \mathcal{T}$, $\texttt{t}_2 \in \mathcal{T}$, and $\texttt{t}_3 \in \mathcal{T}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}$. □

Since inductive definitions are ubiquitous in the study of programming languages, it is worth pausing for a moment to examine this one in detail. The first clause tells us three simple expressions that are in $\mathcal{T}$. The second and third clauses give us rules by which we can judge that certain compound expressions are in $\mathcal{T}$. Finally, the word "smallest" tells us that $\mathcal{T}$ has no elements besides the ones required by these three clauses.

Like the grammar on page 24, this definition says nothing about the use of parentheses to mark compound subterms. Formally, what's really going on is that we are defining $\mathcal{T}$ as a set of *trees,* not as a set of strings. The use of parentheses in examples is just a way of clarifying the relation between the linearized form of terms that we write on the page and the real underlying tree form.

A different shorthand for the same inductive definition of terms employs the two-dimensional *inference rule* format commonly used in "natural deduction style" presentations of logical systems:

3.2.2 DEFINITION [TERMS, BY INFERENCE RULES]: The set of terms is defined by the following rules:

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}}$$

□

The first three rules here restate the first clause of Definition 3.2.1; the next four capture clauses (2) and (3). Each rule is read, "If we have established the

statements in the premise(s) listed above the line, then we may derive the conclusion below the line." The fact that $\mathcal{T}$ is the *smallest* set satisfying these rules is often (as here) not stated explicitly.

Two points of terminology deserve mention. First, rules with no premises (like the first three above) are often called *axioms*. In this book, the term *inference rule* is used generically to include both axioms and "proper rules" with one or more premises. Axioms are usually written with no bar, since there is nothing to go above it. Second, to be completely pedantic, what we are calling "inference rules" are actually *rule schemas*, since their premises and conclusions may include metavariables. Formally, each schema represents the infinite set of *concete rules* that can be obtained by replacing each metavariable consistently by all phrases from the appropriate syntactic category—i.e., in the rules above, replacing each `t` by every possible term.

Finally, here is yet another definition of the same set of terms in a slightly different, more "concrete" style that gives an explicit procedure for *generating* the elements of $\mathcal{T}$.

3.2.3 DEFINITION [TERMS, CONCRETELY]: For each natural number i, define a set S_i as follows:

$$\begin{array}{lcl} S_0 & = & \varnothing \\ S_{i+1} & = & \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ & \cup & \{\texttt{succ}\ \texttt{t}_1, \texttt{pred}\ \texttt{t}_1, \texttt{iszero}\ \texttt{t}_1 \mid \texttt{t}_1 \in S_i\} \\ & \cup & \{\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i\}. \end{array}$$

Finally, let

$$S = \bigcup_i S_i.$$

S_0 is empty; S_1 contains just the constants; S_2 contains the constants plus the phrases that can be built with constants and just one `succ`, `pred`, `iszero`, or `if`; S_3 contains these and all phrases that can be built using `succ`, `pred`, `iszero`, and `if` on phrases in S_2; and so on. S collects together all the phrases that can be built in this way—i.e., all phrases built by some finite number of arithmetic and conditional operators, beginning with just constants. □

3.2.4 EXERCISE [⋆⋆]: How many elements does S_3 have? □

3.2.5 EXERCISE [⋆⋆]: Show that the sets S_i are *cumulative*—that is, that for each i we have $S_i \subseteq S_{i+1}$. □

The definitions we have seen characterize the same set of terms from different directions: Definitions 3.2.1 and 3.2.2 simply *characterize* the set as

the smallest set satisfying certain "closure properties"; Definition 3.2.3 shows how to actually *construct* the set as the limit of a sequence.

To finish off the discussion, let us verify that these two views actually define the same set. We'll do the proof in quite a bit of detail, to show how all the pieces fit together.

3.2.6 PROPOSITION: $\mathcal{T} = S$. □

Proof: $\mathcal{T}$ was defined as the smallest set satisfying certain conditions. So it suffices to show (a) that S satisfies these conditions, and (b) that any set satisfying the conditions has S as a subset (i.e., that S is the *smallest* set satisfying the conditions).

For part (a), we must check that each of the three conditions in Definition 3.2.1 holds of S. First, since $S_1 = \{\texttt{true}, \texttt{false}, \texttt{0}\}$, it is clear that the constants are in S. Second, if $\texttt{t}_1 \in S$, then (since $S = \bigcup_i S_i$) there must be some i such that $\texttt{t}_1 \in S_i$. But then, by the definition of S_{i+1}, we must have $\texttt{succ t}_1 \in S_{i+1}$, hence $\texttt{succ t}_1 \in S$; similarly, we see that $\texttt{pred t}_1 \in S$ and $\texttt{iszero t}_1 \in S$. Third, if $\texttt{t}_1 \in S$, $\texttt{t}_2 \in S$, and $\texttt{t}_3 \in S$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in S$, by a similar argument.

For part (b), suppose that some set S' satisfies the three conditions in Definition 3.2.1. We will argue, by complete induction on i, that every $S_i \subseteq S'$, from which it clearly follows that $S \subseteq S'$.

Suppose that $S_j \subseteq S'$ for all $j < i$; we must show that $S_i \subseteq S'$. Since the definition of S_i has two clauses (for $i = 0$ and $i > 0$), there are two cases to consider. If $i = 0$, then $S_i = \emptyset$; but $\emptyset \subseteq S'$ trivially. Otherwise, $i = j + 1$ for some j. Let $\texttt{t}$ be some element of S_{j+1}. Since S_{j+1} is defined as the union of three smaller sets, $\texttt{t}$ must come from one of these sets; there are three possibilities to consider. (1) If $\texttt{t}$ is a constant, then $\texttt{t} \in S'$ by condition 1. (2) If $\texttt{t}$ has the form $\texttt{succ t}_1$, $\texttt{pred t}_1$, or $\texttt{iszero t}_1$, for some $\texttt{t}_1 \in S_j$, then, by the induction hypothesis, $\texttt{t}_1 \in S'$, and so, by condition (2), $\texttt{t} \in S'$. (3) If $\texttt{t}$ has the form $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3$, for some $\texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i$, then again, by the induction hypothesis, $\texttt{t}_1$, $\texttt{t}_2$, and $\texttt{t}_3$ are all in S', and, by condition 3, so is $\texttt{t}$.

Thus, we have shown that each $S_i \subseteq S'$. By the definition of S as the union of all the S_i, this gives $S \subseteq S'$, completing the argument. □

It is worth noting that this proof goes by *complete* induction on the natural numbers, not the more familiar "base case / induction case" form. For each i, we suppose that the desired predicate holds for all numbers strictly less than i and prove that it then holds for i as well. In essence, *every* step here is an induction step; the only thing that is special about the case where $i = 0$ is that the set of smaller values of i, for which we can invoke the induction hypothesis, happens to be empty. The same remark will apply to most induction

proofs we will see throughout the book—particularly proofs by "structural induction."

3.3 Induction on Terms

The explicit characterization of the set of terms $\mathcal{T}$ in Proposition 3.2.6 justifies an important principle for reasoning about its elements. If $\mathtt{t} \in \mathcal{T}$, then one of three things must be true about $\mathtt{t}$: (1) $\mathtt{t}$ is a constant, or (2) $\mathtt{t}$ has the form $\mathtt{succ}\ \mathtt{t}_1$, $\mathtt{pred}\ \mathtt{t}_1$, or $\mathtt{iszero}\ \mathtt{t}_1$ for some *smaller* term $\mathtt{t}_1$, or (3) $\mathtt{t}$ has the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ for some *smaller* terms $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$. We can put this observation to work in two ways: we can give *inductive definitions* of functions over the set of terms, and we can give *inductive proofs* of properties of terms. For example, here is a simple inductive definition of a function mapping each term $\mathtt{t}$ to the set of constants used in $\mathtt{t}$.

3.3.1 DEFINITION: The set of constants appearing in a term $\mathtt{t}$, written $\mathit{Consts}(\mathtt{t})$, is defined as follows:

$$
\begin{array}{lcl}
\mathit{Consts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{Consts}(\mathtt{0}) & = & \{\mathtt{0}\} \\
\mathit{Consts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \cup \mathit{Consts}(\mathtt{t}_3) \quad \square
\end{array}
$$

Another property of terms that can be calculated by an inductive definition is their size.

3.3.2 DEFINITION: The *size* of a term $\mathtt{t}$, written $\mathit{size}(\mathtt{t})$, is defined as follows:

$$
\begin{array}{lcl}
\mathit{size}(\mathtt{true}) & = & 1 \\
\mathit{size}(\mathtt{false}) & = & 1 \\
\mathit{size}(\mathtt{0}) & = & 1 \\
\mathit{size}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) + 1
\end{array}
$$

That is, the size of $\mathtt{t}$ is the number of nodes in its abstract syntax tree. Similarly, the *depth* of a term $\mathtt{t}$, written $\mathit{depth}(\mathtt{t})$, is defined as follows:

$$
\begin{array}{lcl}
\mathit{depth}(\texttt{true}) & = & 1 \\
\mathit{depth}(\texttt{false}) & = & 1 \\
\mathit{depth}(\texttt{0}) & = & 1 \\
\mathit{depth}(\texttt{succ t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{pred t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{iszero t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3) & = & \max(\mathit{depth}(\texttt{t}_1), \mathit{depth}(\texttt{t}_2), \mathit{depth}(\texttt{t}_3)) + 1
\end{array}
$$

Equivalently, $\mathit{depth}(\texttt{t})$ is the smallest i such that $\texttt{t} \in S_i$ according to Definition 3.2.3. □

Here is an inductive proof of a simple fact relating the number of constants in a term to its size. (The property in itself is entirely obvious, of course. What's interesting is the form of the inductive proof, which we'll see repeated many times as we go along.)

3.3.3 LEMMA: The number of distinct constants in a term $\texttt{t}$ is no greater than the size of $\texttt{t}$ (i.e., $|\mathit{Consts}(\texttt{t})| \le \mathit{size}(\texttt{t})$). □

Proof: By induction on the depth of $\texttt{t}$. Assuming the desired property for all terms smaller than $\texttt{t}$, we must prove it for $\texttt{t}$ itself. There are three cases to consider:

Case: $\texttt{t}$ is a constant

Immediate: $|\mathit{Consts}(\texttt{t})| = |\{\texttt{t}\}| = 1 = \mathit{size}(\texttt{t})$.

Case: $\texttt{t} = \texttt{succ t}_1$, $\texttt{pred t}_1$, or $\texttt{iszero t}_1$

By the induction hypothesis, $|\mathit{Consts}(\texttt{t}_1)| \le \mathit{size}(\texttt{t}_1)$. We now calculate as follows: $|\mathit{Consts}(\texttt{t})| = |\mathit{Consts}(\texttt{t}_1)| \le \mathit{size}(\texttt{t}_1) < \mathit{size}(\texttt{t})$.

Case: $\texttt{t} = \texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3$

By the induction hypothesis, $|\mathit{Consts}(\texttt{t}_1)| \le \mathit{size}(\texttt{t}_1)$, $|\mathit{Consts}(\texttt{t}_2)| \le \mathit{size}(\texttt{t}_2)$, and $|\mathit{Consts}(\texttt{t}_3)| \le \mathit{size}(\texttt{t}_3)$. We now calculate as follows:

$$
\begin{array}{lcl}
|\mathit{Consts}(\texttt{t})| & = & |\mathit{Consts}(\texttt{t}_1) \cup \mathit{Consts}(\texttt{t}_2) \cup \mathit{Consts}(\texttt{t}_3)| \\
& \le & |\mathit{Consts}(\texttt{t}_1)| + |\mathit{Consts}(\texttt{t}_2)| + |\mathit{Consts}(\texttt{t}_3)| \\
& \le & \mathit{size}(\texttt{t}_1) + \mathit{size}(\texttt{t}_2) + \mathit{size}(\texttt{t}_3) \\
& < & \mathit{size}(\texttt{t}).
\end{array}
$$
□

The form of this proof can be clarified by restating it as a general reasoning principle. For good measure, we include two similar principles that are often used in proofs about terms.

3.3.4 THEOREM [PRINCIPLES OF INDUCTION ON TERMS]: Suppose P is a predicate on terms.

Induction on depth:

If, for each term `s`,
given $P(\mathtt{r})$ for all `r` such that $depth(\mathtt{r}) < depth(\mathtt{s})$
we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all `s`.

Induction on size:

If, for each term `s`,
given $P(\mathtt{r})$ for all `r` such that $size(\mathtt{r}) < size(\mathtt{s})$
we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all `s`.

Structural induction:

If, for each term `s`,
given $P(\mathtt{r})$ for all immediate subterms `r` of `s`
we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all `s`. □

Proof: EXERCISE (⋆⋆). □

Induction on depth or size of terms is analogous to complete induction on natural numbers (2.4.2). Ordinary structural induction corresponds to the ordinary natural number induction principle (2.4.1) where the induction step requires that $P(n+1)$ be established from just the assumption $P(n)$.

Like the different styles of natural-number induction, the choice of one term induction principle over another is determined by which one leads to a simpler structure for the proof at hand—formally, they are inter-derivable. For simple proofs, it generally makes little difference whether we argue by induction on size, depth, or structure. As a matter of style, it is common practice to use structural induction wherever possible, since it works on terms directly, avoiding the detour via numbers.

Most proofs by induction on terms have a similar structure. At each step of the induction, we are given a term `t` for which we are to show some property P, assuming that P holds for all subterms (or all smaller terms). We do this by separately considering each of the possible forms that `t` could have (`true`, `false`, conditional, `0`, etc.), arguing in each case that P must hold for any `t` of this form. Since the only parts of this structure that vary from one inductive proof to another are the details of the arguments for the individual cases, it is common practice to elide the unvarying parts and write the proof as follows.

Proof: By induction on $\mathtt{t}$.

Case: $\mathtt{t} = \mathtt{true}$

... show $P(\mathtt{true})$ ...

Case: $\mathtt{t} = \mathtt{false}$

... show $P(\mathtt{false})$ ...

Case: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

... show $P(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3)$, using $P(\mathtt{t}_1)$, $P(\mathtt{t}_2)$, and $P(\mathtt{t}_3)$ ...

(And similarly for the other syntactic forms.) □

For many inductive arguments (including the proof of 3.3.3), it is not really worth writing even this much detail: in the base cases (for terms $\mathtt{t}$ with no subterms) $P(\mathtt{t})$ is immediate, while in the inductive cases $P(\mathtt{t})$ is obtained by applying the induction hypothesis to the subterms of $\mathtt{t}$ and combining the results in some completely obvious way. It is actually *easier* for the reader simply to regenerate the proof on the fly (by examining the grammar while keeping the induction hypothesis in mind) than to check a written-out argument. In such cases, simply writing "by induction on $\mathtt{t}$" constitutes a perfectly acceptable proof.

3.4 Semantic Styles

Having formulated the syntax of our language rigorously, we next need a similarly precise definition of how terms are evaluated—i.e., the *semantics* of the language. There are three basic approaches to formalizing semantics:

1. *Operational semantics* specifies the behavior of a programming language by defining a simple *abstract machine* for it. This machine is "abstract" in the sense that it uses the terms of the language as its machine code, rather than some low-level microprocessor instruction set. For simple languages, a *state* of the machine is just a term, and the machine's behavior is defined by a *transition function* that, for each state, either gives the next state by performing a step of simplification on the term or declares that the machine has halted. The *meaning* of a term $\mathtt{t}$ can be taken to be the final state that the machine reaches when started with $\mathtt{t}$ as its initial state.[2]

2. Strictly speaking, what we are describing here is the so-called *small-step* style of operational semantics, sometimes called *structural operational semantics* (Plotkin, 1981). Exercise 3.5.17 introduces an alternate *big-step* style, sometimes called *natural semantics* (Kahn, 1987), in which a single transition of the abstract machine evaluates a term to its final result.

It is sometimes useful to give two or more different operational semantics for a single language—some more abstract, with machine states that look similar to the terms that the programmer writes, others closer to the structures manipulated by an actual interpreter or compiler for the language. Proving that the behaviors of these different machines correspond in some suitable sense when executing the same program amounts to proving the correctness of an implementation of the language.

2. *Denotational semantics* takes a more abstract view of meaning: instead of just a sequence of machine states, the meaning of a term is taken to be some mathematical object, such as a number or a function. Giving denotational semantics for a language consists of finding a collection of *semantic domains* and then defining an *interpretation function* mapping terms into elements of these domains. The search for appropriate semantic domains for modeling various language features has given rise to a rich and elegant research area known as *domain theory*.

   One major advantage of denotational semantics is that it abstracts from the gritty details of evaluation and highlights the essential concepts of the language. Also, the properties of the chosen collection of semantic domains can be used to derive powerful laws for reasoning about program behaviors—laws for proving that two programs have exactly the same behavior, for example, or that a program's behavior satisfies some specification. Finally, from the properties of the chosen collection of semantic domains, it is often immediately evident that various (desirable or undesirable) things are impossible in a language.

3. *Axiomatic semantics* takes a more direct approach to these laws: instead of first defining the behaviors of programs (by giving some operational or denotational semantics) and then deriving laws from this definition, axiomatic methods take the laws *themselves* as the definition of the language. The meaning of a term is just what can be proved about it.

   The beauty of axiomatic methods is that they focus attention on the process of reasoning about programs. It is this line of thought that has given computer science such powerful ideas as *invariants*.

During the '60s and '70s, operational semantics was generally regarded as inferior to the other two styles—useful for quick and dirty definitions of language features, but inelegant and mathematically weak. But in the '80s, the more abstract methods began to encounter increasingly thorny technical problems,[3] and the simplicity and flexibility of operational methods came

3. The *bête noire* of denotational semantics turned out to be the treatment of nondeterminism and concurrency; for axiomatic semantics, it was procedures.

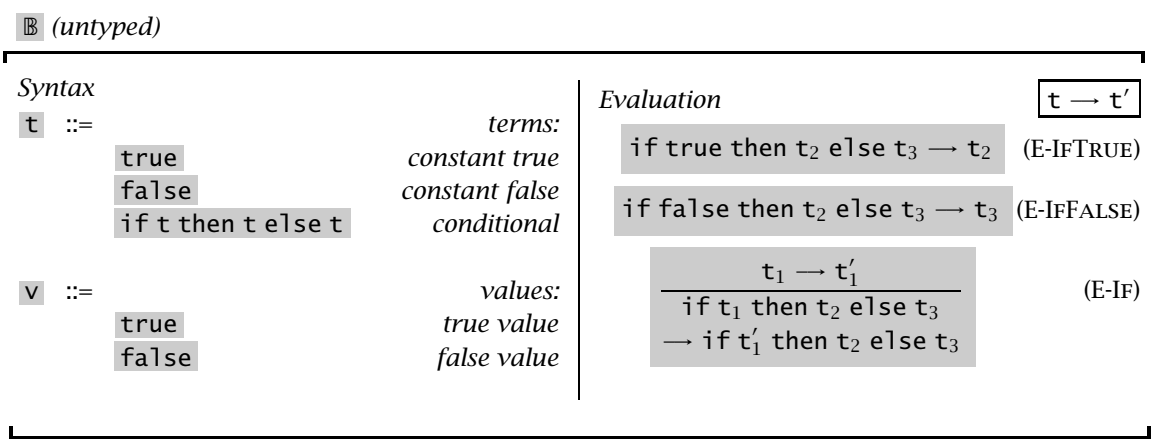

Figure 3-1: Booleans (B)

to seem more and more attractive by comparison—especially in the light of new developments in the area by a number of researchers, beginning with Plotkin's Structural Operational Semantics (1981), Kahn's Natural Semantics (1987), and Milner's work on CCS (1980; 1989; 1999), which introduced more elegant formalisms and showed how many of the powerful mathematical techniques developed in the context of denotational semantics could be transferred to an operational setting. Operational semantics has become an energetic research area in its own right and is often the method of choice for defining programming languages and studying their properties. It is used exclusively in this book.

3.5 Evaluation

Leaving numbers aside for the moment, let us begin with the operational semantics of just boolean expressions. Figure 3-1 summarizes the definition. We now examine its parts in detail.

The left-hand column of Figure 3-1 is a grammar defining two sets of expressions. The first is just a repetition (for convenience) of the syntax of terms. The second defines a subset of terms, called *values,* that are possible final results of evaluation. Here, the values are just the constants `true` and `false`. The metavariable `v` is used throughout the book to stand for values.

The right-hand column defines an *evaluation relation*[4] on terms, written

4. Some experts prefer to use the term *reduction* for this relation, reserving *evaluation* for the "big-step" variant described in Exercise 3.5.17, which maps terms directly to their final values.

$\texttt{t} \longrightarrow \texttt{t}'$ and pronounced "t evaluates to t′ in one step." The intuition is that, if t is the state of the abstract machine at a given moment, then the machine can make a step of computation and change its state to t′. This relation is defined by three inference rules (or, if you prefer, two axioms and a rule, since the first two have no premises).

The first rule, E-IFTRUE, says that, if the term being evaluated is a conditional whose guard is literally the constant `true`, then the machine can throw away the conditional expression and leave the then part, $\texttt{t}_2$, as the new state of the machine (i.e., the next term to be evaluated). Similarly, E-IFFALSE says that a conditional whose guard is literally `false` evaluates in one step to its else branch, $\texttt{t}_3$. The E- in the names of these rules is a reminder that they are part of the evaluation relation; rules for other relations will have different prefixes.

The third evaluation rule, E-IF, is more interesting. It says that, if the guard $\texttt{t}_1$ evaluates to $\texttt{t}_1'$, then the whole conditional $\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3$ evaluates to $\texttt{if t}_1'\ \texttt{then t}_2\ \texttt{else t}_3$. In terms of abstract machines, a machine in state $\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3$ can take a step to state $\texttt{if t}_1'\ \texttt{then t}_2\ \texttt{else t}_3$ if *another* machine whose state is just $\texttt{t}_1$ can take a step to state $\texttt{t}_1'$.

What these rules do *not* say is just as important as what they do say. The constants `true` and `false` do not evaluate to anything, since they do not appear as the left-hand sides of any of the rules. Moreover, there is no rule allowing the evaluation of a then- or else-subexpression of an `if` before evaluating the `if` itself: for example, the term

```
if true then (if false then false else false) else true
```

does not evaluate to `if true then false else true`. Our only choice is to evaluate the outer conditional first, using E-IF. This interplay between the rules determines a particular *evaluation strategy* for conditionals, corresponding to the familiar order of evaluation in common programming languages: to evaluate a conditional, we must first evaluate its guard; if the guard is itself a conditional, we must first evaluate *its* guard; and so on. The E-IFTRUE and E-IFFALSE rules tell us what to do when we reach the end of this process and find ourselves with a conditional whose guard is already fully evaluated. In a sense, E-IFTRUE and E-IFFALSE do the real work of evaluation, while E-IF helps determine where the work is to be done. The different character of the rules is sometimes emphasized by referring to E-IFTRUE and E-IFFALSE as *computation rules* and E-IF as a *congruence rule*.

To be a bit more precise about these intuitions, we can define the evaluation relation formally as follows.

3.5.1 DEFINITION: An *instance* of an inference rule is obtained by consistently replacing each metavariable by the same term in the rule's conclusion and all its premises (if any). □

For example,

$$\texttt{if true then true else (if false then false else false)} \longrightarrow \texttt{true}$$

is an instance of E-IFTRUE, where both occurrences of t_2 have been replaced by `true` and t_3 has been replaced by `if false then false else false`.

3.5.2 DEFINITION: A rule is *satisfied* by a relation if, for each instance of the rule, either the conclusion is in the relation or one of the premises is not. □

3.5.3 DEFINITION: The *one-step evaluation* relation $\longrightarrow$ is the smallest binary relation on terms satisfying the three rules in Figure 3-1. When the pair (t, t') is in the evaluation relation, we say that "the evaluation *statement* (or *judgment*) $t \longrightarrow t'$ is *derivable*." □

The force of the word "smallest" here is that a statement $t \longrightarrow t'$ is derivable iff it is justified by the rules: either it is an instance of one of the axioms E-IFTRUE and E-IFFALSE, or else it is the conclusion of an instance of rule E-IF whose premise is derivable. The derivability of a given statement can be justified by exhibiting a *derivation tree* whose leaves are labeled with instances of E-IFTRUE or E-IFFALSE and whose internal nodes are labeled with instances of E-IF. For example, if we abbreviate

$$\texttt{s} \stackrel{\text{def}}{=} \texttt{if true then false else false}$$
$$\texttt{t} \stackrel{\text{def}}{=} \texttt{if s then true else true}$$
$$\texttt{u} \stackrel{\text{def}}{=} \texttt{if false then true else true}$$

to avoid running off the edge of the page, then the derivability of the statement

$$\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}$$

is witnessed by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\,\text{E-IFTRUE}}{\texttt{t} \longrightarrow \texttt{u}}\,\text{E-IF}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\,\text{E-IF}$$

Calling this structure a tree may seem a bit strange, since it doesn't contain any branches. Indeed, the derivation trees witnessing evaluation statements

will always have this slender form: since no evaluation rule has more than one premise, there is no way to construct a branching derivation tree. The terminology will make more sense when we consider derivations for other inductively defined relations, such as typing, where some of the rules do have multiple premises.

The fact that an evaluation statement $t \longrightarrow t'$ is derivable iff there is a derivation tree with $t \longrightarrow t'$ as the label at its root is often useful when reasoning about properties of the evaluation relation. In particular, it leads directly to a proof technique called *induction on derivations*. The proof of the following theorem illustrates this technique.

3.5.4 THEOREM [DETERMINACY OF ONE-STEP EVALUATION]: If $t \longrightarrow t'$ and $t \longrightarrow t''$, then $t' = t''$. □

Proof: By induction on a derivation of $t \longrightarrow t'$. At each step of the induction, we assume the desired result for all smaller derivations, and proceed by a case analysis of the evaluation rule used at the root of the derivation. (Notice that the induction here is not on the length of an evaluation sequence: we are looking just at a single step of evaluation. We could just as well say that we are performing induction on the structure of t, since the structure of an "evaluation derivation" directly follows the structure of the term being reduced. Alternatively, we could just as well perform the induction on the derivation of $t \longrightarrow t''$ instead.)

If the last rule used in the derivation of $t \longrightarrow t'$ is E-IFTRUE, then we know that t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, where $t_1 = \texttt{true}$. But now it is obvious that the last rule in the derivation of $t \longrightarrow t''$ cannot be E-IFFALSE, since we cannot have both $t_1 = \texttt{true}$ and $t_1 = \texttt{false}$. Moreover, the last rule in the second derivation cannot be E-IF either, since the premise of this rule demands that $t_1 \longrightarrow t_1'$ for some t_1', but we have already observed that $\texttt{true}$ does not evaluate to anything. So the last rule in the second derivation can only be E-IFTRUE, and it immediately follows that $t' = t''$.

Similarly, if the last rule used in the derivation of $t \longrightarrow t'$ is E-IFFALSE, then the last rule in the derivation of $t \longrightarrow t''$ must be the same and the result is immediate.

Finally, if the last rule used in the derivation of $t \longrightarrow t'$ is E-IF, then the form of this rule tells us that t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, where $t_1 \longrightarrow t_1'$ for some t_1'. By the same reasoning as above, the last rule in the derivation of $t \longrightarrow t''$ can only be E-IF, which tells us that t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ (which we already know) and that $t_1 \longrightarrow t_1''$ for some t_1''. But now the induction hypothesis applies (since the derivations of $t_1 \longrightarrow t_1'$ and $t_1 \longrightarrow t_1''$ are subderivations of the original derivations of $t \longrightarrow t'$ and

$\mathtt{t} \longrightarrow \mathtt{t}''$), yielding $\mathtt{t}_1' = \mathtt{t}_1''$. This tells us that $\mathtt{t}' = \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 = \mathtt{if}\ \mathtt{t}_1''\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 = \mathtt{t}''$, as required. □

3.5.5 EXERCISE [⋆]: Spell out the induction principle used in the preceding proof, in the style of Theorem 3.3.4. □

Our one-step evaluation relation shows how an abstract machine moves from one state to the next while evaluating a given term. But as programmers we are just as interested in the final results of evaluation—i.e., in states from which the machine *cannot* take a step.

3.5.6 DEFINITION: A term $\mathtt{t}$ is in *normal form* if no evaluation rule applies to it—i.e., if there is no $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$. (We sometimes say "$\mathtt{t}$ is a normal form" as shorthand for "$\mathtt{t}$ is a term in normal form.") □

We have already observed that `true` and `false` are normal forms in the present system (since all the evaluation rules have left-hand sides whose outermost constructor is an `if`, there is obviously no way to instantiate any of the rules so that its left-hand side becomes `true` or `false`). We can rephrase this observation in more general terms as a fact about values:

3.5.7 THEOREM: Every value is in normal form. □

When we enrich the system with arithmetic expressions (and, in later chapters, other constructs), we will always arrange that Theorem 3.5.7 remains valid: being in normal form is part of what it *is* to be a value (i.e., a fully evaluated result), and any language definition in which this is not the case is simply broken.

In the present system, the converse of Theorem 3.5.7 is also true: every normal form is a value. This will not be the case in general; in fact, normal forms that are not values play a critical role in our analysis of *run-time errors,* as we shall see when we get to arithmetic expressions later in this section.

3.5.8 THEOREM: If $\mathtt{t}$ is in normal form, then $\mathtt{t}$ is a value. □

Proof: Suppose that $\mathtt{t}$ is not a value. It is easy to show, by structural induction on $\mathtt{t}$, that it is not a normal form.

Since $\mathtt{t}$ is not a value, it must have the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ for some $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$. Consider the possible forms of $\mathtt{t}_1$.

If $\mathtt{t}_1 = \mathtt{true}$, then clearly $\mathtt{t}$ is not a normal form, since it matches the left-hand side of E-IFTRUE. Similarly if $\mathtt{t}_1 = \mathtt{false}$.

If $\mathtt{t}_1$ is neither `true` nor `false`, then it is not a value. The induction hypothesis then applies, telling us that $\mathtt{t}_1$ is not a normal form—that is, that there is some $\mathtt{t}_1'$ such that $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$. But this means we can use E-IF to derive $\mathtt{t} \longrightarrow \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$, so $\mathtt{t}$ is not a normal form either. □

It is sometimes convenient to be able to view many steps of evaluation as one big state transition. We do this by defining a multi-step evaluation relation that relates a term to all of the terms that can be derived from it by zero or more single steps of evaluation.

3.5.9 DEFINITION: The *multi-step evaluation* relation $\longrightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that (1) if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $\mathtt{t} \longrightarrow^* \mathtt{t}'$, (2) $\mathtt{t} \longrightarrow^* \mathtt{t}$ for all $\mathtt{t}$, and (3) if $\mathtt{t} \longrightarrow^* \mathtt{t}'$ and $\mathtt{t}' \longrightarrow^* \mathtt{t}''$, then $\mathtt{t} \longrightarrow^* \mathtt{t}''$. □

3.5.10 EXERCISE [⋆]: Rephrase Definition 3.5.9 as a set of inference rules. □

Having an explicit notation for multi-step evaluation makes it easy to state facts like the following:

3.5.11 THEOREM [UNIQUENESS OF NORMAL FORMS]: If $\mathtt{t} \longrightarrow^* \mathtt{u}$ and $\mathtt{t} \longrightarrow^* \mathtt{u}'$, where $\mathtt{u}$ and $\mathtt{u}'$ are both normal forms, then $\mathtt{u} = \mathtt{u}'$. □

Proof: Corollary of the determinacy of single-step evaluation (3.5.4). □

The last property of evaluation that we consider before turning our attention to arithmetic expressions is the fact that *every* term can be evaluated to a value. Clearly, this is another property that need not hold in richer languages with features like recursive function definitions. Even in situations where it does hold, its proof is generally much more subtle than the one we are about to see. In Chapter 12 we will return to this point, showing how a type system can be used as the backbone of a termination proof for certain languages.

Most termination proofs in computer science have the same basic form:[5] First, we choose some well-founded set S and give a function f mapping "machine states" (here, terms) into S. Next, we show that, whenever a machine state $\mathtt{t}$ can take a step to another state $\mathtt{t}'$, we have $f(\mathtt{t}') < f(\mathtt{t})$. We now observe that an infinite sequence of evaluation steps beginning from $\mathtt{t}$ can be mapped, via f, into an infinite decreasing chain of elements of S. Since S is well founded, there can be no such infinite decreasing chain, and hence no infinite evaluation sequence. The function f is often called a *termination measure* for the evaluation relation.

3.5.12 THEOREM [TERMINATION OF EVALUATION]: For every term $\mathtt{t}$ there is some normal form $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow^* \mathtt{t}'$. □

Proof: Just observe that each evaluation step reduces the size of the term and that size is a termination measure because the usual order on the natural numbers is well founded. □

5. In Chapter 12 we will see a termination proof with a somewhat more complex structure.

3.5.13 EXERCISE [RECOMMENDED, ⋆⋆]:

1. Suppose we add a new rule

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-FUNNY1)}$$

to the ones in Figure 3-1. Which of the above theorems (3.5.4, 3.5.7, 3.5.8, 3.5.11, and 3.5.12) remain valid?

2. Suppose instead that we add this rule:

$$\frac{t_2 \longrightarrow t_2'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1 \texttt{ then } t_2' \texttt{ else } t_3} \qquad \text{(E-FUNNY2)}$$

Now which of the above theorems remain valid? Do any of the proofs need to change? □

Our next job is to extend the definition of evaluation to arithmetic expressions. Figure 3-2 summarizes the new parts of the definition. (The notation in the upper-right corner of 3-2 reminds us to regard this figure as an extension of 3-1, not a free-standing language in its own right.)

Again, the definition of terms is just a repetition of the syntax we saw in §3.1. The definition of values is a little more interesting, since it requires introducing a new syntactic category of *numeric values*. The intuition is that the final result of evaluating an arithmetic expression can be a number, where a number is either `0` or the successor of a number (but not the successor of an arbitrary value: we will want to say that `succ(true)` is an error, not a value).

The evaluation rules in the right-hand column of Figure 3-2 follow the same pattern as we saw in Figure 3-1. There are four computation rules (E-PREDZERO, E-PREDSUCC, E-ISZEROZERO, and E-ISZEROSUCC) showing how the operators `pred` and `iszero` behave when applied to numbers, and three congruence rules (E-SUCC, E-PRED, and E-ISZERO) that direct evaluation into the "first" subterm of a compound term.

Strictly speaking, we should now repeat Definition 3.5.3 ("the one-step evaluation relation on arithmetic expressions is the smallest relation satisfying all instances of the rules in Figures 3-1 *and* 3-2…"). To avoid wasting space on this kind of boilerplate, it is common practice to take the inference rules as constituting the definition of the relation all by themselves, leaving "the smallest relation containing all instances…" as understood.

The syntactic category of numeric values (`nv`) plays an important role in these rules. In E-PREDSUCC, for example, the fact that the left-hand side is `pred (succ` $\texttt{nv}_1$`)` (rather than `pred (succ` t_1`)`, for example) means that this rule cannot be used to evaluate `pred (succ (pred 0))` to `pred 0`, since this

$\mathbb{B}$ $\mathbb{N}$ *(untyped)* *Extends* **B** *(3-1)*

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\texttt{pred}\ 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

$$\texttt{iszero}\ 0 \longrightarrow \texttt{true} \quad \text{(E-IszeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'} \quad \text{(E-IsZero)}$$

Figure 3-2: Arithmetic expressions (NB)

would require instantiating the metavariable nv_1 with `pred 0`, which is not a numeric value. Instead, the unique next step in the evaluation of the term `pred (succ (pred 0))` has the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred}\ 0 \longrightarrow 0}\ \text{E-PredZero}}{\texttt{succ}\ (\texttt{pred}\ 0) \longrightarrow \texttt{succ}\ 0}\ \text{E-Succ}}{\texttt{pred}\ (\texttt{succ}\ (\texttt{pred}\ 0)) \longrightarrow \texttt{pred}\ (\texttt{succ}\ 0)}\ \text{E-Pred}$$

3.5.14 Exercise [⋆⋆]: Show that Theorem 3.5.4 is also valid for the evaluation relation on arithmetic expressions: if $t \longrightarrow t'$ and $t \longrightarrow t''$, then $t' = t''$. □

Formalizing the operational semantics of a language forces us to specify the behavior of *all* terms, including, in the case at hand, terms like `pred 0` and `succ false`. Under the rules in Figure 3-2, the predecessor of `0` is defined to be `0`. The successor of `false`, on the other hand, is not defined to evaluate to anything (i.e., it is a normal form). We call such terms *stuck*.

3.5.15 Definition: A closed term is *stuck* if it is in normal form but not a value. □

"Stuckness" gives us a simple notion of *run-time error* for our simple machine. Intuitively, it characterizes the situations where the operational semantics does not know what to do because the program has reached a "meaningless state." In a more concrete implementation of the language, these states might correspond to machine failures of various kinds: segmentation faults, execution of illegal instructions, etc. Here, we collapse all these kinds of bad behavior into the single concept of "stuck state."

3.5.16 EXERCISE [RECOMMENDED, ⋆⋆⋆]: A different way of formalizing meaningless states of the abstract machine is to introduce a new term called `wrong` and augment the operational semantics with rules that explicitly generate `wrong` in all the situations where the present semantics gets stuck. To do this in detail, we introduce two new syntactic categories

`badnat` ::=	*non-numeric normal forms:*
`wrong`	*run-time error*
`true`	*constant true*
`false`	*constant false*
`badbool` ::=	*non-boolean normal forms:*
`wrong`	*run-time error*
`nv`	*numeric value*

and we augment the evaluation relation with the following rules:

$$\texttt{if badbool then } t_1 \texttt{ else } t_2 \longrightarrow \texttt{wrong} \qquad \text{(E-IF-WRONG)}$$

$$\texttt{succ badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-SUCC-WRONG)}$$

$$\texttt{pred badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-PRED-WRONG)}$$

$$\texttt{iszero badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-ISZERO-WRONG)}$$

Show that these two treatments of run-time errors agree by (1) finding a precise way of stating the intuition that "the two treatments agree," and (2) proving it. As is often the case when proving things about programming languages, the tricky part here is formulating a precise statement to be proved—the proof itself should be straightforward. □

3.5.17 EXERCISE [RECOMMENDED, ⋆⋆⋆]: Two styles of operational semantics are in common use. The one used in this book is called the *small-step* style, because the definition of the evaluation relation shows how individual steps of computation are used to rewrite a term, bit by bit, until it eventually becomes a value. On top of this, we define a multi-step evaluation relation that allows us to talk about terms evaluating (in many steps) to values. An alternative style,

called *big-step* semantics (or sometimes *natural semantics*), directly formulates the notion of "this term evaluates to that final value," written $t \Downarrow v$. The big-step evaluation rules for our language of boolean and arithmetic expressions look like this:

$$v \Downarrow v \qquad \text{(B-Value)}$$

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_2} \qquad \text{(B-IfTrue)}$$

$$\frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_3} \qquad \text{(B-IfFalse)}$$

$$\frac{t_1 \Downarrow nv_1}{\texttt{succ } t_1 \Downarrow \texttt{succ } nv_1} \qquad \text{(B-Succ)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{pred } t_1 \Downarrow 0} \qquad \text{(B-PredZero)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{pred } t_1 \Downarrow nv_1} \qquad \text{(B-PredSucc)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{iszero } t_1 \Downarrow \texttt{true}} \qquad \text{(B-IszeroZero)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{iszero } t_1 \Downarrow \texttt{false}} \qquad \text{(B-IszeroSucc)}$$

Show that the small-step and big-step semantics for this language coincide, i.e. $t \longrightarrow^* v$ iff $t \Downarrow v$. □

3.5.18 Exercise [⋆⋆ ↛]: Suppose we want to change the evaluation strategy of our language so that the `then` and `else` branches of an `if` expression are evaluated (in that order) *before* the guard is evaluated. Show how the evaluation rules need to change to achieve this effect. □

3.6 Notes

The ideas of abstract and concrete syntax, parsing, etc., are explained in dozens of textbooks on compilers. Inductive definitions, systems of inference rules, and proofs by induction are covered in more detail by Winskel (1993) and Hennessy (1990).

The style of operational semantics that we are using here goes back to a technical report by Plotkin (1981). The big-step style (Exercise 3.5.17) was developed by Kahn (1987). See Astesiano (1991) and Hennessy (1990) for more detailed developments.

Structural induction was introduced to computer science by Burstall (1969).

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.
A: The definitions are almost always wrong. —*Anonymous*