Principles and Practice in Programming Languages: A Project-Based Course

Bor-Yuh Evan Chang

UNIVERSITY OF COLORADO BOULDER
Email address: evan.chang@colorado.edu

Draft as of September 5, 2018.

Disclaimer: This manuscript is a draft of a set of course notes for the Principles of Programming Languages at the University of Colorado Boulder. There may be typos, bugs, or inconsistencies that have yet to be resolved.

Contents

CHAPTER 1

Introduction and Preliminaries

1.1. Getting Your Money's Worth

This course is about principles, concepts, and ideas that underly programming languages. But what does this statement mean?

As a student of computer science, it is completely reasonable to think and ask, "Why bother? I'm proficient and like programming in Ruby. Isn't that enough? Isn't language choice just a matter of taste? If not, should I be using another language?"

Certainly, there are social factors and an aspect of personal preference that affect the programming languages that we use. But there is also a body of principles and mathematical theories that allow us to discuss and think about languages in a rigorous manner. We study these underpinnings because a language affects the way one approaches problems working in that language and affects the way one implements that language. At the end of this course, we hope that you will have grown in the following ways.

You will be able to learn new languages quickly and select a suitable one for your task. This goal is very much a practical one. Languages that are "popular" vary quickly. The TIOBE Programming Community Index[1] surveys the popularity of programming languages over time. While it is just one indicator, the take home message seems to be that a large number of languages are active at any one time, and the level of activity of any language varies widely over time. The "hot" languages now or the languages that you study now will almost certainly not be the ones you need later in your career.

There is a lingo for describing programming languages. The introduction to any programming language is likely to include a statement that aims to succinctly captures various design choices.

> **Python:** "Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes."[2]

[1] `http://www.tiobe.com/content/paperinfo/tpci/`, January 2012.

[2] `http://docs.python.org/faq/general#what-is-python`, January 2012.

OCaml: OCaml offers "a power type system, equipped with parametric polymorphism and type inference [...], user-definable algebraic data types and pattern matching [...], automatic memory management [...], separate compilation of stand-alone applications [...], a sophisticated module system [...], an expressive object-oriented layer [...], and efficient native code compilers."[3]

Haskell: Haskell is "a polymorphically statically typed, lazy, purely functional language, quite different from most other programming languages."[4]

Scala: "Scala is a blend of object-oriented and functional programming concepts in a statically typed language" [3].

At this point, it is understandable if the above statements seem as if they are written a foreign language.

You will gain new ways of viewing computation and approaching algorithmic problems. There are fundamental *models of computation* or *programming paradigms* that persist (e.g., imperative programming and functional programming). Most general-purpose languages mix paradigms but generally have a bias. These biases can shape the way you approach problems.

For natural languages, linguistic relativity, the hypothesis that the language one speaks influences the way one perceives the world, is both tantalizing and controversial. Many have espoused this notion to programming languages by analogy. Setting aside the controversy and assuming at least a kernel of truth, practicing and working with different programming models may expose ideas in new contexts. For example, MapReduce is the programming model created by Google for data processing on large clusters inspired by the functional programming paradigm [1].

This course is not a survey of programming languages present and past. We may make references to programming languages as examples of particular design decisions, but the goal is not to "learn" lots of languages. The analogy to natural languages is perhaps apt. It does not particularly help one understand the structure of natural languages by learning to say "hello" in as many as possible.

You will gain new ways of viewing programs. The meaning of program is given by how it executes, but a program is also artifact in itself that has properties. What a program does or how a program executes is perhaps the primary way one views programs—a program computes

[3] `http://caml.inria.fr/about/`, January 2012.

[4] `http://www.haskell.org/haskellwiki/Introduction/`, January 2012.

something. At the same time, a program can be transformed into a different one that "behaves the same." How do we characterize "behaves the same"? This question is one that can be discussed using programming language theory.

It is also a question of practical importance for language implementation. A compiler translates a program that a human developer writes into one a computational machine can execute. The compiler must abide by the contract that it outputs a program for the machine that "behaves the same" as the program written by the developer.

You will gain insight into avoiding mistakes for when you design languages. When (not if!) you design and implement a language, you will avoid the mistakes of the past. You may not design a general-purpose programming language, but you may have a need to create a "little" configuration, mark-up, or layout language. "Little" languages are often created without much regard to good design because they are "little," but they can quickly become not so "little."

Avoiding bad language design is tricky. Experts make mistakes, and mistakes can have long-lasting effects. Turing award winner Sir C.A.R. Hoare has called his invention of the null reference a "billion dollar mistake" [2].

1.1.1. How? We will construct language interpreters to get experience with the "guts" of programming language design and implementation. The semester project will be to build and understand an interpreter for JavaScript (or rather, variants of it)–our example *source language*. The source language is sometimes called the *object language*. Along the way, we will consider the design decisions made and think about alternatives, and we will study the programming language theory that enable us to reason carefully about them. Our approach will be gradual in that we will initially consider a small subset of our source language and then slowly grow the aspects of the language that we consider.

Our *implementation language* of study will be Scala[5]. The implementation language is sometimes called the *meta language*. Scala is a modern, general-purpose programming language that includes many advanced ideas from programming language research. In particular, we are interested in it because it is especially well suited for building language tools. As quoted above, Scala "blends" concepts from object-oriented and functional programming [3] and in many ways tries to support each in its "native environment." Scala has also found a myriad of other applications, including being a hot language for web services right now. It is compatible with Java and runs on the Java Virtual Machine (JVM) and

[5] `http://www.scala-lang.org/`, August 2012.

has been applied in industrial practice by such companies as LinkedIn and Twitter.[6]

1.2. Is a Program Executed or Evaluated?

Broadly speaking, the "schism" between *imperative* programming and *functional* programming comes down to the basic notion of what defines a computation step. In the *imperative* computational model, we focus on *executing statements* for its *effects* on a *memory*. An imperative program consists of a sequence of statements (or sometimes called *commands* or *instructions*) that is largely viewed as fixed and separate from the memory (or sometimes called the *store*) that it is modifying. Assembly languages and C are often held as examples of imperative programming. In the *functional* computational model, we focus on *evaluating expressions*, that is, rewriting expressions until we obtain a *value*. A program and the computation "state" is an expression (also sometimes called a *term*). Expression rewriting is actually not so unfamiliar. Primary school arithmetic is expression evaluation:

$$(1+1)+(1+1) \quad \longrightarrow \quad 2+(1+1) \quad \longrightarrow \quad 2+2 \quad \longrightarrow \quad 4$$

where the $\longrightarrow$ arrow signifies an evaluation step.

In actuality, the "schism" is false. Few languages are exclusively imperative or exclusively functional in the sense defined above. "Imperative programming languages" have effect-free expression subsets (e.g., for arithmetic), while "functional programming languages" have effectful expressions (e.g., for printing to the screen). Being effect-free or *pure* has certain advantages by being independent of how a machine evaluates expressions. For example, the final result does not depend on the *order of evaluation* (e.g., whether the left $(1+1)$ or the right $(1+1)$ is evaluated first), which makes it easier to reason about programs in isolation (e.g., the meaning of $(1+1)+(1+1)$). At the same time, interacting with the underlying execution engine can be powerful, and thus we will at times want effects in well-controlled ways. The potentially surprising idea at this point and in this course is how much we can program effectively without effects.

We will consider and want to support both effect-free and effectful computation. The take-home message here is it is too simplistic to say a programming language is imperative or functional. Rather, we see that it is a bias in perspective in how we see computation and programs. For imperative languages, programs, and constructs, we speak of *statement execution* that modifies a *memory* or data store. For functional languages,

[6] `http://www.scala-lang.org/node/1658/`, January 2012.

programs, and constructs, we think of *expression evaluation* that reduces to a *value* or terminal result. We will see how this bias affects, for example, how we program repetition (i.e., looping versus recursion or comprehensions).

Note that the term "functional programming language" is quite overloaded in practice. For example, it may refer to the language having the expression rewriting bias described above, being pure and free of effectful expressions, or having *higher-order functions* (discussed in **??**).

Both JavaScript and Scala have aspects of both, including the features that are often considered the most characteristic: *mutation* and *higher-order functions*.

1.2.1. Basic Values, Types, and Expressions. We begin our language study by focusing on a small subset of Scala.

Basic expressions, values, and types are seemingly boring, but they also form the basis of any programming language. A *value* has a *type*, which defines the operations that can be applied to it. Scala has all the familiar basic types, such as `Int`, `Float`, `Double`, `Boolean`, `Char`, and `String`. We can directly write down values of these types using *literals*:

```
42: Int
1.618f: Float
1.618: Double
true: Boolean
'a': Char
"Hello!": String
```

An *expression* can be a literal or consist operations that await to be evaluated. For example, here are some expected expressions:

```
40 + 2: Int
1 < 2: Boolean
if (1 < 2) 3 else 4: Int
"Hello" + "!": String
```

Often, we want to refer to arbitrary values, types, or expressions in a programming language. To do so, we use *meta-variables* that stand for entities in our language of interest, such as v for a value, τ for a type, and e for an expression.

We have annotated types on all of the expressions above, that is, we assert that the value that results from evaluating that expression (if one results) should have that type. In this case, all of these examples are *well-typed* expressions, that is, the typing assertion holds for them. Scala is *statically typed*, which means that the Scala compiler will perform a

validation of expressions at *compile-time* called *type checking* and only translates well-typed expressions. We discuss type checking further in section 4.1; for now, it suffices to view type checking as making sure all operations in subexpressions have the "expected type." We state that an expression e is well-typed with type τ using essentially the same notation as Scala, that is, we write

$$e : \tau \qquad \text{for expression } e \text{ has type } \tau.$$

An expression may not always yield a value. For example, a divide-by-zero expression

```
42 / 0: Int
```

generates a *run-time error*, that is, an error that is raised during evaluation. Some languages are described as being *dynamically typed*, which means no type checking is performed before evaluation. Rather, a run-time type error is raised when evaluation encounters an operations that cannot be applied to the argument values. In general, the term *static* means before evaluating the program, while the term *dynamic* means during the evaluation of the program.

1.2.2. Evaluation. We need a way to write down evaluation to describe how values are computed. Recall that in our setting, the computation state is an expression, so we write

$$e \longrightarrow e' \qquad \text{for expression } e \textit{ steps to} \text{ expression } e' \text{ in one step.}$$

What exactly is "one step" is a matter of definition, which we do not worry about much at this point. Rather, we may write

$$e \longrightarrow^* e' \qquad \text{for expression } e \textit{ steps to } e' \text{ in 0 or more steps,}$$

that is, in some number of steps. For any expression, the possible next steps dictate how evaluation proceeds and is related to concepts like *evaluation order* and *eager versus lazy evaluation*, which we revisit later in Sections 3.1 and **??**. Eager evaluation means that subexpressions are evaluated to values before applying the operation. At this point, it may be hard to imagine anything but eager evaluation. In our current subset of Scala, eager evaluation applies (though Scala supports both).

Sometimes, we only care about the final value of an expression (i.e., its value), so we write

$$e \Downarrow v \qquad \text{for expression } e \textit{ evaluates to} \text{ value } v.$$

1.2.3. Binding Names. Thus far, our expressions consist only of operations on literals, which is certainly restricting! Like other languages, we would like to introduce *names* that are *bound* to other items, such as values.

To introduce a *value binding* in Scala, we use a **val** declaration, such as the following:

```
val two = 2
val four = two + two
```

The first declaration binds the name `two` to the value 2, and the second declaration binds the name `four` to the value of `two + two` (i.e., 4). The syntax of value bindings is as follows:

val x`:` τ `=` e

for a variable x, type τ, and expression e. For the value binding to be well typed, expression e must be of type τ. The type annotation `:` τ is optional, which if elided is inferred from typing expression e. At run-time, the name x is bound to value of expression e (i.e., the value obtained by evaluating e to a value). If e does not evaluate to a value, then no binding occurs.

A binding makes a new name available to an expression in its *scope*. For example, the name `two` must be in scope for the expression

```
two + two
```

to have meaning. Intuitively, to evaluate this expression, we need to know to what value the name `two` is bound. We can view **val** declarations as evaluating to a value *environment*. A value environment is a finite map from names to values, which we write as follows:

$$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$$

For example, the first binding in our example above yields the following environment:

$$[\texttt{two} \mapsto 2]$$

Intuitively, to evaluate the expression `two + two`, we replace or *substitute* the value of the binding for the name `two` and then reduce as before:

$$[\texttt{two} \mapsto 2](\texttt{two + two}) \quad = \quad \texttt{2 + 2} \quad \longrightarrow \quad \texttt{4}.$$

In the above, we are conceptually "applying" the environment as a substitution to the expression `two + two` to get the expression `2 + 2`, which reduces to the value `4`.

For type checking, we need a similar type environment that maps names to types. For example, the type environment

$$[\texttt{two} \mapsto \texttt{Int}]$$

may be used to type check the expression `two + two`.

Declarations may be *sequenced* as seen in the example above where the binding of the name `two` is then used in the binding of the name `four`.

Another kind of binding is for types where we can bind one type name to another creating a *type alias*, such as

```
type Str = String
```

Type binding is not so useful in our current Scala subset, but such bindings become particularly relevant later on in **??**.

1.2.4. Scoping. At this point, all our bindings are placed into the *global scope*. A *scope* is simply a window of the program where a name applies. We can limit the scope of a name by using blocks:

```
{
  val two = 2
}
two // error: name two is out of scope
```

A block introduces a new scope where the name in an inner scope may *shadow* one in an outer scope:

LISTING 1.1. Nested Scopes and Shadowing

```
1  val a = 1
2  val b = 2
3  val c = {
4    val a = 3
5    a + b
6  } + a
```

Here, the use of `a` on line 5 refers to the inner binding on line 4, while the use of `a` on line 6 refers to the outer binding on line 1. Also note that the use of `b` on line 5 refers to the binding of `b` in the outer scope, as `b` is not bound in the inner one. The name `c` ends up being bound to the value 6. In particular, after applying the environment, we end up evaluating the expression `3 + 2 + 1`. From this example, be sure to take note that value binding is *not* imperative assignment: after the inner binding of name `a` on line 4, the outer binding of name `a` still exists but is simply hidden within that scope.

Scala uses *static scoping* (or also called *lexical scoping*), which means that the binding that applies to the use of any name can be determined by examining the program text. Specifically, the binding that applies is not dependent on evaluation order. The scoping rule for Scala is that for any use of a name x, the innermost scope that (a) contains the use of x and (b) has a binding for x is the one that applies. There are only two scopes in the above example (e.g., not one for each **val** declaration). Thus as technicality, the following modification of the example has a compile-time error:

```
1   val a = 1
2   val b = {
3     val c = a // error: use of name a before its binding
4     val a = 2
5     c
6   }
```

The use of name `a` at line 3 refers to the binding at line 4, and the use comes before the binding.

Consider again the nested scopes and shadowing example (Listing 1.1), which is repeated below:

```
1   val a = 1
2   val b = 2
3   val c = {
4     val a = 3
5     a + b
6   } + a
```

How do we describe the evaluation of this expression? The substitution-based evaluation rule for names described previously in subsection 1.2.3 needs to be more nuanced. In particular, eliminating the binding of the name `a` in the outer scope should replace the use of name `a` on line 6 but not the use of name `a` on line 5. In particular, applying the environment $[\texttt{a} \mapsto 1, \texttt{b} \mapsto 2]$ to lines 3 to 6 yields the following:

```
3   val c = {
4     val a = 3
5     a + 2
6   } + 1
```

This notion of substitution is directly linked to terms free and bound variables. In any given expression e, a *bound variable* is one whose binding location is in e, while a *free variable* is one whose binding location is not in e. For example, in the expression

```
4       val a = 3
5       a + b,
```

variable `a` is bound, while variable `b` is free. So to respect shadowing, substitution should only replace *free* occurrences of variables in the environment. In the example, applying $[\texttt{a} \mapsto 1, \texttt{b} \mapsto 2]$ replaces the use of `b`, which is free, but not the use of `a`, which is bound.

Here, we are using the term *variable* in the same sense as name in the above and from mathematical logic rather than the notion of variable in imperative programming. The notion of variable in imperative programming in contrast corresponds to an updatable memory cell.

1.2.5. Function Definitions and Tuples. The most basic and perhaps most important form of abstraction in programming languages is defining functions. For example, the type of functions taking an `Int` and returning an `Int` is

```
Int => Int,
```

and an example Scala function for squaring an `Int` `x` is as follows:

```
(x: Int) => x * x
```

where `x` is a *formal parameter* of type `Int`. Note that this expression is also a value because there is no next step from this expression. That is, it is an `Int => Int` function literal—just like `42` is an `Int` literal. Now, we might want to call the function, and such an expression does have next steps:

```
((x: Int) => x * x)(3)  ⟶  3 * 3  ⟶  3.
```

Often, we want to bind functions to names so that we can reuse them:

```
val square = (x: Int) => x * x
square(3)
square(4).
```

It is really important to note that `square` is not the function but rather the name bound to the function `(x: Int) => x * x`, even though we will often informally (and somewhat sloppily) say "the `square` function." Function literals are often called *lambdas* for historical reasons from the the λ-calculus (read, lambda-calculus) or also *closures* referencing their implementations in compilers.

As defining functions is so common, there is specific syntax for it:

```
def square(x: Int): Int = x * x
square(3)
square(4)
```

that additionally specifies that the function returns a value of type `Int`. Schematically, a function definition has the following form:

def x`(`x_1`: `τ_1`, ..., `x_n`: `τ_n`): `τ` = `e

where the formal parameter types $\tau_1,\ldots,\tau_n$ are always required and the return type τ is sometimes required. However, we adopt the convention of always giving the return type. This convention is good practice in documenting function interfaces, and it saves us from worrying about when Scala actually requires or does not require it.

Note that braces `{}` are not part of the syntax of a function definition. For example, the following code is valid:

```
def max(x: Int, y: Int): Int =
  if (x > y)
    x
  else
    y
```

As a convention, we will not use braces `{}` unless we need to introduce bindings.

In Scala, when the parameter type of a function can be determined by its context, it can be dropped.

```
val square: Int => Int = x => x * x.
```

Dropping the type of a parameter of a function literal is often appropriate when it is an argument to a higher-order function (cf. **??**). When function literals get long, it is a common style to then wrap them with braces:

```
val square = { (x: Int) =>
  x * x
}.
```

We can easily return multiple values by returning a *tuple*. For example, we can write a function `divRem` that takes two integers `x` and `y` and returns a pair of their quotient and their remainder:

```
def divRem(x: Int, y: Int): (Int, Int) = (x / y, x % y)
```

A tuple is a simple data structure that combines a fixed number of values, and there's nothing special about it being used as a return type of `divRem`. We call also write the following tuples:

```
(2, 1): (Int, Int)
(true, 2, 1): (Boolean, Int, Int)
((4, 3), 2, 1): ((Int, Int), Int, Int)
```

A n-tuple expression annotated with a n-tuple type is written as follows:

$$(e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) .$$

The i^{th} component of a tuple e can be obtained using the expression $e._i$ following the example below:

```
val divRemSevenThree: (Int, Int) = divRem(7, 3)
val divSevenThree: Int = divRemSevenThree._1
val remSevenThree: Int = divRemSevenThree._2
```

Typically, a better way to get the components of a tuple is using *pattern matching*:

```
val divRemSevenThree: (Int, Int) = divRem(7, 3)
val (divSevenThree, remSevenThree) = divRemSevenThree
```

Note that the bottom line is a binding of two names `divSevenThree` and `remSevenThree`, which are bound to the first and second components of the tuple `divRemSevenThree`, respectively. The parentheses `()` are necessary in the code above. We will revisit pattern matching in detail in **??**.

There is no 1-tuple type, but there is a 0-tuple type that is specially called `Unit`. There is only one value of type `Unit` (also typically called the unit value). We write down the unit value using the expression `()` (i.e., open-close parentheses). Conceptually, the unit value represents "nothing interesting returned." When we introduce side-effects, a function with return type `Unit` is a good indication that its only purpose is to be executed for side effects because "nothing interesting" is returned. A block that does not have a final expression (e.g., only has declarations) implicitly returns the unit value:

```
val u: Unit = { }
```

Scala has an alternative syntax for functions that have a `Unit` return type:

```
def doNothing() { }
```

Specifically, the = is dropped and no type annotation is needed for the return type since it is fixed to be `Unit`. This syntax makes imperative Scala code look a bit more like C or Java code.

1.3. Recursion, Induction, and Iteration

In our current subset of Scala, we have no way to repeat. A natural way to repeat is using recursive functions. Let us consider defining a Scala function that computes factorial. Recall from discrete mathematics that factorial, written $n!$, corresponds to the number of permutations of n elements and is defined as follows:

$$\begin{array}{rcl} n! & \stackrel{\text{def}}{=} & n \cdot (n-1) \cdot \cdots \cdot 1 \\ 0! & \stackrel{\text{def}}{=} & 1\,. \end{array}$$

From the definition above, we see that factorial satisfies the following equation for $n \geq 0$:

$$(n+1)! = (n+1) \cdot n!\,.$$

In other words, the factorial function can be defined by induction as follows for $n \geq 0$:

$$\begin{array}{rcl} (n+1)! & \stackrel{\text{def}}{=} & (n+1) \cdot n! \\ 0! & \stackrel{\text{def}}{=} & 1\,. \end{array}$$

Now, we can translate this definition to a Scala function to compute factorial as follows:

LISTING 1.2. Factorial: A Basic Implementation

```
def factorial(n: Int): Int =
  if (n == 0) 1 else factorial(n - 1) * n
```

Let us write out some steps of evaluating `factorial(3)`:

```
factorial(3)
  ⟶*   if (3 == 0) 1 else factorial(3 - 1) * 3
  ⟶*   factorial(2) * 3
  ⟶*   factorial(1) * 2 * 3
  ⟶*   factorial(0) * 1 * 2 * 3
  ⟶*   1 * 1 * 2 * 3
  ⟶*   6
```

where the sequence above is shorthand for expressing that each successive pair of expressions is related by the evaluation relation written between them. In the above, we were a bit sloppy in identifying the function value bound to `factorial` with the variable name `factorial` in order to be concise.

1.3.1. Induction: Reasoning about Recursive Programs. In this course, the key take-away is to be able to program by *thinking inductively*. Induction is an important proof technique for reasoning about recursively-defined objects that you might recall from a discrete mathematics course.

Here, we begin work on "thinking inductively" by considering basic proofs of properties of recursive Scala functions.

The simplest form of induction is what we call *mathematical induction*, that is, induction over natural numbers. Intuitively, to prove a property P over all natural numbers (i.e., $\forall n \in \mathbb{N}.P(n)$), we consider two cases: (a) we prove the property holds for 0 (i.e., $P(0)$), which is called the base case; and (b) we prove that the property holds for $n+1$ assuming it holds for an $n \geq 0$ (i.e., $\forall n \in \mathbb{N}.(P(n) \implies P(n+1))$), which is called the inductive case.

As an example, let us prove that our Scala function `factorial` computes the mathematical definition of factorial $n!$. To state this property precisely, we need a way to relate mathematical numbers with Scala values. To do so, we use the notation $\llcorner n \lrcorner$ to mean the Scala integer value corresponding to the mathematical number n.

THEOREM 1.1. *For all integers n such that $n \geq 0$,*

$$\texttt{factorial}(\llcorner n \lrcorner) \quad \longrightarrow^* \quad \llcorner n! \lrcorner .$$

PROOF. By mathematical induction on n.

BASE CASE ($n = 0$). Note that $\llcorner 0 \lrcorner = 0$. Taking a few steps of evaluation, we have that

$$\texttt{factorial(0)} \quad \longrightarrow^* \quad \texttt{1} .$$

Then, the Scala value `1` can also be written as $\llcorner 0! \lrcorner$ because mathematically $0! = 1$.

INDUCTIVE CASE ($n = n' + 1$ for some $n' \geq 0$). The induction hypothesis is as follows:

$$\texttt{factorial}(\llcorner n' \lrcorner) \quad \longrightarrow^* \quad \llcorner n'! \lrcorner .$$

Let us evaluate $\texttt{factorial}(\llcorner n \lrcorner)$ a few steps, and we have the following:

$$\texttt{factorial}(\llcorner n \lrcorner) \quad \longrightarrow^* \quad \texttt{factorial}(\llcorner n-1 \lrcorner) * \llcorner n \lrcorner$$

because we know that $n \neq 0$. Applying the induction hypothesis, we have that

$$\llcorner n \lrcorner * \texttt{factorial}(\llcorner n-1 \lrcorner) \quad \longrightarrow^* \quad \llcorner n'! \lrcorner * \llcorner n \lrcorner$$

noting that $n' = n - 1$. By further evaluation, we have that

$$\llcorner n'! \lrcorner * \llcorner n \lrcorner \quad \longrightarrow \quad \llcorner n'! \cdot n \lrcorner .$$

Note that $n'! \cdot n = n \cdot n'! = n \cdot (n-1)! = n!$, which completes this case. □

In the above, we are actually using an abstract notion of evaluation where Scala integer values are unbounded. In implementation, Scala integers are in fact 32-bit signed two's complement integers that we have ignored in our evaluation relation. It is often convenient to use abstract models of evaluation to essentially separate concerns. Here, we use an abstract model of evaluation to ignore overflow.

1.3.2. Pattern Matching. There is another style of writing recursive functions using pattern matching that looks somewhat closer to the structure of an inductive proof. For example, we can write an implementation of factorial equivalent to Listing 1.2 as follows:

LISTING 1.3. Factorial: With Pattern Matching

```
def factorial(n: Int): Int = n match {
  case 0 => 1
  case _ => factorial(n - 1) * n
}
```

The **match** expression has the following form:

e **match** {
  case $pattern_1$ => e_1
  ...
  case $pattern_n$ => e_n
}

and evaluates by comparing the value of expression *e* against the patterns given by the **case**s. Patterns are tried in sequence from $pattern_1$ to $pattern_n$. Evaluation continues with the corresponding expression for the first pattern that matches. Again, we will revisit pattern matching in more detail in subsection 1.3.5. For the moment, simply recognize that patterns in general bind names (like seen previously in subsection 1.2.5). In Listing 1.3, we use the "wildcard" pattern _ to match anything that is non-zero.

1.3.3. Function Preconditions. The definitions of `factorial` given in both Listing 1.2 and Listing 1.3 implicitly assume that they are called with non-negative integer values. Consider evaluating `factorial(-2)`:

```
factorial(-2)
  ⟶*   factorial(-3) * -2
  ⟶*   factorial(-4) * -2 * -3
  ⟶*   factorial(-5) * -2 * -3 * -4
  ⟶*   ....
```

We see that we have non-termination with infinite recursion. In implementation, we recurse until the run-time yields a stack overflow error.

Following principles of good design, we should at least document in a comment the requirement on the input parameter `n` that it should be non-negative.

LISTING 1.4. Factorial: With Documentation

```
/** Factorial.
  *
  * @param n the Int to compute factorial
  * @return the factorial of n (i.e., n!)
  * @note n must be non-negative (i.e., n >= 0)
  */
def factorial(n: Int): Int = n match {
  case 0 => 1
  case _ => factorial(n - 1) * n
}
```

In Scala, we can do something a bit better in that we can specify such *preconditions* in code:

LISTING 1.5. Factorial: With a Specified Precondition

```
def factorial(n: Int): Int = {
  require(n >= 0)
  n match {
    case 0 => 1
    case _ => factorial(n - 1) * n
  }
}
```

If this version of `factorial` is called with a negative integer, it will result in a run-time exception:

$$\texttt{factorial(-2)} \longrightarrow^* \texttt{IllegalArgumentException}$$

The `require` function does nothing if its argument evaluates to **true** and otherwise throws an exception if its argument evaluates to **false**.

For `factorial`, it is clear that the `require` will never fail in any recursive call. We really only need to check the initial `n` from the initiating call to `factorial`. One way we can do this is to use a helper function that actually performs the recursive computation:

LISTING 1.6. Factorial: Separating the Precondition from the Recursion

```
def factorial(n: Int): Int = {
```

```
    require(n >= 0)
    /**
    * @param n the Int to compute factorial
    * @return the factorial of n (i.e., n!)
    * @note n must be non-negative (i.e., n >= 0)
    */
    def rec(n: Int): Int = n match {
      case 0 => 1
      case _ => rec(n - 1) * n
    }
    rec(n)
  }
```

Here, the `rec` function is local to the `factorial` function. The `rec` function does not do any checking of its argument, but the `require` check in `factorial` will ensure that the call to `rec(n)` always terminates.

We have also copied the documentation from Listing 1.4 to the `rec` definition to revisit *thinking inductively* following subsection 1.3.1. It is natural to test your program by writing out some steps of evaluation by "unrolling the recursion" as we have done thus far in this Section, but it will become crucial to move beyond "unrolling" to thinking in terms "assume" and "guarantee" (or "requires" and "ensures"). The `rec` function implementation assumes that its input `n` is non-negative (i.e, $n \geq 0$), and it guarantees that it returns the factorial of `n` (i.e., `n`!). As a client of the `rec` function (i.e., code that calls `rec`), we, in turn, need to guarantee that we call `rec` with a non-negative n, but we get to assume that it indeed returns $n!$. In general, this style of thinking in terms of a functions pre- and post-conditions is called design-by-contract, and thus, the pre- and post-conditions of a function are sometimes called its *contract* or *specification*.

The essence of *thinking inductively* is using both implementation *and* client thinking when defining a recursive function. In particular, when making a recursive call, think of that function as a black-box library implementation that satisfies its contract. As long as we make a recursive call on a substructure (i.e., "something smaller") that satisfies any pre-conditions, we can assume that it yields a return value satifying its post-condition. For example, let's talk through thinking inductively for `rec`. We assume `n` is non-negative, and we need to return `n`!. In the case that `n` is `0`, we return `1` and indeed `0` ! = `1`. Otherwise, we have a black-box library function `rec` for `rec(n - 1)`. Indeed, `n - 1` is non-negative because $n \neq 0$ and $n \geq 0$, so we can assume that `rec(n - 1)` returns `(n - 1)`!. To return `n`!, we just need to multiply `(n - 1)`! by `n` (i.e., `rec(n - 1) * n`).

1.3.4. Iteration: Tail Recursion with an Accumulator. Examining the evaluation of the various versions of `factorial` in this section, we observe that they all behave similarly: (1) the recursion builds up an expression consisting of a sequence of multiplication $*$ operations, and then (2) the multiplication operations are evaluated to yield the result. In a typical run-time system, step (1) grows the call stack of activation records with recursive calls recording pending evaluation (i.e., the $*$ operation), and each individual $*$ operation in step (2) is executed while unwinding the call stack on return. Our abstract notation for evaluation does not represent a call stack explicitly, but we can see the corresponding behavior in the growing "pending" expression.

Not all recursive functions require a call stack of activation records. In particular, when there's nothing left to do on return, there is no "pending computation" to record. This kind of recursive function is called *tail recursive*. A tail recursive version of the factorial function is given below in Listing 1.7.

LISTING 1.7. Tail-Recursive Factorial: Using an Accumulator

```
def factorial(n: Int): Int = {
  require(n >= 0)
  def loop(acc: Int, n: Int): Int = n match {
    case 0 => acc
    case _ => loop(acc * n, n - 1)
  }
  loop(1, n)
}
```

Let us write out some steps of evaluating `factorial(3)` for this version:

```
factorial(3)
  ⟶*   loop(1, 3)
  ⟶*   loop(1 * 3, 2)
  ⟶    loop(3, 2)
  ⟶*   loop(3 * 2, 1)
  ⟶    loop(6, 1)
  ⟶*   loop(6 * 1, 0)
  ⟶    loop(6, 0)
  ⟶*   6
```

Observe that the `acc` variable serves to *accumulate* the result. When we reach the base case (i.e., `0`), then we simply return the accumulator variable `acc`. Notice that no expression gets built up during the course of the recursion. When the last call to `loop` returns, we have the final result. It is

an important optimization for compilers to recognize tail recursion and avoid building a call stack unnecessarily.

The `loop` helper function is slightly more general than the corresponding `rec` function from Listing 1.6 and thus thinking inductively (or reasoning in terms of specification) becomes even more crucial. Let us consider the more general specification of the `loop` helper function:

LISTING 1.8. Tail-Recursive Factorial: With Specification

```
def factorial(n: Int): Int = {
  require(n >= 0)
  /**
  * @param acc the Int accumulator
  * @param n the Int to compute
  * @return acc times the factorial of n (i.e., acc * n!)
  * @note n must be non-negative (i.e., n >= 0)
  */
  def loop(acc: Int, n: Int): Int = n match {
    case 0 => acc
    case _ => loop(acc * n, n - 1)
  }
  loop(1, n)
}
```

In particular, the specification says that the `loop` function returns $\texttt{acc} \cdot \texttt{n}!$ instead of simply $\texttt{n}!$. The top-level `factorial` function can use this more general helper function to return $\texttt{n}!$ by simply calling it with `acc` bound to `1`. Now to implement `loop`, in the case that `n` is `0`, we need to return $\texttt{acc} \cdot 0!$, which is $\texttt{acc} \cdot 1$ or `acc`. Otherwise, we can assume that $\texttt{loop}(\texttt{acc} * \texttt{n}, \texttt{n} - 1)$ returns $(\texttt{acc} * \texttt{n}) \cdot (\texttt{n} - 1)!$, which is $\texttt{acc} \cdot \texttt{n}!$.

A tail recursive function corresponds closely to a loop (e.g., a **while** loop in a language like Java) but does not require mutation. For example, consider the following version of `factorial` in Java using a **while** loop and variable assignment:

```
int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

Conceptually, each iteration of the **while** loop corresponds to a call of `loop`. The value of `acc` and `n` in each iteration of the **while** loop correspond to the values bound to `acc` and `n` on each recursive call to `loop`. Tail call optimization is essentially about lowering tail recursive functions to loops like this one. We can thus see iteration (with loops) as a lower-level, special case of recursion.

1.3.5. Inductive Data Types and Pattern Matching. If we compare our implementations of `factorial` thus far with the inductive proof in subsection 1.3.1, we find a small dissonance because our implementations use the Scala type `Int` to represent natural numbers $\mathbb{N}$. To help us see the inductive structure of `factorial` more directly, let us define a user-defined data type to represent natural numbers in unary:

LISTING 1.9. User-Defined Unary Representation of Natural Numbers

```
sealed abstract class Nat
case object Z extends Nat
case class S(n: Nat) extends Nat
```

We introduce a new type `Nat` (i.e., an **abstract class**). A value of type `Nat` could be the object `Z`, representing the natural number 0 or a `S`(n) where n is some other value of type `Nat`. The **case class** declaration in Scala defines a constructor `S` that takes as a parameter `n : Nat`, representing the successor (i.e., +1) of a natural number `n`. We can see that the values of type `Nat` are

$$\texttt{Z}, \texttt{S(Z)}, \texttt{S(S(Z))}, \ldots .$$

Now, we can define `factorial` over `Nat` similar to Listing 1.3 over `Int`:

LISTING 1.10. Factorial: With `Nat`

```
def factorial(n: Nat): Nat = n match {
  case Z => one
  case S(nprime) => times(factorial(nprime), n)
}
```

assuming an in-scope binding for `one` and a multiplication function defined over `Nat`:

```
val one: Nat = S(Z)
val times: (Nat, Nat) => Nat .
```

Defining `times` is a good exercise, which will in turn require defining a `plus` function[7]. In defining `times` and `plus`, it will become extremely evident that the the `Nat` type is indeed a really inefficient representation for numbers, but our goal here in this subsection is not a practical implementation but rather to deepen our understanding of the relationship between recursive programs and inductive data types.

So in Listing 1.10, take note of the use of pattern matching to bind `nprime` to the predecessor of `n` and how the code matches up with the proof of Theorem 1.1 in subsection 1.3.1 or the *thinking inductively* discussion in subsection 1.3.3.

[7]As an aside, one can define operators like $*$ and + for user-defined data types in Scala.

1.4. Lab 1

1.4.1. Scala Basics: Binding and Scope. For each the following uses of names, give the line where that name is bound. Briefly explain your reasoning (in no more than 1–2 sentences).

(1) Consider the following Scala code.

```
1  val pi = 3.14
2  def circumference(r: Double): Double = {
3    val pi = 3.14159
4    2.0 * pi * r
5  }
6  def area(r: Double): Double =
7    pi * r * r
```

The use of `pi` at line 4 is bound at which line? The use of `pi` at line 7 is bound at which line?

(2) Consider the following Scala code.

```
1  val x = 3
2  def f(x: Int): Int =
3    x match {
4      case 0 => 0
5      case x => {
6        val y = x + 1
7        ({
8           val x = y + 1
9           y
10         } * f(x - 1))
11     }
12   }
13 val y = x + f(x)
```

The use of x at line 3 is bound at which line? The use of x at line 6 is bound at which line? The use of x at line 10 is bound at which line? The use of x at line 13 is bound at which line?

1.4.2. Scala Basics: Typing. In the following, I have left off the return type of function g. The body of g is well-typed if we can come up with a valid return type. Is the body of g well-typed?

```
1  def g(x: Int) = {
2    val (a, b) = (1, (x, 3))
3    if (x == 0) (b, 1) else (b, a + 2)
```

```
4    }
```

If so, give the return type of g and explain how you determined this type. For this explanation, first, give the types for the names `a` and `b`. Then, explain the body expression using the following format:

$e : \tau$ because
    $e_1 : \tau_1$ because
        ...
    $e_2 : \tau_2$ because
        ...

where e_1 and e_2 are subexpressions of e. Stop when you reach values (or names).

As an example of the suggested format, consider the `plus` function:

```
def plus(x: Int, y: Int) = x + y
```

Yes, the body expression of `plus` is well-typed with type `Int`.

`x + y : Int` because
    `x : Int`
    `y : Int`

1.4.3. Run-Time Library. Most languages come with a standard library with support for things like data structures, mathematical operators, string processing, etc. Standard library functions may be implemented in the object language (perhaps for portability) or the meta language (perhaps for implementation efficiency).

For this question, we will implement some library functions in Scala, our meta language, that we can imagine will be part of the run-time for our object language interpreter. In actuality, the main purpose of this exercise is to warm-up with Scala.

(1) Write a function `abs`

```
def abs(n: Double): Double
```

that returns the absolute value of `n`. This a function that takes a value of type `Double` and returns a value of type `Double`. This function corresponds to the JavaScript library function `Math.abs`.

Instructor Solution: 1 line.

(2) Write a function `xor`

```
def xor(a: Boolean, b: Boolean): Boolean
```

that returns the exclusive-or of `a` and `b`. The exclusive-or returns **true** if and only if exactly one of `a` or `b` is **true**. For practice, do not use the Boolean operators. Instead, only use the **if**-**else** expression and the Boolean literals (i.e., **true** or **false**).

Instructor Solution: 4 lines (including 1 line for a closing brace).

1.4.4. Run-Time Library: Recursion.

(1) Write a recursive function `repeat`

```
def repeat(s: String, n: Int): String
```

where `repeat(s, n)` returns a string with `n` copies of `s` concatenated together. For example, `repeat("a",3)` returns `"aaa"`. This function corresponds to the function `goog.string.repeat` in the Google Closure library.

Instructor Solution: 4 lines (including 1 line for a closing brace).

(2) In this exercise, we will implement the square root function—`Math.sqrt` in the JavaScript standard library. To do so, we will use Newton's method (also known as Newton-Raphson).

Recall from Calculus that a root of a differentiable function can be iteratively approximated by following tangent lines. More precisely, let f be a differentiable function, and let x_0 be an initial guess for a root of f. Then, Newton's method specifies a sequence of approximations $x_0, x_1, \ldots$ with the following recursive equation:[8]

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} .$$

The square root of a real number c for $c > 0$, written $\sqrt{c}$, is a positive x such that $x^2 = c$. Thus, to compute the square root of a number c, we want to find the positive root of the function:

$$f(x) = x^2 - c .$$

Thus, the following recursive equation defines a sequence of approximations for $\sqrt{c}$:

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} .$$

(a) First, implement a function `sqrtStep`

```
def sqrtStep(c: Double, xn: Double): Double
```

that takes one step of approximation in computing $\sqrt{c}$ (i.e., computes x_{n+1} from x_n).

Instructor Solution: 1 line.

(b) Next, implement a function `sqrtN`

[8] The following link is a refresher video on this algorithm: `http://www.youtube.com/watch?v=1uN8cBGVpfs`, January 2012

```
def sqrtN(c: Double, x0: Double, n: Int): Double
```

that computes the nth approximation x_n from an initial guess x_0. You will want to call `sqrtStep` implemented in the previous part.
Challenge yourself to implement this function using recursion and no mutable variables (i.e., **var**s)—you will want to use a recursive helper function. It is also quite informative to compare your recursive solution with one using a **while** loop.
Instructor Solution: 7 lines (including 2 lines for closing braces and 1 line for a `require`).

(c) Now, implement a function `sqrtErr`

```
def sqrtErr(c: Double, x0: Double,
            epsilon: Double): Double
```

that is very similar to `sqrtN` but instead computes approximations x_n until the approximation error is within ε (`epsilon`), that is,

$$|x_n^2 - c| < \varepsilon\,.$$

You can use your absolute value function `abs` implemented in a previous part. A wrapper function `sqrt` is given in the template that simply calls `sqrtErr` with a choice of `x0` and `epsilon`.
Again, challenge yourself to implement this function using recursion and compare your recursive solution to one with a **while** loop.
Instructor Solution: 5 lines (including 1 line for a closing brace and 1 line for a `require`).

1.4.5. Data Structures Review: Binary Search Trees. In this question, we will review implementing operations on binary search trees from Data Structures. Balanced binary search trees are common in standard libraries to implement collections, such as sets or maps. For example, the Google Closure library for JavaScript has `goog.structs.AvlTree`. For simplicity, we will not worry about balancing in this question.

Trees are important structures in developing interpreters, so this question is also critical practice in implementing tree manipulations.

A binary search tree is a binary tree that satisfies an ordering invariant. Let n be any node in a binary search tree whose data value is d, left child is l, and right child is r. The ordering invariant is that all of the data values in the subtree rooted at l must be $< d$, and all of the data values in the subtree rooted at r must be $\geq d$.

We will represent a binary trees containing integer data using the following Scala **case class**es and **case object**s:

```
sealed abstract class SearchTree
case object Empty extends SearchTree
case class Node(l: SearchTree, d: Int, r: SearchTree) extends SearchTree
```

A `SearchTree` is either `Empty` or a `Node` with left child `l`, data value `d`, and right child `r`.

For this question, we will implement the following four functions.

(1) The function `repOk`

```
def repOk(t: SearchTree): Boolean
```

checks that an instance of `SearchTree` is valid binary search tree. In other words, it checks using a traversal of the tree the ordering invariant. This function is useful for testing your implementation. A skeleton of this function has been provided for you in the template.

Instructor Solution: 7 lines (including 2 lines for closing braces).

(2) The function `insert`

```
def insert(t: SearchTree, n: Int): SearchTree
```

inserts an integer into the binary search tree. Observe that the return type of `insert` is a `SearchTree`. This choice suggests a functional style where we construct and return a new output tree that is the input tree `t` with the additional integer `n` as opposed to destructively updating the input tree.

Instructor Solution: 4 lines (including 1 line for a closing brace).

(3) The function `deleteMin`

```
def deleteMin(t: SearchTree): (SearchTree, Int)
```

deletes the smallest data element in the search tree (i.e., the left-most node). It returns both the updated tree and the data value of the deleted node. This function is intended as a helper function for the `delete` function. Most of this function is provided in the template.

Instructor Solution: 9 lines (including 2 lines for closing braces and 1 line for a `require`).

(4) The function `delete`

```
def delete(t: SearchTree, n: Int): SearchTree
```

removes the first node with data value equal to `n`. This function is trickier than `insert` because what should be done depends on whether the node to be deleted has children or not. We advise that you take advantage of pattern matching to organize the cases.

Instructor Solution: 10 lines (including 2 lines for closing braces).

1.4.6. JavaScripty Interpreter: Numbers. JavaScript is a complex language and thus difficult to build an interpreter for it all at once. In this course, we will make some simplifications. We consider subsets of JavaScript and incrementally examine more and more complex subsets during the course of the semester. For clarity, let us call the language that we implement in this course JAVASCRIPTY.

For the moment, let us define JAVASCRIPTY to be a proper subset of JavaScript. That is, we may choose to omit complex behavior in JavaScript, but we want any programs that we admit in JAVASCRIPTY to behave in the same way as in JavaScript.

In actuality, there is not one language called JavaScript but a set of closely related languages that may have slightly different semantics. In deciding how a JAVASCRIPTY program should behave, we will consult a reference implementation that we fix to be Google's V8 JavaScript Engine. We will run V8 via Node.js, and thus, we will often need to write little test JavaScript programs and run it through Node.js to see how the test should behave.

In this lab, we consider an arithmetic sub-language of JavaScript (i.e., an extremely basic calculator). The first thing we have to consider is how to represent a JAVASCRIPTY *program as data* in Scala, that is, we need to be able to represent a program in our object/source language JAVASCRIPTY as data in our meta/implementation language Scala.

To a JAVASCRIPTY programmer, a JAVASCRIPTY program is a text file—a string of characters. Such a representation is quite cumbersome to work with as a language implementer. Instead, language implementations typically work with trees called *abstract syntax trees* (ASTs). What strings are considered JAVASCRIPTY programs is called the *concrete syntax* of JAVASCRIPTY, while the trees (or *terms*) that are JAVASCRIPTY programs is called the *abstract syntax* of JAVASCRIPTY. The process of converting a program in concrete syntax (i.e., as a string) to a program in abstract syntax (i.e., as a tree) is called *parsing*.

For this lab, a parser is provided for you that reads in a JAVASCRIPTY program-as-a-string and converts into an abstract syntax tree. We will represent abstract syntax trees in Scala using **case class**es and **case object**s.

```
sealed abstract class Expr
case class N(n: Double) extends Expr
  N(n)   n
case class Unary(uop: Uop, e1: Expr) extends Expr
  Unary(uop, e1)   uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  Binary(bop, e1, e2)   e1 bop e2

sealed abstract class Uop
case object Neg extends Uop
  Neg   -

sealed abstract class Bop
case object Plus extends Bop
  Plus   +
case object Minus extends Bop
  Minus   -
case object Times extends Bop
  Times   *
case object Div extends Bop
  Div   /
```

FIGURE 1.1. Representing in Scala the abstract syntax of JAVASCRIPTY. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.

The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 1.1.

(1)

INTERPRETER 1.1. Implement the `eval` function

```
def eval(e: Expr): Double
```

that evaluates a JAVASCRIPTY expression `e` to the Scala double-precision floating point number corresponding to the *value* of `e`.

Consider a JAVASCRIPTY program *e*; imagine *e* stands for the concrete syntax or text of the JAVASCRIPTY program. This text is parsed into a JAVASCRIPTY AST `e`, that is, a Scala value of type `Expr`. Then, the result of `eval` is a Scala number of type `Double` and should match the interpretation of *e* as a JavaScript expression. These distinctions can be subtle

but learning to distinguish between them will go a long way in making sense of programming languages.

At this point, you have implemented your first language interpreter!

CHAPTER 2

Approaching a Programming Language

We have studied subsets of Scala up to this point mostly by example. At some point, we may wonder (1) what are all the Scala programs that we can write, and (2) what do they mean? The answer to question (1) is given by a definition of Scala's *syntax*, while the answer to question (2) is given by a definition of Scala's *semantics*.

As a language designer, it is critical to us that we define unambiguously the syntax and semantics so that everyone understands our intent. Language users need to know what they can write and how the programs they write will execute as alluded to in the previous paragraph. Language implementers need to know what are the possible input strings and what they mean in order to produce *semantically-equivalent* output code.

2.1. Syntax: Grammars and Scoping

Stated informally, the syntax of a language is concerned with the form of programs, that is, the strings that we consider programs. The semantics of a language is concerned with the meaning of programs, that is, how programs evaluate. Because there an unbounded number of possible programs in a language, we need tools to speak more abstractly about them. Here, we focus on describing the syntax of programming languages. We will consider defining the semantics of programming languages in section 3.1.

The *concrete syntax* of a programming language is concerned with how to write down expressions, statements, and programs as strings. Concrete syntax is the primary interface between the language user and the language implementation. Thus, the design of concrete syntax focuses on improving readability and perhaps writability for software developers. There are significant sociological considerations, such as appealing to tradition (e.g., using curly braces $\{\ldots\}$ to denote blocks of statements). A large part of concrete syntax design is a human-computer interaction problem, which is outside of what we can consider in this course.

The *abstract syntax* of a programming language is the representation of programs used by language implementations and thus an important mental model for language implementers and language users. We will

draw out precisely the distinction between concrete and abstract syntax in this section.

2.1.1. Context-Free Languages and Context-Free Grammars. A *language* $\mathscr{L}$ is a set of strings composed of characters drawn from some *alphabet* Σ (i.e., $\mathscr{L} \subseteq \Sigma^*$). A string in a language is sometimes called a *sentence*.

The standard way to describe the concrete syntax of a language is using *context-free grammars*. A context-free grammar is a way to describe a class of languages called *context-free languages*. A context-free grammar defines a language inductively and consists of *terminals*, *nonterminals*, and *productions*. Terminals and non-terminals are generically called *symbols*. The terminals of a grammar correspond to the alphabet of the language being defined and are the basic building blocks. Nonterminals are defined via productions and conceptually recognize a sequence of symbols belonging to a sublanguage. A production has the form $N ::= \alpha$ where N is a non-terminal from the set of non-terminals $\mathscr{N}$ and α is a sequence of symbols (i.e., $\alpha \in (\Sigma \cup \mathscr{N})^*$). A set of of productions with the same non-terminal, such as $N ::= \alpha_1, \ldots, N ::= \alpha_n$, is usually written with one instance of the non-terminal and the right-hand sides separated by |, such as $N ::= \alpha_1 \mid \cdots \mid \alpha_n$. Such a set of productions can be read informally as, "N is generated by either α_1, ..., or α_n." For any non-terminal N, we can talk about the language or *syntactic category* defined by that non-terminal.

As an example, let us consider defining a language of integers as follows:

integers	$i ::= -n \mid n$
numbers	$n ::= d \mid d\ n$
digits	$d ::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}$

with the alphabet $\Sigma \stackrel{\text{def}}{=} \{0,1,2,3,4,5,6,7,8,9,-\}$. We identify the overall language by the *start non-terminal* (also called the *start symbol*). By convention, we typically consider the non-terminal listed first as the start non-terminal. Here, we have strings like `1`, `2`, `42`, `100`, and $-$`7` in our language of integers. Note that strings like `012` and $-$`0` are also in this language.

2.1.1.1. *Derivation of a Sentence in a Grammar.* Formally, a string is in the language described by a grammar if and only if we can give a *derivation* for it from the start symbol. We say a sequence of symbols β is derived from another sequence of symbols α, written as

$$\alpha \implies \beta,$$

when β is obtained by replacing a non-terminal N in α with the right-hand side of a production of N. We can give a witness that a string s belongs to a language by showing derivation steps from the start symbol to the string s. For example, we show that `012` is in the language of integers defined above:

$$\begin{array}{rcl} i & \Longrightarrow & n \\ & \Longrightarrow & d\ n \\ & \Longrightarrow & \texttt{0}\ n \\ & \Longrightarrow & \texttt{0}\ d\ n \\ & \Longrightarrow & \texttt{01}\ n \\ & \Longrightarrow & \texttt{01}\ d \\ & \Longrightarrow & \texttt{012}. \end{array}$$

In the above, we have shown a *leftmost derivation*, that is, one where we always choose to expand the leftmost non-terminal. We can similarly define a *rightmost derivation*. Note that there are typically several derivations that witness a string belonging the language described by a grammar.

We can now state precisely the language described by a grammar. Let $\mathscr{L}(G)$ be the language described by grammar G over the alphabet Σ, start symbol S, and derivation relation $\Longrightarrow$. We define the relation $\alpha \Longrightarrow^* \beta$ as holding if and only if β can be derived from α with the one-step derivation relation $\Longrightarrow$ in zero or more steps (i.e., $\Longrightarrow^*$ is the reflexive-transitive closure of $\Longrightarrow$). Then, $\mathscr{L}(G)$ is defined as follows:

$$\mathscr{L}(G) \stackrel{\text{def}}{=} \left\{ s \;\middle|\; s \in \Sigma^* \text{ and } S \Longrightarrow^* s \right\}.$$

2.1.2. Lexical and Syntactic. In language implementations, we often want to separate the simple grouping of characters from the identification of structure. For example, when we read the string `23 + 45`, we would normally see three pieces: the number twenty-three, the plus operator, and the number forty-five, rather than the literal sequence of characters '2', '3', ' ', '+', ' ', '4', and '5'.

Thus, it is common to specify the *lexical* structure of a language separately from the *syntactic* structure. The lexical structure is this simple grouping of characters, which is often specified using regular expressions. A *lexer* transforms a sequence of literal characters into a sequence of *lexemes* classified into *tokens*. For example, a lexer might transform the string `"23+45"` into the following sequence:

$$num(\texttt{"23"}), +, num(\texttt{"45"})$$

consisting of three tokens: a *num* token with lexeme `"23"`, a plus token with lexeme `"+"`, and a *num* token with lexeme `"45"`. Since there is only

one possible lexeme for the plus token, we abuse notation slightly and name the token by the lexeme.

A *parser* then recognizes strings of tokens, typically specified using context-free grammars. For example, we might define a language of expressions with numbers and the plus operator:

$$expr ::= num \mid expr + expr\,.$$

Note that *num* is a terminal in this grammar.

There is an analogy to parsing sentences in natural languages. Grouping letters into words in a sentence corresponds essentially to lexing, while classifying words into grammatical elements (e.g., nouns, verbs, noun phrases, verb phrases) corresponds to parsing.

2.1.3. Ambiguous Grammars. Consider the following arithmetic expression:

$$100/10/5\,.$$

Should it be read as $(100/10)/5$ or $100/(10/5)$? The former equals 2, while the latter equals 50. In mathematics, we adopt conventions that, for example, choosing the former over the latter.

Now consider a language implementation that is given the following input:

```
100/10/5.
```

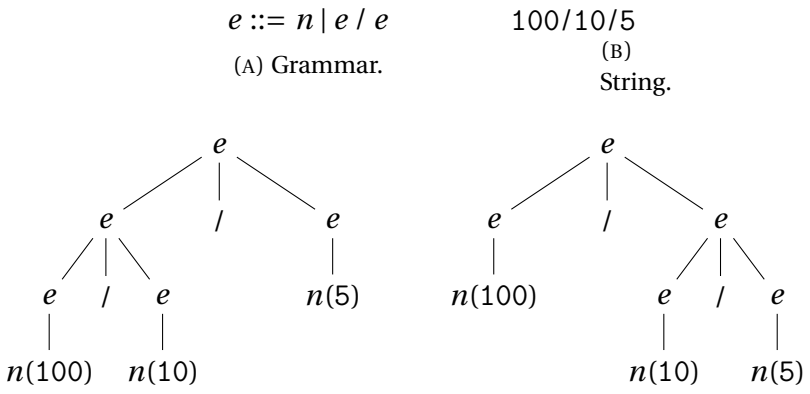
Which reading should it take? In particular, consider the grammar

$$e ::= n \mid e \,/\, e$$

where n is the terminal for numbers. We can diagram the two ways of reading the string `100/10/5` as shown in Figures 2.1c and 2.1d where we write the lexemes for the n tokens in parentheses for clarity. These diagrams are called *parse trees*, and they are another way to demonstrate that a string is the language described by a grammar. In a parse tree, a parent node corresponds to a non-terminal where its children correspond to the sequence of symbols in a production of that non-terminal. Parse trees capture syntactic structure and distinguishes between the two ways of "reading" `100/10/5`. We call the grammar given above *ambiguous* because we can witness a string that is "read" in two ways by giving two parse trees for it.

As an aside, in this way, a parse tree can be viewed as "recognizing" a string by a grammar in a "bottom-up manner." In contrast, derivations intuitively capture generating strings described a grammar in a "top-down manner."

Can we rewrite the above grammar to make it unambiguous? That is, can we rewrite the above grammar such that the set of strings accepted



(A) Grammar. (B) String. (C) Parse tree. (D) Parse tree.

FIGURE 2.1. An ambiguous grammar is exhibited by two parse trees for a string in the language described by the grammar.

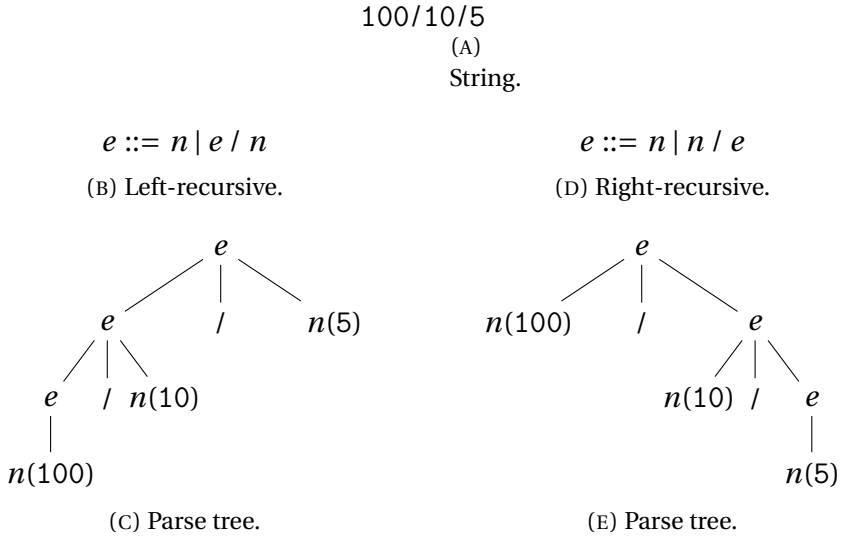
Ambiguous	Unambiguous	
	Left Recursive	Right Recursive
$e ::= n \mid e \,/\, e$	$e ::= n \mid e \,/\, n$	$e ::= n \mid n \,/\, e$

FIGURE 2.2. Rewriting a grammar to eliminate ambiguity with respect to associativity.

by the grammar is the same but is also unambiguous. Yes, we can rewrite the above grammar in two ways to eliminate ambiguity as shown in Figure 2.2. One grammar is *left recursive*, that is, the production $e ::= e \,/\, n$ is recursive only on the left of the binary operator token /. Analogously, we can write a *right recursive* grammar that accepts the same strings. Intuitively, these grammars enforce a particular linearization of the possible parse trees: either to the left or to the right as shown in Figure 2.3. As a terminological shorthand, we say that a binary operator is *left associative* to mean that expression trees involving that operator are linearized to the left, as in Figure 2.3c). Analogously, a binary operator is *right associative* means expression trees involving that operator are linearized to the right, as in Figure 2.3e.

A related syntactic issue appears when we consider multiple operators, such as the ambiguous grammar in Figure 2.4. For example, the string

$$10 - 10/10$$



(A) String.

(B) Left-recursive.

(C) Parse tree.

(D) Right-recursive.

(E) Parse tree.

FIGURE 2.3. Grammars that enforce a particular associativity.

Ambiguous	Unambiguous
$e ::= n \mid e \,/\, e \mid e - e$	$e ::= f \mid e - f$ $f ::= n \mid f \,/\, n$

FIGURE 2.4. Rewriting a grammar to eliminate ambiguity and enforce a particular associativity (left for both operators) and precedence (/ higher than −).

has two parse trees corresponding to the following two readings:

$$(10-10)/10 \qquad \text{or} \qquad 10-(10/10)\,.$$

We may want to enforce that the / operator "binds tighter," that is, has *higher precedence* than the − operator, which corresponds to the reading on the right. To enforce the desired precedence, we can refactor the ambiguous grammar into the unambiguous one shown in Figure 2.4. We layer the grammar by introducing a new non-terminal f that describes expressions with only / operator. The non-terminal f is left recursive, so we enforce that / is left associative. The start non-terminal e can be either an f or an expression with a − operator. Intuitively from a top-down, derivation perspective, once $e \Longrightarrow f$, then there is no way to derive a − operator. Thus, in any parse tree for a string that includes both − and / operators, the − operators must be "higher" in the tree. Note that *higher*

precedence means "binding tighter" or "lower in the parse tree" and similarly for *lower precedence*.

An important observation is that ambiguity is a syntactic concern: which tree do we get when we parse a string? This concern is different than and distinct with respect to what do the / or the − operators mean (e.g., perhaps division and subtraction), that is, the *semantics* of our expression language or to what *value* does an expression *evaluate*. The issue is the same if we consider a language with a pair operators that have a less ingrained meaning, such as @ and #.

If we know semantics of the language, then we can sometimes probe to determine associativity or precedence. For example, let us suppose we are interested in seeing what is relative precedence of the / and − operators in Scala. Knowing that / means division and − means subtraction, then observing the value of the expression `10 - 10/10` tells us the relative precedence of these two operators. Specifically, if the value is 9, then / has higher precedence, but if the value is 0, then − has higher precedence.

2.1.4. Abstract Syntax. Consider again the grammar of expressions involving the / and − operators in Figure 2.4, with subscripts to make explicit the instances of the symbols:

$$e ::= n \mid e_1 \ / \ e_2 \mid e_1 - e_2$$

To represent expressions e in Scala, we declare the following types and **case class**es:

```
sealed abstract class Expr
case class N(n: Int) extends Expr
case class Divide(e1: Expr, e2: Expr) extends Expr
case class Minus(e1: Expr, e2: Expr) extends Expr
```

We define a new type `Expr` (i.e., an **abstract class**). Each **case class** is a constructor for an expression e of type `Expr` corresponding to one of the productions defining the non-terminal e.

If we rewrite the above grammar to use these constructor names in each production, we get the following:

$$e \in \texttt{Expr} ::= \texttt{N}(n) \mid \texttt{Divide}(e_1, e_2) \mid \texttt{Minus}(e_1, e_2)\,.$$

An example sentence in this language is

$$\texttt{Minus}(\texttt{N}(10), \texttt{Divide}(\texttt{N}(10), \texttt{N}(10)))\,,$$

which corresponds to the following sentence in the first grammar:

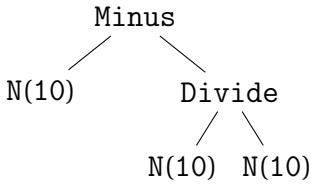
$$10 - 10/10\,.$$

Observe that a different sentence in the second grammar

$$\texttt{Divide}(\texttt{Minus}(\texttt{N}(10), \texttt{N}(10)), \texttt{N}(10))$$

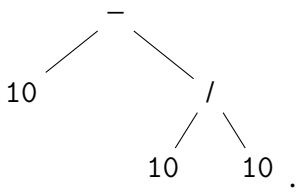
also corresponds to `10 – 10/10`. While the first grammar is ambiguous, the second one is unambiguous.

In a language implementation, we do not want to be constantly worrying about the "grouping" or parsing of a string (i.e., resolving ambiguity), so we prefer to work with *terms* in this second grammar. We call this second grammar, *abstract syntax*, where the tree structure is evident. Each instance of **case class** is a node in an n-ary tree and each argument of a non-terminal type to a constructor is a sub-tree. For example, the term `Minus(N(10), Divide(N(10), N(10)))` can be read visually as the following:



And thus the first phase of language tool is the *parser* that converts the concrete syntax of strings into the abstract syntax of terms.

Because the concrete syntax is more concise visually and human friendly, it is standard practice to give (ambiguous) grammars like the first grammar above and treat them as the corresponding abstract syntax specification given in the second grammar. In other words, we give a grammar that define the strings of a language and leave it as an implementation detail of the parser to convert strings to the appropriate terms or *abstract syntax trees*. We even often draw abstract syntax trees using concrete syntax notation, such as



2.2. Structural Induction

As we have seen, a convenient way to represent programs for manipulation by algorithms are as abstract syntax trees. In Scala, case classes are particularly useful for representing trees. Operations over trees are naturally given by recursive traversals, and pattern matching makes it easier to implement such recursive walks over abstract syntax trees.

In section 1.3, we saw that there is a tight connection between recursive programs and inductive reasoning. Learning to think inductively enables us to more easily implement correct, complex recursive traversals that are requisite for implementing any language tools.

Here, we introduce the concept of structural induction that enables us to reason not just over natural numbers but over any inductive type. Like in subsection 1.3.1, we focus on proving properties of recursive programs really with the goal of learning to think inductively.

2.2.1. Structural Induction over Lists. Consider the definition of Scala lists (simplified):

```
sealed abstract class List[T]
case object Nil extends List[Nothing]
case class ::[T](head: T, tail: List[T]) extends List[T]
```

Let us consider the possibles values of type `List[Int]`, for example. Here's what this set looks like schematically:

$$\begin{aligned}\{\quad & \texttt{Nil},\\ & \texttt{0}::\texttt{Nil},\ \ \texttt{1}::\texttt{Nil},\ \ -\texttt{1}::\texttt{Nil},\ \ \ldots,\\ & \texttt{0}::\texttt{0}::\texttt{Nil},\ \ \texttt{1}::\texttt{0}::\texttt{Nil},\ \ -\texttt{1}::\texttt{0}::\texttt{Nil},\ \ \ldots,\\ & \texttt{0}::\texttt{1}::\texttt{Nil},\ \ \texttt{1}::\texttt{1}::\texttt{Nil},\ \ -\texttt{1}::\texttt{1}::\texttt{Nil},\ \ \ldots \quad \}\end{aligned}$$

This set is inductively defined and can be viewed as generated by applying "rules" corresponding to `Nil` and ::. Thus, we have an induction principle for proving properties about `List[T]` values. Intuitively, to prove a property P over all `List[T]` values (i.e., $\forall l : \texttt{List[T]}.P(l)$), we consider two cases: (a) we prove that the property holds for `Nil`, that is, $P(\texttt{Nil})$—a base case; and (b) we prove that the property holds for $h::t$ assuming it holds for t (for any $h : \texttt{T}$ and any $t : \texttt{List[T]}$), that is, $\forall h : \texttt{T}, t : \texttt{List[T]}.(P(t) \implies P(h :: t))$—an inductive case. Notice the similarity to mathematical induction described in subsection 1.3.1. In fact, mathematical induction is simply a special case of structural induction over natural numbers.

As an example, consider the definition of list `append` given in Listing 2.1. Let us show that `append` terminates for any input. This property is extremely simple, but it illustrates the structure of such a proof.

LISTING 2.1. List Append

```
def append[T](xl: List[T], yl: List[T]): List[T] =
  xl match {
    case Nil => yl
    case xh :: xt => xh :: append(xt, yl)
  }
```

THEOREM 2.1 (Termination of `append`). *For all values xl and yl of type* `List[T]`,

$$\texttt{append}(xl, yl) \quad \longrightarrow^* \quad l$$

for some value l.

PROOF. By structural induction on *xl*.

BASE CASE ($xl = \texttt{Nil}$). Taking a few steps of evaluation, we have that

$$\begin{aligned}\texttt{append}(xl, yl) &= \texttt{append}(\texttt{Nil}, yl)\\ &\longrightarrow^* yl \quad \text{(by the definition of append)}\end{aligned}$$

Note that *yl* is a value, so this case is complete.

INDUCTIVE CASE ($xl = h::t$ for some values $h : \texttt{T}$ and $t : \texttt{List[T]}$). The induction hypothesis is as follows:

$$\texttt{append}(t, yl) \quad \longrightarrow^* \quad tl \qquad \text{for some value } tl.$$

Let us evaluate $\texttt{append}(xl, yl)$ a few steps, and we have the following:

$$\begin{aligned}\texttt{append}(xl, yl) &= \texttt{append}(h::t, yl)\\ &\longrightarrow^* h :: \texttt{append}(t, yl) \quad \text{(by the definition of append)}\end{aligned}$$

By the induction hypothesis (i.h.) on t, we have that

$$\texttt{append}(t, yl) \longrightarrow^* tl\,.$$

for some value *tl*. Thus, we have that

$$h :: \texttt{append}(t, yl) \longrightarrow^* h::tl$$

Note that $h::tl$ is a value, so this case is complete.

□

There are a number of key things to observe about the above proof. First, we chose *xl* as the induction variable. We have two inductively defined values *xl* and *yl* over which we are universally quantifying, so we could induct on either one. However, *xl* is the only choice that will allow us to complete this proof. Choosing the appropriate structure on which to apply induction can get tricky in general and sometimes requires trial-and-error. However, for correctness of recursive functions, choosing the value on which the recursion is over is almost always a good choice. In this case, `append` in Listing 2.1 is defined recursively over its first argument (i.e., `xl`).

Once the induction variable is chosen, the "proof template" is automatic. There is a case for each way the value could be constructed (e.g., `Nil` and :: in the case of `List[T]` values). Any case that has a subcomponent of the inductively-defined type is an inductive case (e.g., ::),

while any case that does not is a base case (e.g., `Nil`). Furthermore, the inductive hypothesis is also automatic in each inductive case—the property being proven is assumed to hold for any sub-component of the inductively-defined type (e.g., the tail of the list). Once you get familiar enough with induction, it becomes acceptable to leave off stating the induction hypothesis, as it is understood. When applying the induction hypothesis, it is important to state to which sub-component the induction hypothesis is being applied, as in some inductively-defined types there can be multiple sub-components. In the above, for example, we stated that the induction hypothesis was applied to the tail list t.

To see the "template" for such inductive proofs. Let us consider a proof of correctness of `append` stated somewhat informally. Again, this proof is extremely simple, but you can easily compare and contrast this proof with the one for Theorem 2.1.

THEOREM 2.2 (Correctness of `append`). *For all values xl and yl of type* `List[T]`,

$$\texttt{append}(xl, yl) \quad \longrightarrow^* \quad l$$

where l is the list value with the elements of xl followed by the elements of yl.

PROOF. By structural induction on *xl*.

BASE CASE ($xl = \texttt{Nil}$). Taking a few steps of evaluation, we have that

$$\begin{aligned}\texttt{append}(xl, yl) &= \texttt{append}(\texttt{Nil}, yl)\\ &\longrightarrow^* yl \quad \text{(by the definition of append)}\end{aligned}$$

Since *xl* is empty (i.e., `Nil`), the list containing the elements of *xl* followed by the elements of *yl* is exactly *yl*.

INDUCTIVE CASE ($xl = h::t$ for some values $h : \texttt{T}$ and $t : \texttt{List[T]}$). The induction hypothesis is as follows:

$$\texttt{append}(t, yl) \quad \longrightarrow^* \quad tl$$

where *tl* is the list value with the elements of t followed by the elements of *yl*.

Let us evaluate $\texttt{append}(xl, yl)$ a few steps, and we have the following:

$$\begin{aligned}\texttt{append}(xl, yl) &= \texttt{append}(h::t, yl)\\ &\longrightarrow^* h :: \texttt{append}(t, yl) \quad \text{(by the definition of append)}\end{aligned}$$

By the induction hypothesis (i.h.) on t, we have that

$$\texttt{append}(t, yl) \longrightarrow^* tl\,.$$

where *tl* is the list value with the elements of *t* followed by the elements of *yl*. Thus, we have that

$$h :: \texttt{append}(t, yl) \longrightarrow^* h::tl$$

Since $xl = h::t$, then $h::tl$ is the list with the elements of *xl* followed by the elements of *yl*.

□

2.2.2. Structural Induction over Abstract Syntax Trees. Recall that structural induction works for any inductively-defined type. Of particular importance to us is structural induction over abstract syntax trees. Consider the interpreter `eval` for the simple arithmetic language shown in Listing 2.2. We want to show that our interpreter implementation is correct.

LISTING 2.2. Evaluation of Arithmetic Expressions

expressions $e ::= n \mid -e_1 \mid e_1 + e_2$
integers n

```
sealed abstract class Expr
case class N(n: Int) extends Expr
case class Neg(e1: Expr) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case N(n) => n
  case Neg(e1) => eval(e1)
  case Plus(e1, e2) => eval(e1) + eval(e2)
}
```

Let us first define what it means to be correct. For any Scala value *e* of type `Expr`, we define a function $[\![e]\!]$ that gives the mathematical integer that we want to correspond to *e*. In particular, we inductively define this function as follows:

$$[\![\texttt{N}(n)]\!] \stackrel{\text{def}}{=} [\![n]\!]$$
$$[\![\texttt{Neg}(e_1)]\!] \stackrel{\text{def}}{=} -[\![e_1]\!]$$
$$[\![\texttt{Plus}(e_1, e_2)]\!] \stackrel{\text{def}}{=} [\![e_1]\!] + [\![e_2]\!]$$

where we assume a corresponding function $[\![n]\!]$ that takes a Scala value *n* of type `Int` to the corresponding mathematical integer. The $-$ and $+$ operators above are over mathematical integers. We can now state and prove the correctness of `eval`.

THEOREM 2.3 (Correctness of `eval`). *For all values e of type* `Expr`*,*

$$\texttt{eval}(e) \quad \longrightarrow^* \quad \llcorner [\![e]\!] \lrcorner$$

PROOF. By structural induction on e.

BASE CASE ($e = \texttt{N}(n)$ for some value $n : \texttt{Int}$). Taking a few steps of evaluation, we have that

$$\begin{aligned} \texttt{eval}(e) &= \texttt{eval}(\texttt{N}(n)) \\ &\longrightarrow^* n \quad \text{(by the definition of } \texttt{eval}) \end{aligned}$$

Note that $\llcorner [\![\texttt{N}(n)]\!] \lrcorner = \llcorner [\![n]\!] \lrcorner = n$, so this case is complete.

INDUCTIVE CASE ($e = \texttt{Neg}(e_1)$ for some value $e_1 : \texttt{Expr}$). Taking a few steps of evaluation, we have that

$$\begin{aligned} \texttt{eval}(e) &= \texttt{eval}(\texttt{Neg}(e_1)) \\ &\longrightarrow^* -\texttt{eval}(e_1) \quad \text{(by the definition of } \texttt{eval}) \end{aligned}$$

By the i.h. on e_1, we have that

$$\texttt{eval}(e_1) \longrightarrow^* \llcorner [\![e_1]\!] \lrcorner .$$

Thus, we have that

$$\begin{aligned} -\texttt{eval}(e_1) &\longrightarrow^* -\llcorner [\![e_1]\!] \lrcorner \\ &\longrightarrow^* \llcorner -[\![e_1]\!] \lrcorner \quad \text{(by the definition of } - \text{ in Scala}) \end{aligned}$$

Note that $\llcorner [\![\texttt{Neg}(e_1)]\!] \lrcorner = \llcorner -[\![e_1]\!] \lrcorner$, so this case is complete.

INDUCTIVE CASE ($e = \texttt{Plus}(e_1, e_2)$ for some values $e_1 : \texttt{Expr}$ and $e_2 : \texttt{Expr}$). Taking a few steps of evaluation, we have that

$$\begin{aligned} \texttt{eval}(e) &= \texttt{eval}(\texttt{Plus}(e_1, e_2)) \\ &\longrightarrow^* \texttt{eval}(e_1) + \texttt{eval}(e_1) \quad \text{(by the definition of } \texttt{eval}) \end{aligned}$$

By the i.h. on e_1, we have that

$$\texttt{eval}(e_1) \longrightarrow^* \llcorner [\![e_1]\!] \lrcorner .$$

And by the i.h. on e_2, we have that

$$\texttt{eval}(e_2) \longrightarrow^* \llcorner [\![e_2]\!] \lrcorner .$$

Thus, we have that

$$\begin{aligned} \texttt{eval}(e_1) + \texttt{eval}(e_1) &\longrightarrow^* \llcorner [\![e_1]\!] \lrcorner + \llcorner [\![e_2]\!] \lrcorner \\ &\longrightarrow^* \llcorner [\![e_1]\!] + [\![e_2]\!] \lrcorner \quad \text{(by the definition of } + \text{ in Scala}) \end{aligned}$$

Note that $\llcorner [\![\texttt{Plus}(e_1, e_2)]\!] \lrcorner = \llcorner [\![e_1]\!] + [\![e_2]\!] \lrcorner$, so this case is complete.

□

informal description	set of syntactic objects	grammar
natural numbers	**Nat**	$n ::= \mathsf{z} \mid \mathsf{s}(n)$

FIGURE 2.5. A language describing the natural numbers.

2.3. Judgments

A *judgment* is a statement about a set of objects. It asserts a relation between a set of objects. The relation itself is often called a *judgment form*. Judgments are used pervasively in describing programming languages.

We have previously seen judgment forms, for example, relating an expression and a type:

$$e : \tau$$

that is read "expression e has type τ." This relation takes two parameters: an expression e and a type τ. The colon : is simply punctuation. For readability, it is common for judgment forms to use a mix of punctuation symbols. Parameters are typically written in italic font (e.g., e and τ).

Judgment forms are defined inductively using a set of *inference rules*. An inference rule takes the following form:

$$\frac{J_1 \qquad J_2 \qquad \cdots \qquad J_n}{J}$$

where the meta-variable J stands for a judgment. The judgments above the horizontal line are the *premises*, while the judgment below the line is the *conclusion*. An inference rule states that if the premises can be shown to hold, then the conclusion also holds (i.e., the premises are sufficient to *derive* the conclusion). The set of premises may be empty, and such an inference rule is called an *axiom*.

2.3.1. Example: Syntax. Recall from section 2.1 that a grammar defines inductively a set of syntactic objects. For example, we can describe the natural numbers using a unary notation in Figure 2.5. We give an explicit name **Nat** to the set of syntactic objects describing natural numbers.

We can also define the language of natural numbers using judgments and inference rules. Let $n \in \mathbf{Nat}$ be the (unary) judgment that says, "Syntactic object n is a natural number in set **Nat**." We define this judgment in Figure 2.6 with two inference rules ZERO and SUCCESSOR. Rule ZERO is an axiom that says that z is in set **Nat**, while rule SUCCESSOR says that $\mathsf{s}(n)$ is in set **Nat** if n is in **Nat**.

$\boxed{n \in \mathbf{Nat}}$

$$\frac{}{\mathsf{z} \in \mathbf{Nat}}\ \text{ZERO} \qquad \frac{n \in \mathbf{Nat}.}{\mathsf{s}(n) \in \mathbf{Nat}}\ \text{SUCCESSOR}$$

FIGURE 2.6. Defining the language of natural numbers judgmentally.

$\boxed{n_1 =_{\mathbf{Nat}} n_2}$

$$\frac{}{\mathsf{z} =_{\mathbf{Nat}} \mathsf{z}}\ \text{ZERO-EQ} \qquad \frac{n_1 =_{\mathbf{Nat}} n_2}{\mathsf{s}(n_1) =_{\mathbf{Nat}} \mathsf{s}(n_2)}\ \text{SUCCESSOR-EQ}$$

FIGURE 2.7. Defining structural equality of natural numbers.

2.3.2. Derivations of Judgments. A set of inference rules defines a judgment as the *least* relation *closed under* the rules. This statement means a judgment holds if and only if we can compose applications of the inference rules to demonstrate it. Such a demonstration is called a *derivation*. A *derivation* is a tree where each node in the tree is an application of an inference rule and whose children are derivations of the rule's premises. The leaves of a derivation tree are applications of axioms.

For example, to demonstrate that the judgment $\mathsf{s}(\mathsf{s}(\mathsf{z})) \in \mathbf{Nat}$ holds, we give the following the derivation:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{z} \in \mathbf{Nat}}\ \text{ZERO}}{\mathsf{s}(\mathsf{z}) \in \mathbf{Nat}}\ \text{SUCCESSOR}}{\mathsf{s}(\mathsf{s}(\mathsf{z})) \in \mathbf{Nat}}\ \text{SUCCESSOR} \quad .$$

We write the rule that is applied to the right of the horizontal line.

2.3.3. Structural Induction on Derivations. Judgments are inductively-defined relations. They yield an induction principle based on the structure of derivations. In particular, to show a property $P(J)$ whenever J holds, it suffices to consider each rule from which J may be derived:

$$\frac{J_1 \qquad J_2 \qquad \cdots \qquad J_n}{J}$$

and show $P(J)$ under the inductive hypotheses $P(J_1)$, $P(J_2)$, ..., and $P(J_n)$.

To give an example of structural induction on derivations, let us first explicitly define structural equality over our language of natural numbers with the judgment form $n_1 =_{\mathbf{Nat}} n_2$ in Figure 2.7.

Now, let us prove that $=_{\mathbf{Nat}}$ is reflexive.

> *If* $n \in \mathbf{Nat}$*, then* $n =_{\mathbf{Nat}} n$.

Note that this statement states that if the judgment $n \in \mathbf{Nat}$ holds, then the judgment $n =_{\mathbf{Nat}} n$ also holds. A judgment holds if and only if there is a derivation that exhibits it, so more verbosely, the above is a shorthand for the following:

> *For all* n *and all derivations* $\mathscr{D}$*, if* $\mathscr{D}$ *is a derivation for the judgment* $n \in \mathbf{Nat}$*, then there is a derivation* $\mathscr{E}$ *for the judgment* $n =_{\mathbf{Nat}} n$.

To annotate a judgment with a derivation that exhibits it, we write $\mathscr{D} :: J$ to mean $\mathscr{D}$ is a derivation that ends in judgment J.

THEOREM 2.4 ($=_{\mathbf{Nat}}$ is a reflexive). *If* $\mathscr{D} :: n \in \mathbf{Nat}$*, then* $n =_{\mathbf{Nat}} n$.

PROOF. By structural induction on $\mathscr{D}$.

CASE ($\mathscr{D} = \dfrac{}{\mathsf{z} \in \mathbf{Nat}}\ \text{ZERO}$). We create a derivation $\mathscr{E}$ to exhibit the judgment $\mathsf{z} =_{\mathbf{Nat}} \mathsf{z}$ by applying rule ZERO-EQ:

$$\mathscr{E} = \quad \dfrac{}{\mathsf{z} =_{\mathbf{Nat}} \mathsf{z}}\ \text{ZERO-EQ} \quad .$$

CASE ($\mathscr{D} = \dfrac{\mathscr{D}_1 :: n' \in \mathbf{Nat}}{\mathsf{s}(n') \in \mathbf{Nat}}\ \text{SUCCESSOR}$). By the induction hypothesis on $\mathscr{D}_1$, we have a derivation $\mathscr{E}_1$ that exhibits the judgment $n' =_{\mathbf{Nat}} n'$. We create a derivation $\mathscr{E}$ that exhibits the judgment $\mathsf{s}(n') =_{\mathbf{Nat}} \mathsf{s}(n')$ by applying rule SUCCESSOR-EQ as follows:

$$\mathscr{E} = \quad \dfrac{\mathscr{E}_1 :: n' =_{\mathbf{Nat}} n'}{\mathsf{s}(n') =_{\mathbf{Nat}} \mathsf{s}(n')}\ \text{SUCCESSOR-EQ} \quad .$$

□

This theorem is rather obvious, but it highlights the structure of such proofs. Here, we have been very explicit about the manipulation of derivations. Because the existence of a derivation coincides with a judgment holding, derivations are sometimes left more implicit.

CHAPTER 3

Language Design and Implementation

3.1. Operational Semantics

In section 2.1, we began the discussion of language specification and the importance specifying languages clearly, crisply, and precisely. Grammars is the main tool by which the *syntax* of a language, that is, the programs that we can write are specified. In this section, we introduce a tool for defining the *semantics* of a language, that is, the meaning of programs.

There are several ways to think about the meaning of programs. One natural way is to think about how programs evaluate. An *operational semantics* is a way to describe how programs evaluate in terms of the language itself (rather than by compilation to a machine model). One way to see an operational semantics is as describing an interpreter for the language of interest.

3.1.1. Syntax: JavaScripty. We consider a small subset of JavaScript, which we will affectionately call JAVASCRIPTY. The syntax of JAVASCRIPTY is given in Figure 3.1. Recall that we interpret such a definition as the abstract syntax of JAVASCRIPTY using elements from its concrete syntax. That is, we write concrete syntax for readability but assume that we are given abstract syntax trees that resolve ambiguity in the grammar.

expressions	$e ::= x \mid n \mid b \mid \textbf{undefined} \mid uop\, e_1 \mid e_1\ bop\ e_2$
	$\quad \mid e_1 \,?\, e_2 : e_3 \mid \textbf{const}\ x = e_1; e_2 \mid \textbf{console.log}(e_1)$
values	$v ::= n \mid b \mid \textbf{undefined}$
unary operators	$uop ::= \texttt{-} \mid \texttt{!}$
binary operators	$bop ::= \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}$
	$\quad \mid \texttt{===} \mid \texttt{!==} \mid \texttt{\&\&} \mid \texttt{\|\|}$
variables	x
numbers (doubles)	n
booleans	$b ::= \textbf{true} \mid \textbf{false}$

FIGURE 3.1. Syntax of JAVASCRIPTY

Figure 3.1 describes JAVASCRIPTY using a number of syntactic categories. The main syntactic category is expressions. We consider a program to be an expression. Expressions e consist of variables, value literals, unary operator expressions, binary operator expressions, a conditional if-then-else expression, a variable binding expression, and a print expression. Values v can be numbers (double-precision floating point), booleans, and a unique undefined value. This set of essentially arithmetic expressions is the usual core of any programming language. Functions are notably missing.

3.1.2. A Big-Step Operational Semantics. We might guess the semantics of particular expressions based on common conventions. For example, we probably guess that expression

$$e_1 + e_2$$

adds two numbers that result from evaluating e_1 and e_2. However, note that this statement is something about the semantics of $e_1 + e_2$, which has yet to be specified.

One aspect that makes the JavaScript specification complex is the presence of implicit conversions (e.g., boolean values may be implicitly converted to numeric values depending on the context in which values are used). For example,

true + 2

evaluates to 3. How can we describe how to implement a JAVASCRIPTY interpreter for all programs?

It is possible to specify the semantics of a programming language using natural language prose. However, just like with specifying syntax using natural language prose, it is very easy to leave ambiguity in the description. Furthermore, trying to minimize ambiguity can create very verbose descriptions. The JavaScript specification, specifically ECMA-262 standard [], is actually rather precise specification based on natural language prose, but the descriptions are quite verbose.

In this section, we introduce some mathematical notation that enables us to specify semantics with less ambiguity in a very compact form. Like any mathematical notation, its precise and compact nature makes it easier, for example, to spot errors or inconsistencies in specification. However, there will necessarily be a learning curve to reading the notation.

We want to write out as unambiguously as possible how a program should evaluate independent of an implementation (e.g., a compiler and

machine architecture). We use a method specification known as an *operational semantics*. An operational semantics can be thought as describing an interpreter for the language of interest with relations between syntactic objects. We have already used a notation for describing an evaluation relation:

$$e \Downarrow v\,.$$

This notation is a judgment form stating informally, "Expression e evaluates to value v." Defining this judgment describes how to evaluate expressions to values and thus corresponds closely to writing a recursive interpreter of the abstract syntax trees representing expressions.

We will use a slight richer judgment form with an additional parameter:

$$E \vdash e \Downarrow v\,,$$

which says informally, "In value environment E, expression e evaluates to value v." This relation has three parameters: E, e, and v. The other parts of the judgment is simply punctuation that separates the parameters. The $\vdash$ symbol is called the "turnstile" symbol.

A value environment E is a finite map from variables x to values v and can be described by the following grammar:

$$E ::= \cdot \mid E[x \mapsto v]\,.$$

We write $\cdot$ for the empty environment and $E[x \mapsto v]$ as the environment that maps x to v but is otherwise the same as E (i.e., extends E with mapping x to v). Additionally, we write $E(x)$ for looking up the value of x in environment E. More precisely, we can define look up as follows:

$$\begin{array}{rcll} E[y \mapsto v](x) & \stackrel{\text{def}}{=} & v & \text{if } y = x \\ E[y \mapsto v](x) & \stackrel{\text{def}}{=} & E(x) & \text{otherwise} \\ \cdot(x) & & & \text{undefined}\,. \end{array}$$

The inference rules that define this evaluation judgment form is given in Figure 3.2. Let us first consider the two axioms:

$$\textsc{EvalVar}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \textsc{EvalVal}\ \frac{}{E \vdash v \Downarrow v}\,.$$

The EvalVar rule says that a variable use x evaluates to the value to which it is bound in the environment E. Or operationally, to evaluate a variable use x, look up the value corresponding to x in the environment E. For an expression that is already a value v, it evaluates to itself as stated by the EvalVal rule.

Now, back to the original example in this section, we are trying to specify how the expression

$$e_1 + e_2$$

evaluates. Thinking operationally, we want to say something like: evaluate e_1 to a number, evaluate e_2 to a number, and then return the number that is the addition of those numbers. Consider the following inference rule:

$$\frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2 \qquad n' = n_1 + n_2}{E \vdash e_1 + e_2 \Downarrow n'}$$

Reading top-down, this rule says if we know that in environment E, expression e_1 evaluates to a number n_1 and e_2 evaluates to n_2, then expression e_1 + e_2 evaluates to n' in environment E where n' is the addition of the n_1 and n_2. Note that the + in the premise is "plus" in the meta language (i.e., the implementation language) in contrast to the + in the conclusion that is the syntactic symbol + in the object language (i.e., the source language). Here, we have highlighted the meta-language "plus" for clarity, but often, the reader is asked to determine this distinction based on context. To be completely explicit, we could use an alternative notation for the abstract syntax:

$$\frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2 \qquad n' = n_1 + n_2}{E \vdash \texttt{Binary(Plus, } e_1\texttt{, } e_2\texttt{)} \Downarrow n'}$$

This rule defines a semantics that does not fully match JavaScript because it requires e_1 and e_2 in e_1 + e_2 to evaluate to number values (i.e., n_1 and n_2). JavaScript permits other types of values and then performs a conversion before performing the addition. We can express this semantics using the following inference rule:

$$\frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad n' = \text{toNumber}(v_1) + \text{toNumber}(v_2)}{E \vdash e_1 + e_2 \Downarrow n'}$$

Reading top-down, it says if we know that in environment E, expression e_1 evaluates to v_1 and e_2 evaluates to v_2, then expression e_1 + e_2 evaluates to n' in environment E where n' is the addition of the toNumber conversions of v_1 and v_2. We define the toNumber conversion in Figure 3.3. This rule specifies the semantics that we want, though it is not the only way to do so.

Any evaluation rule can also be read bottom-up, which matches more closely to an implementation. For example, the above rule says, "To evaluate e_1 + e_2 in environment E, evaluate e_1 in environment E to get value v_1, evaluate e_2 in environment E to get value v_2, convert v_1 and v_2 to numbers, and return the addition of those two numbers."

In Figure 3.2, we lump all of the arithmetic operators +, −, ∗, and / together in the same rule: EVALARITH. We use one notational shortcut

$\boxed{E \vdash e \Downarrow v}$

$$\textsc{EvalVar}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \textsc{EvalVal}\ \frac{}{E \vdash v \Downarrow v} \qquad \textsc{EvalNeg}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad n' = -\,\mathrm{toNumber}(v_1)}{E \vdash -e_1 \Downarrow n'}$$

$$\textsc{EvalNot}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad b' = \neg\,\mathrm{toBoolean}(v_1)}{E \vdash !e_1 \Downarrow b'} \qquad \textsc{EvalSeq}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\,, e_2 \Downarrow v_2}$$

$$\textsc{EvalArith}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad n' = \mathrm{toNumber}(v_1)\ bop\ \mathrm{toNumber}(v_2) \qquad bop \in \{+, -, *, /\}}{E \vdash e_1\ bop\ e_2 \Downarrow n'}$$

$$\textsc{EvalInequality}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad b' = \mathrm{toNumber}(v_1)\ bop\ \mathrm{toNumber}(v_2) \qquad bop \in \{<, <=, >, >=\}}{E \vdash e_1\ bop\ e_2 \Downarrow b'}$$

$$\textsc{EvalEquality}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad b' = (v_1\ bop\ v_2) \qquad bop \in \{===, !==\}}{E \vdash e_1\ bop\ e_2 \Downarrow b'}$$

$$\textsc{EvalAndTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{true} = \mathrm{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ \&\&\ e_2 \Downarrow v_2} \qquad \textsc{EvalAndFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{false} = \mathrm{toBoolean}(v_1)}{E \vdash e_1\ \&\&\ e_2 \Downarrow v_1}$$

$$\textsc{EvalOrTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{true} = \mathrm{toBoolean}(v_1)}{E \vdash e_1\ ||\ e_2 \Downarrow v_1} \qquad \textsc{EvalOrFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{false} = \mathrm{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ ||\ e_2 \Downarrow v_2}$$

$$\textsc{EvalPrint}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1\ \text{printed}}{E \vdash \mathbf{console.log}(e_1) \Downarrow \mathbf{undefined}} \qquad \textsc{EvalIfTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{true} = \mathrm{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ ?\ e_2 : e_3 \Downarrow v_2}$$

$$\textsc{EvalIfFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad \mathbf{false} = \mathrm{toBoolean}(v_1) \qquad E \vdash e_3 \Downarrow v_3}{E \vdash e_1\ ?\ e_2 : e_3 \Downarrow v_3} \qquad \textsc{EvalConst}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{E \vdash \mathbf{const}\ x = e_1; e_2 \Downarrow v_2}$$

FIGURE 3.2. Big-step operational semantics of JAVASCRIPTY.

by treating the *bop* as the corresponding meta-language operator in the premise.

It is informative to study the complete set of inference rules and think about how the rules correspond to implementing an interpreter. The EVALCONST rule is particularly interesting because we see explicitly that

$$\mathbf{const}\ x = e_1; e_2$$

the scope of variable x is the expression e_2 because e_2 is evaluated in an extended environment with a binding for x.

$$
\begin{array}{rcll}
\text{toNumber}(n) & \stackrel{\text{def}}{=} & n & \\
\text{toNumber}(\textbf{true}) & \stackrel{\text{def}}{=} & 1 & \\
\text{toNumber}(\textbf{false}) & \stackrel{\text{def}}{=} & 0 & \\
\text{toNumber}(\textbf{undefined}) & \stackrel{\text{def}}{=} & \mathsf{NaN} & \\
\text{toBoolean}(n) & \stackrel{\text{def}}{=} & \textbf{false} & \text{if } n = 0 \text{ or } n = \mathsf{NaN} \\
\text{toBoolean}(n) & \stackrel{\text{def}}{=} & \textbf{true} & \text{otherwise} \\
\text{toBoolean}(b) & \stackrel{\text{def}}{=} & b & \\
\text{toBoolean}(\textbf{undefined}) & \stackrel{\text{def}}{=} & \textbf{false} &
\end{array}
$$

FIGURE 3.3. JAVASCRIPTY conversion functions.

3.2. Small-Step Operational Semantics

3.2.1. Evaluation Order. In subsection 3.1.2, we have carefully specified several aspects of how the expression

$$e_1 + e_2$$

should be evaluated. In essence, it adds two integers that result from evaluating e_1 and e_2. However, there is still at least one more semantic question that we have not specified, "Is e_1 evaluated first and then e_2 or vice versa, or are they evaluated concurrently?"

Why does this question matter? Consider the expression:

(3.2.1.1) `(jsy.print(1), 1) + (jsy.print(2), 2).`

The , operator is a sequencing operator. In particular, e_1 , e_2 first evaluates e_1 to a value and then evaluates e_2 to value; the value of the whole expression is the value of e_2, while the value of e_1 is simply thrown away. Furthermore, **console**.**log**(e_1) evaluates its argument to a value and then prints to the screen a representation of that value. If the left operand of a + is evaluated first before the right operand, then the above expression (3.2.1.1) prints 1 and then 2. If the operands of + are evaluated in the opposite order, then 2 is printed first followed by 1. Note that the final value is 3 regardless of the evaluation order.

The evaluation order matters because the **console**.**log**(e_1) expression has a *side effect*. It prints to the screen. As alluded to in section 1.2, an expression free of side effects (i.e., is pure) has the advantage that the evaluation order cannot be observed (i.e., does not matter from the programmer's perspective). Having this property is also known as being *referentially transparent*, that is, taking an expression and replacing any of its subexpressions by the subexpression's value cannot be observed as evaluating any differently than evaluating the expression itself. In JAVASCRIPTY, our only side-effecting expression is **console**.**log**(e_1). If we remove the

prints from the above expression (3.2.1.1), then the evaluation order cannot be observed.

3.2.2. A Small-Step Operational Semantics of JAVASCRIPTY. The big-step operational semantics given in subsection 3.1.2 does give us a nice specification for implementing an interpreter, but it does leave some semantic choices like evaluation order implicit. Intuitively, it specifies what the value of an expression should be (if it exists) but not precisely the steps to get to the value.

We have already used a notation for describing a one-step evaluation relation:

$$e \longrightarrow e' .$$

This notation is a judgment form stating informally, "Expression e can take one step of evaluation to expression e'." Defining this judgment allows us to more precisely state how to take one step of evaluation, that is, how to make a single *reduction* step. Once we know how to reduce expressions, we can evaluate an expression e by repeatedly applying reduction until reaching a value. Thus, such a definition describes an operational semantics and intuitively an interpreter for expressions e. This style of operational semantics where we specify reduction steps is called a *small-step operational semantics*.

In contrast to subsection 3.1.2, we will not extend this judgment form with value environments. Instead, we define the one-step reduction relation on *closed* expressions, that is, expressions without any free variables. If we require the "top-level" program to be a closed expression, then we can ensure reduction only sees closed expressions by intuitively "applying the environment" eagerly via *substitution*. That is, variable uses are replaced by the values to which they are bound before reduction gets to them. As an example, we will define reduction so that the following judgment holds:

$$\textbf{const}\ \texttt{x = 1; x + x} \longrightarrow \texttt{1 + 1}.$$

This choice to use substitution instead of explicit environments is orthogonal to specifying the semantics using small-step or big-step (i.e., one could use subsitution with big-step or environments with small-step). Explicit environments just get a bit more unwieldy here.

First, we need to describe what action does an operation perform. For example, we want to say that the + operator adds two numbers, which we say with the following rule:

$$\frac{n' = \text{toNumber}(v_1) + \text{toNumber}(v_2)}{v_1 + v_2 \longrightarrow n'}\ \textsc{DoPlus}$$

This rules says the expression $v_1 + v_2$ reduces in one step to an integer value n' that is the addition of the toNumber conversion of values v_1 and v_2. We use the meta-variables v_1, v_2, and n' to express constraints that particular positions in the expressions must be values or numeric values. Note that the + in the conclusion is the syntactic + operator, while the + in the premise expresses mathematical addition of two numbers. As we discussed in subsection 3.1.2, this symbol clash is rather unfortunate, but context usually allows us to determine which + is which. We sometimes call this kind of rule that performs an operation a *local reduction* rule. We will prefix all rules for this kind of rule with DO (and so will sometimes call them DO rules).

Second, we need to describe how we find the next operation to perform. These rules will capture issues like evaluation order described informally in subsection 3.2.1. To specify that $e_1 + e_2$ should be evaluated left-to-right, we use the following two rules:

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \text{SEARCHPLUS}_1 \qquad \frac{e_2 \longrightarrow e_2'}{v_1 + e_2 \longrightarrow v_1 + e_2'}\ \text{SEARCHPLUS}_2$$

The SEARCHPLUS$_1$ rule states for an arbitrary expression of the form $e_1 + e_2$, if e_1 steps to e_1', then the whole expression steps to $e_1' + e_2$. We can view this rule as saying that we should look for an operation to perform somewhere in e_1. The rest of the expression $\bullet + e_2$ is a context that gets carried over untouched. The SEARCHPLUS$_2$ rule is similar except that it applies only if the left expression is a value (i.e., $v_1 + e_2$). Together, these rules capture precisely a left-to-right evaluation order for an expression of the form $e_1 + e_2$ because (1) if e_1 is not a value, then only SEARCHPLUS$_1$ could possibly apply, and (2) if e_1 is a value, then only SEARCHPLUS$_2$ could possibly apply. We sometimes call this kind of rule that finds the next operation to perform a *global reduction* rule (or a SEARCH rule). The subexpression that is the next operation to perform is called the *redex*.

Considering these three rules, there is at most one rule that applies that specifies the "next" step. If our set of inference rules defining reduction has this property, then we say that our reduction system is *deterministic*. In other words, there is always at most one "next" step. Determinism is a property that we could prove about certain reduction systems, which we can state formally as follows:

PROPERTY 3.1 (Determinism). *If $e \longrightarrow e'$ and $e \longrightarrow e''$, then $e' = e''$.*

In general, such a proof would proceed by structural induction on the derivation of the reduction step (i.e., $e \longrightarrow e'$). We do not yet such proofs here in detail (cf., subsection 2.3.3).

$\boxed{e \longrightarrow e'}$

$$\textsc{DoNeg}\ \frac{n' = -\,\text{toNumber}(v)}{-\,v \longrightarrow n'} \qquad \textsc{DoNot}\ \frac{b' = \neg\,\text{toBoolean}(v)}{!\,v \longrightarrow b'} \qquad \textsc{DoSeq}\ \frac{}{v_1\,,\,e_2 \longrightarrow e_2}$$

$$\textsc{DoArith}\ \frac{n' = \text{toNumber}(v_1)\ bop\ \text{toNumber}(v_2) \qquad bop \in \{+,-,*,/\}}{v_1\ bop\ v_2 \longrightarrow n'}$$

$$\textsc{DoInequality}\ \frac{b' = \text{toNumber}(v_1)\ bop\ \text{toNumber}(v_2) \qquad bop \in \{<,<=,>,>=\}}{v_1\ bop\ v_2 \longrightarrow b'}$$

$$\textsc{DoEquality}\ \frac{b' = (v_1\ bop\ v_2) \qquad bop \in \{===,!==\}}{v_1\ bop\ v_2 \longrightarrow b'} \qquad \textsc{DoAndTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ \&\&\ e_2 \longrightarrow e_2} \qquad \textsc{DoAndFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ \&\&\ e_2 \longrightarrow \textbf{false}}$$

$$\textsc{DoOrTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ ||\ e_2 \longrightarrow \textbf{true}} \qquad \textsc{DoOrFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ ||\ e_2 \longrightarrow e_2} \qquad \textsc{DoPrint}\ \frac{v_1\ \text{printed}}{\textbf{console.log}(v_1) \longrightarrow \textbf{undefined}}$$

$$\textsc{DoIfTrue}\ \frac{\textbf{true} = \text{toBoolean}(v_1)}{v_1\ ?\ e_2 : e_3 \longrightarrow e_2} \qquad \textsc{DoIfFalse}\ \frac{\textbf{false} = \text{toBoolean}(v_1)}{v_1\ ?\ e_2 : e_3 \longrightarrow e_3} \qquad \textsc{DoConst}\ \frac{}{\textbf{const}\ x = v_1; e_2 \longrightarrow e_2[v_1/x]}$$

$$\textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{uop\,e_1 \longrightarrow uop\,e_1'} \qquad \textsc{SearchBinary}_1\ \frac{e_1 \longrightarrow e_1'}{e_1\ bop\ e_2 \longrightarrow e_1'\ bop\ e_2} \qquad \textsc{SearchBinary}_2\ \frac{e_2 \longrightarrow e_2'}{v_1\ bop\ e_2 \longrightarrow v_1\ bop\ e_2'}$$

$$\textsc{SearchPrint}\ \frac{e_1 \longrightarrow e_1'}{\textbf{console.log}(e_1) \longrightarrow \textbf{console.log}(e_1')} \qquad \textsc{SearchIf}\ \frac{e_1 \longrightarrow e_1'}{e_1\ ?\ e_2 : e_3 \longrightarrow e_1'\ ?\ e_2 : e_3}$$

$$\textsc{SearchConst}\ \frac{e_1 \longrightarrow e_1'}{\textbf{const}\ x = e_1; e_2 \longrightarrow \textbf{const}\ x = e_1'; e_2}$$

FIGURE 3.4. Small-step operational semantics of JAVASCRIPTY.

In Figure 3.4, we give all of the inference rules that define the one-step evaluation relation $e \longrightarrow e'$ for JAVASCRIPTY. The DONEG states that the unary operator $-$ is integer negation, while DONOT states that ! is boolean negation. Observe that to "do" the operation, we require that the sub-expression under the unary operators $-$ or !, respectively, is a value. Contrast these rules to EVALNEG and EVALNOT in Figure 3.2. If the sub-expression under the unary operator is not a value, then instead the rule SEARCHUNARY applies telling us to look for something to reduce inside this sub-expression. The DOSEQ rules states that the ',' operator is used

to indicate sequencing: for e_1 , e_2, first e_1 is evaluated to a value, then that value is ignored, and we continue by evaluating e_2. The DoArith, DoInequality, and DoEquality specify how the arithmetic, inequality, and equality operators behave, respectively. The DoArith includes the case for + that we separated out in our discussion above.

We say that a *short-circuit evaluation* of expression is one where a value is produced before evaluating all subexpressions to values. The next four rules DoAndTrue, DoAndFalse, DoOrTrue, and DoOrFalse say that the boolean expressions e_1 && e_2 and e_1 || e_2 may short-circuit. In particular, the rule

$$\frac{\textsc{DoAndFalse}\quad \textbf{false} = \text{toBoolean}(v_1)}{v_1 \;\&\&\; e_2 \longrightarrow \textbf{false}}$$

says that v_1 && e_2 where v_1 converts to **false** evaluates to **false** without ever evaluating e_2. The analogous rule for || is

$$\frac{\textsc{DoOrTrue}\quad \textbf{true} = \text{toBoolean}(v_1)}{v_1 \;||\; e_2 \longrightarrow \textbf{true}}$$

The DoPrint rule

$$\frac{\textsc{DoPrint}\quad v_1 \text{ printed}}{\textbf{console.log}(v_1) \longrightarrow \textbf{undefined}}$$

is somewhat informal. In particular, since printing is outside of our model, the"v_1 printed" in the premise of the rule is not any required condition but should be viewed as comment for when this rule is applied. What is stated is the result of a **print** is the value **undefined**.

For e_1 ? e_2 : e_3, the rules DoIfTrue and DoIfFalse specify with which expression to continue evaluation in the expected way depending on what Boolean value to which the guard converts.

The DoConst rule for the variable binding expression **const** $x = e_1; e_2$

$$\frac{\textsc{DoConst}}{\textbf{const}\; x = v_1; e_2 \longrightarrow e_2[v_1/x]}$$

is a bit more interesting. The expression-to-be-bound should already be a value v_1. We then proceed with e_2 with the value v_1 replacing the variable x. In general, the notation $e_1[e_2/x]$ is read as capture-avoiding substitution of expression e_2 for variable x in e_1. We describe substitution in more detail below in subsubsection 3.2.2.1.

The remaining rules in Figure 3.4 describe how to find the next operation to perform (i.e., the global reduction rules). They specify that all expressions are evaluated left-to-right.

3.2.2.1. *Substitution.* The term *capture-avoiding subsitution* means that we get the expression that is like e_1, but we have replaced all instances of variable x with e_2 while carefully respecting static scoping (cf., subsection 1.2.4). There are two thorny issues that arise.

Shadowing: The substitution

$$\underbrace{\texttt{(}\textbf{const}\ \texttt{a = 1; a + b)}}_{e_1}\texttt{[}\underbrace{\texttt{2}}_{e_2}\texttt{/}\underbrace{\texttt{a}}_{x}\texttt{]}$$

should yield (**const** `a = 1; a + b`). That is, only *free* instances of `a` in e_1 should be replaced.

Free Variable Capture: The substutition

$$\underbrace{\texttt{(}\textbf{const}\ \texttt{a = 1; a + b)}}_{e_1}\texttt{[}\underbrace{\texttt{(a + 2)}}_{e_2}\texttt{/}\underbrace{\texttt{b}}_{x}\texttt{]}$$

should yield something like (**const** `c = 1; c + (a + 2)`). In particular, the following result is wrong:

(**const** `a = 1; a + (a + 2)`)

because the free variable `a` in e_2 gets "captured" by the **const** binding of `a`.

In both cases, the issues could be resolved by renaming all *bound* variables in e_1 so that there are no name conflicts with free variables in e_2 or x. In other words, it is clear what to do if e_1 were instead

const `c = 1; c + b`

in which case textual substitution would suffice.

The observation is that renaming *bound* variables should preserve the meaning of the expression, that is, the following two expressions are somehow equivalent:

(**const** `a = 1; a`) $\equiv_\alpha$ (**const** `b = 1; b`)

For historical reasons, this equivalence is known α-equivalence, and the process of renaming bound variables is called α-renaming. This observation also leads to coming up with an abstract syntax representation so that the above two expressions are represented with the same object. As an aside, one way to do this is to use variables in the meta language to represent variables in the object langauge. This idea is known as *higher-order abstract syntax.*

In DoConst, our situation is slight more restricted than the general case discussed above. In particular, the substitution is of the form $e[v/x]$

$$\boxed{e[e'/x] = e''}$$

$$\begin{array}{rcll}
x_1[e'/x] & \stackrel{\text{def}}{=} & e' & \text{if } x = x_1 \\
x_1[e'/x] & \stackrel{\text{def}}{=} & x_1 & \text{if } x \neq x_1 \\
(\textbf{const}\ x_1 = e_1; e_2)[e'/x] & \stackrel{\text{def}}{=} & \textbf{const}\ x_1 = (e_1[e'/x]); e_2 & \text{if } x = x_1 \\
(\textbf{const}\ x_1 = e_1; e_2)[e'/x] & \stackrel{\text{def}}{=} & \textbf{const}\ x_1 = (e_1[e'/x]); (e_2[e'/x]) & \text{if } x \neq x_1 \\
n[e'/x] & \stackrel{\text{def}}{=} & n & \\
b[e'/x] & \stackrel{\text{def}}{=} & b & \\
\textbf{undefined}[e'/x] & \stackrel{\text{def}}{=} & \textbf{undefined} & \\
(uop\, e_1)[e'/x] & \stackrel{\text{def}}{=} & uop\,(e_1[e'/x]) & \\
(e_1\ bop\ e_2)[e'/x] & \stackrel{\text{def}}{=} & (e_1[e'/x])\ bop\ (e_2[e'/x]) & \\
(e_1\ ?\ e_2 : e_3)[e'/x] & \stackrel{\text{def}}{=} & (e_1[e'/x])\ ?\ (e_2[e'/x]) : (e_3[e'/x]) & \\
(\textbf{console.log}(e_1))[e'/x] & \stackrel{\text{def}}{=} & \textbf{console.log}(e_1[e'/x]) &
\end{array}$$

FIGURE 3.5. Defining substitution $e[e'/x]$ assuming e and e' use disjoint sets of bound variables

where the replacement for x has to be value. Values have no free variables, so only the shadowing issue arises.

In Figure 3.5, we define substitution $e[e'/x]$ by induction over the structure of expression e. As a pre-condition, we assume that e and e' use disjoint sets of bound variables. This pre-condition can always be satisfied by renaming bound variables appropriate as described above. Or if we require that e' has to be value, then this pre-condition is trivally satisfied. The most interesting cases are for variable uses and **const** bindings. For variable uses, we yield e' if the variable matches the variable being substituted for; otherwise, we leave the variable use unchanged. For a binding **const** $x_1 = e_1; e_2$, we recall that the scope of x_1 is e_2, so we substitute in e_2 depending on whether x_1 is x. The remaining expression forms simply "pass through" the substitution.

3.2.2.2. *Multi-Step Evaluation.* We have now defined how to take one-step of evaluation. The multi-step evaluation judgment

$$e \longrightarrow^* e'$$

says, "Expression e can step to expression e' in zero-or-more steps." This judgment is defined using the following two rules:

$$\boxed{e \longrightarrow^* e'}$$

$$\frac{}{e \longrightarrow^* e}\ \textsc{ZeroSteps} \qquad \frac{e \longrightarrow e' \qquad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ \textsc{AtLeastOneStep}$$

In other words, $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.

A property that we want is that our big-step semantics and our small-step semantics are "the same." We can state this property formally as follows.

PROPERTY 3.2 (Big-Step and Small-Step Equivalence). $\cdot \vdash e \Downarrow v$ *if and only if* $e \longrightarrow^* v$.

CHAPTER 4

Static Checking

4.1. Type Checking

In section 3.2, we defined a one-step reduction relation such that for any closed expression e: either e is a value or $e \longrightarrow e'$ for some e', that is, e can take a step to e'. This property is very nice but it came at a cost in complexity: we defined conversions between all types of values.

However, with a complex enough language, some types of values simply do not have sensible conversions. For example, let us consider extending JAVASCRIPTY with function values. How should the number 3 convert to a function value?

In Figure 4.1, we extend JAVASCRIPTY with first-class functions. The language of expressions are extended with function expressions

$$(x) \Rightarrow e_1\,,$$

which we consider shorthand for the following concrete syntax:

$$\textbf{function}\ (x)\ \{\ \textbf{return}\ e_1\ \}$$

in JavaScript. For simplicity, we restrict functions to be anonymous and with exactly one argument. Function calls are written as $e_1(e_2)$. The language of values are also extend with function expressions, that is, function expressions are themselves considered values.

4.1.1. Getting Stuck. In Figure 4.2, we give additional rules for evaluating function calls. Observe in the DOCALL rule that an evaluation step only makes sense if we are calling a function value. Otherwise, the set of rules simply say that call expressions are evaluated left-to-right and that both the function and the argument expressions must be values before continuing to evaluating with the body of the function. This latter choice is known as call-by-value semantics; we will return to this notion in **??**.

expressions	$e ::= \cdots \mid (x) \Rightarrow e_1 \mid e_1(e_2)$
values	$v ::= \cdots \mid (x) \Rightarrow e_1$

FIGURE 4.1. Syntax of JAVASCRIPTY with first-class functions.

$$\textsc{DoCall}\ \frac{}{((x) => e_1)(v_2) \longrightarrow e_1[v_2/x]} \qquad \textsc{SearchCall}_1\ \frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)}$$

$$\textsc{SearchCall}_2\ \frac{e_2 \longrightarrow e_2'}{\big((x) => e_1\big)(e_2) \longrightarrow \big((x) => e_1\big)(e_2')}$$

FIGURE 4.2. Small-step operational semantics of JAVASCRIPTY with first-class functions (extends Figure 3.4).

$$\textsc{TypeErrorCall}\ \frac{v_1 \neq p(x) => e_1}{v_1(e_2) \longrightarrow \mathsf{typeerror}} \qquad \textsc{PropagateCall}_1\ \frac{}{\mathsf{typeerror}(e_2) \longrightarrow \mathsf{typeerror}}$$

$$\textsc{PropagateCall}_2\ \frac{}{v_1(\mathsf{typeerror}) \longrightarrow \mathsf{typeerror}}$$

Other PROPAGATE rules for other expression forms not shown.

FIGURE 4.3. Extending the small-step semantics of JAVASCRIPTY from Figure 4.2 with dynamic type errors.

Note that these rules do not say anything about how to evaluate an ill-typed expression, such as

```
3(4).
```

Intuitively, evaluating this expression should result in an error. We do not state this error explicitly. Rather, we see that this an expression that is (1) not value and (2) can make no further progress (i.e., there's no rule that specifies a next expression). We call such an expression a *stuck expression*, which captures the idea that it is erroneous in some way.

4.1.2. Dynamic Typing. Another formalization and implementation choice would be to make such ill-typed expressions step to an error token. For example, we add to the expression language a token typeerror representing a dynamic type error:

$$\text{expressions} \quad e ::= \cdots \mid \mathsf{typeerror}$$

An ill-typed function call now steps to typeerror with rule TYPEERRORCALL. We also need to extend rules for evaluating other all other expression

expressions	$e ::= \cdots \mid (x : \tau) \Rightarrow e_1$
values	$v ::= \cdots \mid (x : \tau) \Rightarrow e_1$
types	$\tau ::= \textbf{number} \mid \textbf{bool} \mid \textbf{Undefined} \mid (x : \tau) \Rightarrow \tau'$
type environments	$\Gamma ::= \cdot \mid \Gamma[x \mapsto \tau]$

FIGURE 4.4. Restricting JAVASCRIPTY with static typing.

forms that propagate the typeerror token if one is encountered in searching for a redex. We show PROPAGATECALL$_1$ and PROPAGATECALL$_2$, which are two such rules, that correspond to SEARCHCALL$_1$ and SEARCHCALL$_2$, respectively.

With this instrumentation, we distinguish a dynamic type error for any other reason for getting stuck. For example, an expression with free variables, such as

```
x.
```

Recall that our one-step evaluation relation is intended for closed expressions, so we might view this an internal error of the interpreter implementation rather than an error in the input JAVASCRIPTY program.

4.1.3. Static Typing. In subsection 4.1.1, we saw how "bad" expressions, such as,

```
3(4)
```

are erroneous according to our operational semantics in that they "get stuck." This expression gets stuck because a call expression $e_1(e_2)$ only applicable to function values. We say that such an expression `3(4)` is *ill-typed* or *not well-typed*.

A *type* is a classification of values that characterize the valid operations for these values. A *type system* consists of a language of types and a typing judgment that defines when an expression has a particular type. When we say that an expression e has a type τ, we mean that if e evaluates to a value, then that value should be of type τ. In this way, a type system predicts some property about how an expression evaluates at run-time.

In Figure 4.4, we show a language of types τ for JAVASCRIPTY that includes base types for numbers n : **number**, Booleans b : **bool**, and the undefined value **undefined** : **Undefined**, as well as a constructed type for function values. A function type $(x : \tau) \Rightarrow \tau'$ classifies function values whose return value has type τ' assuming its called with an argument of type τ. Our expression language e has modified slightly to add type annotations to function parameters.

A typing judgment form is defined by a set of *typing rules* that is the first step towards defining a *type checking* algorithm. A *type error* is an

expression that violates the prescribed typing rules (i.e., may produce a value outside the set of values that it is supposed to have). We define a typing judgment form inductively on the syntactic structure of program objects (e.g., expressions).

Recall from our earlier discussion on binding (subsection 1.2.3) that the type of an expression with free variable depends on an environment. In other words, consider the expression

```
x + 1.
```

Is this expression well-typed? It depends. If in the environment, `x` is stated to type `Int`, then it is well-typed; otherwise, it is not. We see that the type of an expression e depends on a *type environment* Γ that gives the types of the free variables of e. Thus, our typing judgment form is as follows:

$$\Gamma \vdash e : \tau$$

that says informally, "In typing environment Γ, expression e has type τ." Observe how similar this judgment form is to our big-step evaluation judgment form from subsection 3.1.2. This observation is a bit more than a coincidence. A standard type checker works by inferring the type of an expression by recursively inferring the type of each sub-expression. A big-step interpreter computes the value of an expression by recursively computing the value of each sub-expression. In essence, we can view a type checker as an *abstract evaluator* over a *type abstraction* of *concrete values*.

In Figure 4.5, we define typing of JAVASCRIPTY. The first four rules TYPENUMBER, TYPEBOOL, TYPEUNDEFINED, and TYPEFUNCTION describe the types of values. The types of the primitive values n, b, and **undefined** are as expected. The TYPEFUNCTION is more interesting:

$$\textsc{TypeFunction}\ \frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash\ (x:\tau) \Rightarrow e\ :\ (x:\tau) \Rightarrow \tau'}$$

A function value has a function type $(x:\tau) \Rightarrow \tau'$ (also sometimes called simply an "arrow" type) whose parameter type is τ and return type is τ'. The return type τ' is obtained by inferring the type of the body expression e under the extended environment $\Gamma[x \mapsto \tau]$.

Examining the typing rules for the unary operators, we see that we have decided to restrict the input programs beyond those that get stuck with the small-step semantics defined in Figure 4.2:

$$\textsc{TypeNeg}\ \frac{\Gamma \vdash e_1 : \textbf{number}}{\Gamma \vdash -e_1 : \textbf{number}} \qquad \textsc{TypeNot}\ \frac{\Gamma \vdash e_1 : \textbf{bool}}{\Gamma \vdash\ !e_1 : \textbf{bool}}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{TypeNumber}\ \frac{}{\Gamma \vdash n : \textbf{number}} \qquad \textsc{TypeBool}\ \frac{}{\Gamma \vdash b : \textbf{bool}} \qquad \textsc{TypeUndefined}\ \frac{}{\Gamma \vdash \textbf{undefined} : \textbf{Undefined}}$$

$$\textsc{TypeFunction}\ \frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (x:\tau) => e : (x:\tau) \Rightarrow \tau'} \qquad \textsc{TypeVar}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{TypeNeg}\ \frac{\Gamma \vdash e_1 : \textbf{number}}{\Gamma \vdash -e_1 : \textbf{number}}$$

$$\textsc{TypeNot}\ \frac{\Gamma \vdash e_1 : \textbf{bool}}{\Gamma \vdash !e_1 : \textbf{bool}} \qquad \textsc{TypeSeq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 , e_2 : \tau_2}$$

$$\textsc{TypeArith}\ \frac{\Gamma \vdash e_1 : \textbf{number} \qquad \Gamma \vdash e_2 : \textbf{number} \qquad bop \in \{+, -, *, /\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{number}}$$

$$\textsc{TypeInequality}\ \frac{\Gamma \vdash e_1 : \textbf{number} \qquad \Gamma \vdash e_2 : \textbf{number} \qquad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypeEquality}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \text{ has no function types} \qquad bop \in \{===, !==\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypeAndOr}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool} \qquad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}}$$

$$\textsc{TypePrint}\ \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \textbf{console.log}(e_1) : \textbf{Undefined}} \qquad \textsc{TypeIf}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : \tau}$$

$$\textsc{TypeConst}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{const}\ x = e_1; e_2 : \tau_2} \qquad \textsc{TypeCall}\ \frac{\Gamma \vdash e_1 : (x:\tau) \Rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(e_2) : \tau'}$$

FIGURE 4.5. Typing of JAVASCRIPTY.

For example, with rule TYPENEG, we say that $-e_1$ is well-typed if e_1 has type **number**. Furthermore, this rule is the only rule for $-e_1$, so we are only permitting unary negation $-$ of numbers. Similarly, we are only permit not ! applied to booleans. Thus, only code that does not need conversions is well typed. Therefore, we can simplify our interpreter to get rid

of conversions if only allow executing well-typed programs. We continue with this design choice in this set of rules.

In the sequencing rule TYPESEQ

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \, , e_2 : \tau_2} \text{TYPESEQ},$$

the type of the sequencing expression is τ_2. The type of e_1 is checked but then dropped—all we care about is that e_1 is well typed. Even though we are dropping the type, the expression still needs to be type checked because e_1 should evaluate to a value (without getting stuck) before being dropped and continuing with the evaluation e_2.

The rule for conditionals TYPEIF

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 \, ? \, e_2 : e_3 : \tau} \text{TYPEIF}$$

looks a bit different than DOIFTRUE and DOIFFALSE, the corresponding big-step evaluation rules. In evaluation, we evaluate the guard expression e_1 and continue with either e_2 or e_3 depending on whether e_1 evaluates to **true** or **false**. In type checking, we are *predicting* the type of values that arise during execution before executing the program. This phase is known as *static-time* or *compile-time* as opposed to *dynamic-time* or *run-time* during execution.

In a sufficiently complex language (i.e., a Turing-complete language), it is undecidable to precisely determine the value of an expression before executing it (i.e., *statically*), so we must approximate. Here, we require that both branches e_2 and e_3 have the same type τ because we do not know whether e_1 will be **true** or **false** at run-time. This over-approximation throws out some programs that would not get stuck at run-time as a trade-off for being able to guarantee that well-typed programs do not get stuck. Over-approximation also happens in TYPECALL.

This over-approximation requirement is the fundamental trade-off between *static* and *dynamic* typing.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[2] C. A. R. Hoare. Null references: The billion dollar mistake. In *QCon London*, 2009. URL `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`. Presentation.

[3] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008. ISBN 9780981531601. URL `http://books.google.com/books?id=MFjNhTjeQKkC`.