Principles and Practice of Programming Languages

Bor-Yuh Evan Chang

2024-09-08

Table of contents

Preface

Disclaimer: This manuscript is a draft of a set of course notes for CSCI 3155 Principles of Programming Languages at the University of Colorado Boulder. There may be typos, bugs, or inconsistencies that have yet to be resolved.

This version is work-in-progress update and revision from a previous LaTeX version.

The definitive current version is at https://csci3155.cs.colorado.edu/pppl-course/book/.

Part I

Introduction

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.

1.1 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 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." [6]

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."

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

Scala "Scala is a blend of object-oriented and functional programming concepts in a statically typed language" [4].

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

1.2 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.

1.3 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 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.

1.4 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].

1.5 You will be able to use AI assistants to accelerate your creative design.

Generative AI for programming is here to stay. AI is amazing accelerator for creative design by drawing on what has been done before—if you can evaluate what it gives you.

Understanding how programs and programming languages are composed enables to effectively understand what makes sense and what needs refinement to create something new.

2 Course Approach

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 interpreters for mini-versions of JavaScript—our example *source language*. The source language is what called the *object language* (i.e., the under under study). We will see that interpreters are the basis for realizing computation, and we will study the programming language theory that enable us to reason carefully about a language's design and implementation. Our approach will be gradual in that we will initially consider a small subset of JavaScript and then slowly grow the aspects of the language that we consider (and see how they underlie many other programming languages).

Our *implementation language* of study will be Scala. The implementation language is the *meta language* (i.e., the language used to study another). 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 [4] and in many ways tries to support each in its "native environment." Scala has also found a myriad of other applications, including being a important language for building data-processing pipelines. It is compatible with Java and runs on the Java Virtual Machine (JVM) and has been applied in industrial practice by such companies as LinkedIn.

Incoming students often expect this course to be what I will call a trip to the Zoo of Programming Languages. While it is certainly interesting to go to the zoo, we seek a more informative and scientific study of the underlying principles. A more apt analogy is an *anatomy* course where we will study the "guts" and inner-workings of programming languages. After this course, such an anatomical study will enable us to compare and contrast programming languages in a substantive manner and address the learning goals outlined above in Chapter 1.

2.1 Expectations and Finding Success

The study of programming languages gets at the core of computation and introduces abstract concepts. Past experience suggests that the study of programming languages can lead to *unnecessary* panic and anxiety. At times, it may appear like a lot of effort and complexity to study a toy language and to define and describe seemingly simple language features. In the end, this “dissection”-based approach pays off in distilling computation into simple, composable concepts that enable you to see how they appear over-and-over again to realize modern software.

This course is an active learning course, which means the learning is driven primarily by active discovery in doing the assignments. To succeed in the course, we suggest to the student to keep the following in mind:

Principles Exist While everyday programming languages seem complex, underlying principles exist. They take work to uncover and see, but they can be understood. Knowing the underlying principles are there, you should not panic and always seek help from course guides.

Practioners Exist Programming languages come alive from the people that use them to create amazing software. Everyday languages, like Scala, have a community and are well documented. You should join the community and get used to reading documentation.

Learn by Doing Concepts may look simple when the course guides walk you through them. However, until you dive deep and get your hands dirty on code — run it, modify it, write it, play with it, talk about it in your own words, you will not *own* the knowledge. You will make mistakes and get confused along the way, but with hard work and help from your course guides, you will truly master the concepts.

Part II

Programming Preliminaries

3 Expressions

3.1 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*. A 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*). To a first approximation, there is no external memory. Expression rewriting is actually not so unfamiliar. Primary school arithmetic is expression evaluation:

$$\begin{array}{rcl}(1+1)+(1+1) & \longrightarrow & 2+(1+1) \\ & \longrightarrow & 2+2 \\ & \longrightarrow & 4\end{array}$$

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

In actuality, the “schism” is somewhat 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 advantages by being independent of how a machine evaluates expressions (i.e., called *referential transparency*). 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, or whether they are done in parallel), which makes it easier to reason about programs in isolation (e.g., the meaning of $(1+1)+(1+1)$) and for compilers to optimize your code. At the same time, interacting with the underlying execution engine can be powerful, and thus we at times want effects. The potentially surprising idea at this point is how often we can program effectively without effects.

We 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, 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 *first-class functions*(discussed in).

Many languages, including JavaScript and Scala, have aspects of both, including the features that are often considered the most characteristic: *mutation* and *first-class functions*.

3.2 Basic Values, Types, and Expressions

We begin our language study by focusing on a small subset of Scala. Our intent is not to give an exhaustive tutorial or manual on Scala, but rather to use Scala as an example language to highlight concepts that underly many other programming languages. For a tutorial on Scala, see, for example, Programming in Scala [5].

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`, `Long`, `Float`, `Double`, `Boolean`, `Char`, and `String`.

We can directly write down values of these types using *literals*:

```
42
42L
1.618f
1.618
true
'a'
"Hello!"
```

```
res0_0: Int = 42
res0_1: Long = 42L
res0_2: Float = 1.618F
res0_3: Double = 1.618
res0_4: Boolean = true
res0_5: Char = 'a'
res0_6: String = "Hello!"
```

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

```
44 - 2
!true
true && false
1 < 2
if (1 < 2) 3 else 4
"Hello" + "!"
```

```
res1_0: Int = 42
res1_1: Boolean = false
res1_2: Boolean = false
res1_3: Boolean = true
res1_4: Int = 3
res1_5: String = "Hello!"
```

In the above, we see Scala infers the type of each expression and evaluates each expression to its value. A *value* is an expression cannot be evaluated any further — the result of evaluating an expression.

3.2.1 Static Type Checking

We can check that an expression has the expected type as follows:

```
42: Int
42L: Long
1.618f: Float
1.618: Double
true: Boolean
'a': Char
"Hello!": String
44 - 2: Int
!true: Boolean
true && false: Boolean
1 < 2: Boolean
if (1 < 2) 3 else 4: Int
"Hello" + "!": String
```

```
res2_0: Int = 42
res2_1: Long = 42L
```

```
res2_2: Float = 1.618F
res2_3: Double = 1.618
res2_4: Boolean = true
res2_5: Char = 'a'
res2_6: String = "Hello!"
res2_7: Int = 42
res2_8: Boolean = false
res2_9: Boolean = false
res2_10: Boolean = true
res2_11: Int = 3
res2_12: String = "Hello!"
```

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
42: Boolean
```

```
cmd3.sc:1: type mismatch;
 found   : Int(42)
 required: Boolean
val res3 = 42: Boolean
           ^Compilation Failed


:
Compilation Failed
```

The typing assertion is also an expression, so we can annotate sub-expressions to check that they have the expected type. Doing so becomes useful for debugging when an expression e becomes complicated.

```scala
(44: Int) - 2
```

```
res3: Int = 42
```

Scala is *statically typed*. In essence, this statement means that the Scala compiler will perform some validation at *compile-time* (called *type checking*) and only translate well-typed expressions.

```
true - 2
```

```
cmd4.sc:1: value - is not a member of Boolean
val res4 = true - 2
                ^Compilation Failed


:
Compilation Failed
```

We discuss type checking in later chaptersfurther in; for now, it suffices to view type checking as making sure all operations in all sub-expressions have the "expected type."

```
(1 + 2) + (3 + 4): Int
- if 1 + 2: Int
  - if 1: Int
  - if 2: Int
- if 3 + 4: Int
  - if 3: Int
  - if 4: Int
```

Here, we check explicitly that each sub-expression has the expected type:

```
((1: Int) + (2: Int): Int) + ((3: Int) + (4: Int): Int)
```

```
res4: Int = 10
```

We state that an expression e is well-typed with type τ using essentially the same notation as Scala, that is, we write

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

3.2.2 Run-Time Errors

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

```
42 / 0
```

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.

We can also test that an expression has the expected value at run-time with an `assert` expression:

```
assert(44 - 2 == 42)
assert(1 < 2 == true)
assert((if (1 < 2) 3 else 4) == 3)
assert("Hello" + "!" == "Hello!")
```

Nothing is printed in the above because all of the `assert`ions pass (i.e., all of the given expressions to `assert` evaluate to `true`).

```
assert(44 - 2 == 0)
```

The above is now a run-time error.

3.2.3 Unit

Unlike some other common languages (e.g., JavaScript, C, Java), Scala does not distinguish between expressions and statements. Instead, constructs we might consider as "effectful statements" are expressions that have type `Unit`.

```
assert(44 - 2 == 42): Unit
println("Hello!"): Unit
```

```
Hello!
```

The `Unit` type has one single value "`()`" (usually called the "unit" value).

```
(): Unit
```

Since the unit value `()` itself is uninteresting and usually associated with side-effecting expression, the printer in the above simply chooses to not print unit values.

3.2.4 Operators

Scala has the all of the usual operators on numeric, `Boolean`, and `String` data types. For the numeric types, Scala will perform conversions implicitly using methods like `toInt`, `toLong`, and `toString`.

```
3L + 4
3 + 4L
(3L + 4L).toInt
3.toString
```

```
res10_0: Long = 7L
res10_1: Long = 7L
res10_2: Int = 7
res10_3: String = "3"
```

An interesting aspect of Scala is that all operators are actually methods.

```
3 + 4
3.+(4)
"Hello".endsWith("lo")
"Hello" endsWith "lo"
```

```
res11_0: Int = 7
res11_1: Int = 7
res11_2: Boolean = true
res11_3: Boolean = true
```

Binary operators like `3 + 4` is just shorthand for a method call `3.+(4)`. That is, these two *syntactically* different expressions have the same *semantics*. The term "syntactic sugar" is sometimes used in this case (e.g., `3 + 4` is syntactic sugar for `3.+(4)`). Note in the above that this works for any binary method, not just ones using symbols (e.g., `endWith` as above).

3.3 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

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

For example,

$$(1 + 2) + (3 + 4) \longrightarrow 3 + (3 + 4)$$

is the only case assuming left-to-right evaluation order.

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 example, all of the following hold:

$$\begin{array}{lll}
(1+2)+(3+4) & \longrightarrow^* & (1+2)+(3+4) \\
(1+2)+(3+4) & \longrightarrow^* & 3+(3+4) \\
(1+2)+(3+4) & \longrightarrow^* & 3+7 \\
(1+2)+(3+4) & \longrightarrow^* & 10
\end{array}$$

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 laterin **?@sec-operational-semantics** and **?@sec-procedural-abstraction**. Eager evaluation means that sub-expressions 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

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

When we write this particular expression e

```
3 + 4
```

```
res12: Int = 7
```

we are asking the interpreter $\Downarrow$ for its value v, which in this case is `7`. That is, we say that for Scala, the following evaluation relation holds:

$$3 + 4 \Downarrow 7 \;.$$

We can see that Scala evaluates `+` left-to-right by adding side-effecting expressions, such as printing to console:

```
{ println("eval((1 + 2) + (3 + 4))");
  { println("- if eval(1 + 2)");
    { println("  - if eval(1)"); 1 } +
    { println("  - if eval(2)"); 2 }
  } +
  { println("- eval(3 + 4)");
    { println("  - if eval(3)"); 3 } +
    { println("  - if eval(4)"); 4 } } }
```

```
eval((1 + 2) + (3 + 4))
- if eval(1 + 2)
  - if eval(1)
  - if eval(2)
- eval(3 + 4)
  - if eval(3)
  - if eval(4)
```

```
res13_1: Int = 10
```

Here, we add a `println` printing expression to each sub-expression (using the indention format when we considered static type checking above in Section 3.2.1).

4 Binding and Scope

4.1 Binding Names

4.1.1 Value Bindings

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:

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

```
two: Int = 2
four: Int = 4
```

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:

$$\texttt{val}\ x\texttt{:}\ \tau\ \texttt{=}\ e$$

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 $: \tau$ 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:

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

For example, the first binding in our example

```
val two = 2
```

```
two: Int = 2
```

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`

```
two + two
```

```
res2: Int = 4
```

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`.

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

```
two: Int = 2
four: Int = 4
```

We can see that what is printed above are the *value* and *type* environments:

$$[\texttt{two} \mapsto 2, \texttt{four} \mapsto 4] \qquad [\texttt{two} \mapsto \texttt{Int}, \texttt{four} \mapsto \texttt{Int}]\ .$$

Sometimes, we use notation for the mappings that suggest value or type environments, respectively:

$$[\texttt{two} \Downarrow 2, \texttt{four} \Downarrow 4] \qquad [\texttt{two} : \texttt{Int}, \texttt{four} : \texttt{Int}]\ .$$

4.1.2 Type Bindings

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

```scala
type Str = String
"Hello": Str
"Hello"
```

```
defined type Str
res4_1: Str = "Hello"
res4_2: String = "Hello"
```

Type binding is not so useful in our current Scala subset, but such bindings become particularly relevant later onin **?@sec-something**.

4.2 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 variable by using blocks `{...}`:

```scala
{
  { val three = 3 }
  three
}
```

```
cmd5.sc:2: not found: value three
  val res5_1 = three
               ^Compilation Failed


:
Compilation Failed
```

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

```
a: Int = 1
b: Int = 2
c: Int = 6
```

Figure 4.1: Nested scopes and shadowing.

4.2.1 Shadowing

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

In the above Figure 4.1, there are two scopes, and there is a binding to a name `a` in each. 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 refers to the binding of `b` in the outer scope, as `b` is not bound in the inner scope. The name `c` ends up being bound to the value `6`. In particular, after applying the environments, we end up evaluating the expression `{ 3 + 2 } + 1`. Note that value binding is *not* assignment. After the inner binding of name `a` on line 4, the outer binding of `a` still exists but is hidden—called *shadowed*—within the inner scope.

We can rename the two variables named `a` in the expression in Figure 4.1 to a semantically equivalent expression that eliminates the shadowing:

```
val a_outer = 1
val b = 2
val c = {
  val a_inner = 3
  a_inner + b
} + a_outer
```

```
a_outer: Int = 1
b: Int = 2
c: Int = 6
```

Figure 4.2: Renaming to eliminate shadowing.

Observe that in Figure 4.2, the name `a_inner` is regardless unavailable in the outer, global scope.

Also observe that in Scala, blocks `{...}` are also expressions—like `(...)` expressions that introduce a new scope:

```
val a = { 3 + 2 } + 1
val b = ( 3 + 2 ) + 1
```

```
a: Int = 6
b: Int = 6
```

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. For Scala, the rule is that for any use of a variable x, the innermost scope that (a) contains the use of x and (b) has a binding for x is the one that applies. Note that there are only two scopes in the above example in Figure 4.1 (e.g., not one for each declaration). Thus, the following example has a compile-time error:

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

```
cmd8.sc:3: forward reference extends over definition of value c
  val c = a
          ^Compilation Failed

:
Compilation Failed
```

In particular, the use of variable `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 in Figure 4.1. How do we describe the evaluation of this expression? The substitution-based evaluation rule for names described previously in Section 4.1 needs to be more nuanced. In particular, eliminating the binding of the name in the outer scope should replace the use of name `a` on line 6 but not the use of name `a` on 5. In particular, applying the environment $[\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2]$ to lines 3 to 4 yields the following:

```
val c = {
  val a = 3
  a + 2
} + 1
```

```
c: Int = 6
```

4.2.2 Free versus Bound Variables

This notion of substitution is directly linked to the 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 `{ val x = 3; x + y }`, variable `x` is *bound*, while variable `y` is *free*:

```
{
  { val x = 3; x + y }
}
```

```
cmd9.sc:1: not found: value y
val res9 = { val x = 3; x + y }
                            ^Compilation Failed


:
Compilation Failed
```

Note that as an aside about Scala syntax, the ; sequences expressions, which are inferred when using newlines:

```
{
  { val x = 3
    x + y }
}
```

```
cmd9.sc:2: not found: value y
    x + y }
        ^Compilation Failed


:
Compilation Failed
```

We can see *free variables* as inputs to an expression: we do not know how to evaluate it without an environment giving bindings.

Consider again the example of renaming to eliminate shadowing in Figure 4.2. A key property to observe is that when looking at a sub-expression, we can rename the bound variables consistently to a semantically equivalent one, but we cannot rename the free variables.

A *closed* expression is one that has no free variables, which is then one that can be evaluated with an empty environment:

```
{
  { val y = 4; { val x = 3; x + y } }
}
```

```
res9: Int = 7
```

An *open* expression is one that has at least one free variable (and thus cannot be evaluated with an empty environment).

4.3 Mutable Variables

In the above, we are using the term *variable* in the same sense as in mathematical logic where variables are placeholder names and *immutable*. However, in the context of imperative programming, the notion of a program variable is often, by default, considered a name for a *mutable* memory cell.

Just like many other languages (including JavaScript, Java, C), Scala has both immutable and mutable variables.

Language	Immutable Variable Declaration	Mutable Variable Declaration
Scala	`val one = 1`	`var one = 1`
JavaScript	`const one = 1`	`var one = 1`
Java	`final int one = 1`	`int one = 1`
C	`const int one = 1`	`int one = 1`

A mutable variable allocates a memory cell that can be assigned:

```scala
var greeting = "Hello"
println(greeting)
greeting = "Hi"
println(greeting)
greeting = "Hola"
println(greeting)
```

```
Hello
Hi
Hola
```

```
greeting: String = "Hola"
```

where the value stored in the cell depends on when it is read.

There are many reasons why we might prefer `val`. One reason is understandability of the source code. As we see from the above, using `var` breaks referential transparency of variable use (e.g., the value that the expression `greeting` evaluates to depends on when it runs).

Another reason is getting more efficient executable code from the compiler. A mutable requires a new memory allocation for each time it executes, whereas a reference to an immutable can be shared aggressively.

4.4 Functions and Tuples

4.4.1 Function Definitions

The most basic and perhaps most important form of abstraction in programming languages is defining functions. Here's an example Scala function:

```scala
def square(x: Int): Int = x * x
```

```
defined function square
```

where `x` is a *formal parameter* of type for the function that returns a value of type `Int`. Schematically, a function definition has the following form:

$$\texttt{def}\ x(x_1\texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}\ \tau_n\texttt{)}\texttt{:}\ \tau\ \texttt{=}\ e$$

where the formal parameter types $\tau_1, \ldots, \tau_n$ are always required and the return type τ is sometimes required.[1] 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 `def`. For example, the following code is valid:

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

```
defined function max
```

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

4.4.2 First-Class Functions

Functions are values in Scala. That is, expressions can evaluate to values that are functions. Functions are values is sometimes stated as, "Functions are first-class in Scala."

A function literal defines an anonymous function:

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

```
res13: Int => Int = ammonite.$sess.cmd13$Helper$$Lambda$1997/0x0000000800acf040@1f40f380
```

We can see that a function taking a formal parameter of type `Int` and returning a value of type `Int` is written `Int => Int`.

For historical reasons referencing the Lambda Calculus, function values are sometimes called *lambdas*.

As values, we can bind functions to variables:

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

[1]Scala is also an object-oriented language, and this syntax actually introduces a *method*. Fortunately, in most situations, methods can be treated like functions in Scala.

```
square: Int => Int = ammonite.$sess.cmd14$Helper$$Lambda$2006/0x0000000800ad4840@69cd8b89
```

If the type of the formal parameters are clear from context, they can be inferred:

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

```
square: Int => Int = ammonite.$sess.cmd15$Helper$$Lambda$2010/0x0000000800ad7040@46593b21
```

Note that it is a common style to put braces `{}` around function literals to make them stand out more visually, even if there's no need to introduces bindings:

```scala
val square: Int => Int = { x => x * x }
```

```
square: Int => Int = ammonite.$sess.cmd16$Helper$$Lambda$2014/0x0000000800ad9840@11ee4343
```

An expression defining a function can refer to variables bound in an outer scope:

```scala
val four = 4
def plusFour(x: Int): Int = x + four
plusFour(4)
```

```
four: Int = 4
defined function plusFour
res17_2: Int = 8
```

Referencing a detail needed to properly implement static scoping with such functions that can refer to variables bound in an outer scope, first-class functions are also sometimes called *closures*.

NB The `return` keyword does exist in Scala but is rarely used and generally considered bad practice. For the purposes of this course, we should also avoid using `return`, as it can lead to unexpected results without a deeper knowledge about Scala internals.

4.4.3 Tuples

A *tuple* is a simple data structure that combines a fixed number of values. It is a value that is a pair, triple, quadruple, etc. of values:

```
val oneT = (1, true)
```

```
oneT: (Int, Boolean) = (1, true)
```

That is, a pair is a 2-tuple, a triple is a 3-tuple, a quadruple is a 4-tuple and so forth.

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

$$(e_1\texttt{,}\ \dots\ \texttt{,}\ e_n)\texttt{:}\ (\tau_1\texttt{,}\ \dots\ \texttt{,}\ \tau_n)\ .$$

4.4.3.1 Functions Returning Multiple Associated Values

Tuples are often used with functions to return multiple values or to pass around a small number of associated values together. It is generally used when defining a custom data type for a single use does not really make sense, and it is generally advisable not to use tuples larger than 4- or 5-tuples.

As example of a function returning multiple associated values, we can write a function that takes two integers and and returns a pair of their quotient and their remainder:

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

```
defined function divRem
```

4.4.3.2 Deconstructing Tuples with Pattern Matching

The i^{th} component of a tuple e can be obtained using the expression e`._`i (invoking the i^{th} projection method). For example,

```
val divRemSevenThree: (Int, Int) = divRem(7, 3)
val div: Int = divRemSevenThree._1
val rem: Int = divRemSevenThree._2
```

```
divRemSevenThree: (Int, Int) = (2, 1)
div: Int = 2
rem: Int = 1
```

However, it much more common and almost always clearer to get the components of a tuple using *pattern matching*:

```scala
val (div, rem) = divRem(7, 3)
```

```
div: Int = 2
rem: Int = 1
```

Note that the bottom line is a binding of two names `div` and `rem`, which are bound to the first and second components of the tuple returned by evaluating `divRem(7,3)`, respectively. The parentheses `()` are necessary in the code above.

If we do not need one part of the pair, we can use the `_` pattern:

```scala
val (div, _) = divRem(7, 3)
```

```
div: Int = 2
```

4.4.3.3 Side-Effecting Functions

There is no 1-tuple type, but there is a 0-tuple type that is called `Unit` (see Section 3.2.3). There is only one value of type (also typically called the unit value). The unit value is written using the expression `()` (i.e., open-close parentheses), as it is the 0-tuple.

As noted in Section 3.2.3, a good indication of imperative programming are when expressions return `Unit`. Conceptually, the unit value represents "nothing interesting returned." When we introduce side-effects, a function with return type 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) returns the unit value:

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

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

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

```
defined function 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.

4.4.4 Pattern Matching

Another workhorse of defining functions in a functional programming language is pattern matching. We have seen pattern matching to deconstruct tuples (Section 4.4.3.2), such as

```
def fst(pair: (Int, Boolean)): Int = {
  val (i, _) = pair
  i
}
fst(3, true)
```

```
defined function fst
res25_1: Int = 3
```

4.4.4.1 Nested Pattern Matching

Patterns can match deeply, which is particularly powerful.

```
val (_, (i, _)) = (3.14, (42, true))
```

```
i: Int = 42
```

4.4.4.2 Heterogenous Pattern Matching

For tuples, we can pattern match with `val` because the shape of tuples is *homogenous*. Pattern matching can also be used when there are multiple possible cases, such as

```
def isZero(n: Int): Boolean = n match {
  case 0 => true
  case _ => false
}
isZero(10)
```

```
defined function isZero
res27_1: Boolean = false
```

Observe that one possible pattern is a value: `0` in this example.

Cases are attempted from top-to-bottom, so it is important to order more specific patterns before less specific ones:

```scala
def isZero(n: Int): Boolean = n match {
  case _ => false
  case 0 => true
}
isZero(0)
```

```
defined function isZero
res28_1: Boolean = false
```

And it is possible that a run-time error results when no cases match:

```scala
def isZero(n: Int): Boolean = n match {
  case 0 => true
}
isZero(10)
:::

The `val`{.scala} pattern matching is then just a special case:

::: {.cell execution_count=36}
``` {.scala .cell-code}
def fst(pair: (Int, Boolean)): Int = pair match {
  case (i, _) => i
}
fst(3, true)
```

```
defined function fst
res30_1: Int = 3
```

4.5 String Interpolation

While this is not a core language feature, Scala has convenient string construction facilities for constructing strings that can be useful for writing debugging logs.

There are C `printf`-style format strings:

```scala
def helloWorld(greeting: String): String = "%s, World!" format greeting
helloWorld("Hello")
helloWorld("Hi")
```
```

```
defined function helloWorld
res31_1: String = "Hello, World!"
res31_2: String = "Hi, World!"
```

But even more convenient, there is a macro `s` for string interpolation:

```
def helloWorld(greeting: String): String = s"$greeting, World!"
helloWorld("Hello")
helloWorld("Hi")
```

```
defined function helloWorld
res32_1: String = "Hello, World!"
res32_2: String = "Hi, World!"
```

That is, the `$` in the string template informs Scala to evaluate the expression `greeting`. For a more complex expression than a variable use, use braces `${}`:

```
def helloWorld(greeting: String): String = s"${greeting.toLowerCase}, World!"
helloWorld("Hello")
helloWorld("Hi")
```

```
defined function helloWorld
res33_1: String = "hello, World!"
res33_2: String = "hi, World!"
```

We can update the example of inspecting how Scala evaluates from Section 3.3 with a bit more information:

```
def printeval(indent: String, e: String, v: Int): Int =
  { println(s"${indent}eval($e) = $v"); v }

printeval("", "(1 + 2) + (3 + 4)", {
  printeval("- if ", "1 + 2", {
    printeval("  - if ", "1", 1) +
    printeval("  - if ", "2", 2)
  }) +
  printeval("- if ", "3 + 4", {
    printeval("  - if ", "3", 3) +
    printeval("  - if ", "4", 4)
  })
})
```

```
  - if eval(1) = 1
  - if eval(2) = 2
- if eval(1 + 2) = 3
  - if eval(3) = 3
  - if eval(4) = 4
- if eval(3 + 4) = 7
eval((1 + 2) + (3 + 4)) = 10

defined function printeval
res34_1: Int = 10
```

To print the evaluation of the expression with its resulting value, we now print out the *post-order* traversal of the evaluation tree versus the *pre-order* traversal in Section 3.3.

Note that we need to take some care in preserving the structure of the expression `(1 + 2) + (3 + 4)` to test the evaluation order. The expression `{ val r = 3 + 4; (1 + 2) + r }` is semantically equivalent to the first one with respect to their values but not evaluation order:

```
val r = {
  printeval("- if ", "3 + 4", {
    printeval("  - if ", "3", 3) +
    printeval("  - if ", "4", 4)
  })
}
printeval("", "(1 + 2) + (3 + 4)", {
  printeval("- if ", "1 + 2", {
    printeval("  - if ", "1", 1) +
    printeval("  - if ", "2", 2)
  }) + r
})
```

```
  - if eval(3) = 3
  - if eval(4) = 4
- if eval(3 + 4) = 7
  - if eval(1) = 1
  - if eval(2) = 2
- if eval(1 + 2) = 3
eval((1 + 2) + (3 + 4)) = 10

r: Int = 7
res35_1: Int = 10
```

5 Exercise: Binding and Scope

The purpose of this exercise is to warm up on the concepts of binding and scope, by example, in Scala.

For each the following uses of variable names, give the line where that name is bound. Briefly explain your reasoning (in no more than 1–2 sentences).

5.1 Example 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
```

```
pi: Double = 3.14
defined function circumference
defined function area
```

If you are viewing this in Jupyter, you may need to enable line numbers (via `View > Show Line Numbers`).

Exercise 5.1. The use of `pi` at line 4 is bound at which line? Briefly explain.

???

Exercise 5.2. The use of `pi` at line 7 is bound at which line? Briefly explain.

???

5.2 Example 2

Consider the following Scala code:

```scala
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)
```

```
x: Int = 3
defined function f
y: Int = 3
```

Exercise 5.3. The use of `x` at line 3 is bound at which line? Briefly explain.

???

Exercise 5.4. The use of `x` at line 6 is bound at which line? Briefly explain.

???

Exercise 5.5. The use of `x` at line 10 is bound at which line? Briefly explain.

???

Exercise 5.6. The uses of `x` at line 13 is bound at which line? Briefly explain.

???

6 Data Types

6.1 Standard Collections

We have already seen one standard data type in tuples (see Section 4.4.3).

6.1.1 Lists

After tuples, the most commonly-used standard data type is probably `List`. A `List` is a sequential, functional data structure.

```
val numbers = List(1, 2, 3)
numbers.length
```

```
numbers: List[Int] = List(1, 2, 3)
res0_1: Int = 3
```

Like tuples, the `List` type constructor is parametrized by another type. In the above, we have that `numbers` is bound to a value of type `List[Int]`, that is, a list of integers. A list of strings would have type `List[String]`.

6.1.1.1 Indexing

While it is very uncommon to do so, it is possible to index into lists:

```
numbers(0)
numbers(1)
numbers(2)
```

```
res1_0: Int = 1
res1_1: Int = 2
res1_2: Int = 3
```

```scala
numbers(3)
```

Another way for getting the first element is getting the `head` of the list:

```scala
numbers.head
```

```
res3: Int = 1
```

Scalaism: As all operators in Scala are methods, the expression `numbers(0)` is syntactic sugar for a method call to `apply`:

```scala
numbers.apply(0)
```

```
res4: Int = 1
```

6.1.1.2 Nil and Cons

An empty list can also be written as `Nil`

```scala
val empty: List[Int] = List()
val nil: List[Int] = Nil
```

```
empty: List[Int] = List()
nil: List[Int] = List()
```

We can then prepend to a list with `::` as follows:

```scala
val numbers = List(1, 2, 3)
val consZero = 0 :: numbers
val consTen = 10 :: numbers
val numbersHead = numbers.head
val consZeroHead = consZero.head
val consTenHead = consTen.head
```

```
numbers: List[Int] = List(1, 2, 3)
consZero: List[Int] = List(0, 1, 2, 3)
consTen: List[Int] = List(10, 1, 2, 3)
numbersHead: Int = 1
consZeroHead: Int = 0
consTenHead: Int = 10
```

Note that there is no imperative update here. A `List` is an immutable, functional data structure. An immutable, functional data Prepending to `numbers` does not change it. Rather `consZero` and `consTen` are bound to new lists.

Recall from Section 4.3, the note about immutablility enabling efficient representations. Because `Lists` are immutable, prepending is still a constant-time operation (i.e., $O(1)$). The `consZero` and `consTen` can share the same tail (i.e., `numbers`), that is, only 5 nodes are needed in total to represent the lists `numbers`, `consZero`, and `consTen`:

```
val consZeroTail = consZero.tail
val consTenTail = consTen.tail
consZeroTail eq consTenTail
numbers eq consZeroTail
numbers eq List(1, 2, 3)
numbers == List(1, 2, 3)
```

```
consZeroTail: List[Int] = List(1, 2, 3)
consTenTail: List[Int] = List(1, 2, 3)
res7_2: Boolean = true
res7_3: Boolean = true
res7_4: Boolean = false
res7_5: Boolean = true
```

where in Scala, `eq` is the reference equality operator, while `==` is structural equality.

Note that the `List(1, 2, 3)` constructor is equivalent to the following:

```
val numbers = 1 :: 2 :: 3 :: Nil
```

```
numbers: List[Int] = List(1, 2, 3)
```

The `::` operator is often read as "cons", referencing historically the name `cons` in Lisp for the primitive that constructs a cell with two values. Note that `::` is a right-associative binary operator, so it is parsed as like

```
val numbers = 1 :: (2 :: (3 :: Nil))
Nil.::(3).::(2).::(1)
```

```
numbers: List[Int] = List(1, 2, 3)
res9_1: List[Int] = List(1, 2, 3)
```

and as with other binary operators, it is just syntactic sugar for method call.

It is quite common to work with lists directly using pattern matching on `Nil` or `::`:

```
def isEmpty(l: List[Int]): Boolean = l match {
  case Nil => true
  case head :: tail => false
}
isEmpty(numbers)
```

```
defined function isEmpty
res10_1: Boolean = false
```

Note that the List API also defines a method `:::` for appending two lists together:

```
val numbers = List(1, 2, 3)
numbers ::: List(4, 5, 6)
```

```
numbers: List[Int] = List(1, 2, 3)
res11_1: List[Int] = List(1, 2, 3, 4, 5, 6)
```

The append method `:::` is a linear-time operation in the length of its left argument (i.e., $O($`numbers.length`$)$ in this example). Why must this be the case?

6.1.1.3 Immutability

As noted above, a `List` is an immutable, functional data structure.

```
numbers(1) = 20
```

```
cmd12.sc:1: value update is not a member of List[Int]
did you mean updated?
val res12 = numbers(1) = 20
            ^Compilation Failed

:
Compilation Failed
```

While it is not common to use this method, the closest analogue is return a new list that is the original list element at index `1` `updated`:

```
val numbers_ = numbers.updated(1, 20)
```

```
numbers_: List[Int] = List(1, 20, 3)
```

6.1.1.4 Iterators

While it is also not common to use, a `for` loop in Scala enables iteration through a `List`:

```
for (n <- numbers) println(n)
```

```
1
2
3
```

A `for` loop in Scala is actually syntactic sugar for a method call to a higher-order method `foreach`:

```
numbers foreach { n => println(n) }
```

```
1
2
3
```

We call `foreach` higher-order, as it takes a function as a parameter of type `Int => Unit` (e.g., `{ n => println(n) }` in the above example). The function parameter, sometimes called a *callback*, describes what to do for each element of the list.

Note that `println` is a function that conforms to `Int => Unit`, so we could pass it directly:

```
numbers.foreach(println)
```

```
1
2
3
```

Higher-order methods are the most common way to use `List`s. However, `foreach` is less used than others, as the callback of type `Int => Unit` must inherently be imperative to do anything interesting. Why? Consider the type of the function we get that awaits receiving the callback to `foreach`:

```scala
val awaitingCallback: (Int => Unit) => Unit = numbers.foreach(_)
awaitingCallback(println)
```

```
1
2
3

awaitingCallback: Int => Unit => Unit = ammonite.$sess.cmd16$Helper$$Lambda$2150/0x00000000800
```

In the above, read `numbers.foreach(_)` more like `numbers.foreach`. The extra `(_)` is a low-level Scalaism to convert a method into a function that is only needed for this explanation and rarely needed in practice.

6.1.1.4.1 Functional Traversals

Or as another example, consider trying to write a function `sum` that sums up a list of integers `List[Int]` with a `for` loop or `foreach`:

```scala
def sum(l: List[Int]): Int = {
  var acc = 0
  for (n <- l) acc += n
  acc
}
sum(numbers)
```

```
defined function sum
res17_1: Int = 6
```

The only way to remember the accumulated sum so far is with a mutable variable `var acc`.

Instead, the idiomatic way to compute such a sum is to use another higher-order method that permits accumulation in a functional manner, such as `reduce`:

```scala
def sum(l: List[Int]): Int = l reduce { (acc, n) => acc + n }
sum(numbers)
```

```
defined function sum
res18_1: Int = 6
```

Not only does the `sum` definition become a one-liner, but it decouples the scheduling of work on each element of the list and lets the library implement that (e.g., sequentially left-to-right, concurrently, or even distributed!).

6.1.1.4.2 Placeholder Syntax for Function Literals

While not necessarily recommended, one might sometimes see an alternative Scala syntax for function literals using placeholders `_`:

```
def sum(l: List[Int]): Int = l.reduce(_ + _)
sum(numbers)
```

```
defined function sum
res19_1: Int = 6
```

Each placeholder `_` corresponds to a formal parameter, so `_ + _` is syntactic sugar for `(x,y) => x + y`. Like with any diet, take sugar in moderation.

6.1.1.4.3 Composing Higher-Order Methods

We will consider in subsequent chapters how to effectively use such higher-order methods. For the moment, simply recognize that such higher-order methods exist and is the idiomatic way to work with `List`s. Furthermore, this API design is particularly powerful and becoming commonplace in almost all languages (even in Java!). For example, in big-data applications, this design enables *streaming* where the data can be consumed in an online manner as a stream. Consider the following for a taste:

```
val l = List(1, 2, 3, 4, 5, 6)
val sumEvens = l filter { i => i % 2 == 0 } reduce { (acc, n) => acc + n }
```

```
l: List[Int] = List(1, 2, 3, 4, 5, 6)
sumEvens: Int = 12
```

Or,

```
val sumEvens = l.filter(_ % 2 == 0).reduce(_ + _)
```

```
sumEvens: Int = 12
```

6.1.1.4.4 Object-Oriented Iterators

The `foreach` method is an abstraction of the object-oriented Iterator Pattern:

```scala
val it = numbers.iterator
while (it.hasNext) {
  val n = it.next()
  println(n)
}
```

```
1
2
3

it: Iterator[Int] = empty iterator
```

In essence, the `foreach` method uses the callback parameter to allow the client to specify the body of the `while` loop. A benefit is that common programming errors in using such an object-oriented API—like calling `it.next()` after the `it` has no more elements—cannot happen when using `foreach`.

```scala
it.next()
```

6.1.1.5 API Documentation

As alluded to above, Scala has a rich API for `List`s. Such libraries are designed to be extremely generic for many use cases, so they necessarily have a fair amount of complexity. Nonetheless, it is worthwhile getting used to reading such API documentation.

6.1.1.6 Arrays

A `List` is a immutable, functional sequential collection of elements of the same type (i.e., a singly-linked list), while an `Array` is a fixed-size, mutable indexable collection of elements of the same type. In this course, we have little need for `Array`, but it exists in Scala for particular use cases.

6.1.2 Options

Another commonly used built-in data type is `Option`. It is either a `None` for or a `Some` of some value:

```scala
val none: Option[Int] = None
val some: Option[Int] = Some(42)
```

```
none: Option[Int] = None
some: Option[Int] = Some(value = 42)
```

Or, using some API methods on `Option`:

```scala
val none: Option[Int] = Option.empty
val some: Option[Int] = Option(42)
```

```
none: Option[Int] = None
some: Option[Int] = Some(value = 42)
```

The `Option` type is useful for methods that may optionally return a value (i.e., would error in some cases).

For example, we may want to define a division method that returns `None` if the client attempts to divide by zero:

```scala
def div(n: Int, m: Int): Option[Int] = m match {
  case 0 => None
  case _ => Some(n / m)
}
```

```
defined function div
```

Or, as another example, the `head` method for `List`s errors if the input list is empty:

```scala
val emptyList: List[Int] = Nil
```

```
emptyList: List[Int] = List()
```

```scala
val h: Int = emptyList.head
```

Instead, the `headOption` method returns an `Option` using `None` for an empty list and `Some` for a non-empty list:

```
val h: Option[Int] = emptyList.headOption
```

```
h: Option[Int] = None
```

We can then work with options also using pattern matching:

```
def head(l: List[Int]): Option[Int] = l match {
  case Nil => None
  case h :: _ => Some(h)
}
head(List(1, 2, 3))
```

```
defined function head
res30_1: Option[Int] = Some(value = 1)
```

We can think of an `Option` value as a 0-or-1 element list and thus all of the higher-order iteration methods are available:

```
some.foreach(println)
```

```
42
```

6.1.3 Maps

Maps are particularly useful data structures for storing associations between *keys* and *values*. For example, we describe value environments for a programming language as maps from variables to values in Section 4.1.1.

```
type Env = Map[String,Int]
val env: Env = Map("nOranges" -> 4, "nApples" -> 7, "nPears" -> 10)
```

```
defined type Env
env: Env = Map("nOranges" -> 4, "nApples" -> 7, "nPears" -> 10)
```

We can lookup in maps based on a key:

```
env("nApples")
env.apply("nApples")
```

```
res33_0: Int = 7
res33_1: Int = 7
```

```
env("nDogs")
```

As we see above, since a given key may not exist in a map, there is a `get` method that instead returns an `Option`:

```
env contains "nApples"
env get "nApples"
env contains "nDogs"
env get "nDogs"
```

```
res35_0: Boolean = true
res35_1: Option[Int] = Some(value = 7)
res35_2: Boolean = false
res35_3: Option[Int] = None
```

Another commonly-used alterative to `apply` and `get` for lookup in a map is `getOrElse` that takes an extra parameter for what to return in the case that the key does not exist:

```
env getOrElse ("nDogs", 0)
```

```
res36: Int = 0
```

Finally, we often want to extend maps:

```
val env_ = env + ("nBananas" -> 17)
```

```
env_: Map[String, Int] = Map(
  "nOranges" -> 4,
  "nApples" -> 7,
  "nPears" -> 10,
  "nBananas" -> 17
)
```

Note that just like with `List`, the above is a "functional update" that returns a new `Map` where `env` still exists:

```
env
env_
```

```
res38_0: Env = Map("nOranges" -> 4, "nApples" -> 7, "nPears" -> 10)
res38_1: Map[String, Int] = Map(
  "nOranges" -> 4,
  "nApples" -> 7,
  "nPears" -> 10,
  "nBananas" -> 17
)
```

A functional update is sometimes where one might want to intentionally shadow (i.e., write `val env = env + ("nBananas" -> 17)` in the above) to prevent referencing the unextended `env` in a particular scope.

We use the `->` operator with maps for visual clarity, but it is actually not special. It is just an alias for constructing pairs:

```
"nBananas" -> 17
```

```
res39: (String, Int) = ("nBananas", 17)
```

```
val env_ = env + (("nBananas", 17))
```

```
env_: Map[String, Int] = Map(
  "nOranges" -> 4,
  "nApples" -> 7,
  "nPears" -> 10,
  "nBananas" -> 17
)
```

As a collection in the standard library, `Map` also has the usual higher-order iteration methods, such as

```
env.foreach(println)
```

```
(nOranges,4)
(nApples,7)
(nPears,10)
```

6.1.4 Sets

The Scala standard library has many other core functional data structures. Another commonly used one is the `Set` data structure that keeps a single copy of each element while supporting fast membership testing, union, intersection, and iteration operations.

6.2 Classes

Scala is also an object-oriented language where code and data can be encapsulated together. A class declaration introduces a new type name, specifies data that it packages together, and defines methods that operate on that data:

```scala
class Dog(name: String, breed: String, age: Int) {
  override def toString =
    s"Woof! My name is $name, I am a $breed, and I am $age years old."
}
val samuel = new Dog("Samuel", "Alsatian", 11)
val bo = new Dog("Bo", "Portuguese Water Dog", 10)
samuel.toString
```

```
defined class Dog
samuel: Dog = Woof! My name is Samuel, I am a Alsatian, and I am 11 years old.
bo: Dog = Woof! My name is Bo, I am a Portuguese Water Dog, and I am 10 years old.
res42_3: String = "Woof! My name is Samuel, I am a Alsatian, and I am 11 years old."
```

In the above, we define a method `toString` that overrides the default `toString` method that is defined for all objects—that we can see is used as the printer above.

We can see that defining classes in Scala is quite convenient, eliminating a lot of the repetitive boilerplate code with constructors, field declarations, etc. seen in, for example, Java and C++.

Scala has a shorthand for defining a class with single instance (sometimes called a *singleton*) with the `object` keyword:

```scala
object Dog {
  def birth(name: String, breed: String): Dog =
    new Dog(name, breed, 0)
}
val sadie = Dog.birth("Sadie", "Pointer")
sadie.toString
```

```
defined object Dog
sadie: Dog = Woof! My name is Sadie, I am a Pointer, and I am 0 years old.
res43_2: String = "Woof! My name is Sadie, I am a Pointer, and I am 0 years old."
```

A `object` is like a module with function definitions, type definitions, etc. If an `object` has the same name as a `class` in the same file, then it is the *companion* object for the class and has special accessibility to that class's instances.

6.2.1 Data Classes

In relation to Java or C++, we can see the parameters to the class on line 1 as the parameter list to the constructor to create private fields with the same name that are accessible by methods. However, those fields are not accessible outside of the class's methods:

```scala
samuel.name
```

```
cmd44.sc:1: value name is not a member of cmd44.this.cmd42.Dog
val res44 = samuel.name
                   ^Compilation Failed
```

```
:
Compilation Failed
```

Often, we just want classes that store associated data together. If the fields are immutable, it is perfectly acceptable for them to be accessible. We can do so as follows to make public `val` fields:

```scala
class Dog(val name: String, val breed: String, val age: Int)
val samuel = new Dog("Samuel", "Alsatian", 11)
samuel.name
```

```
defined class Dog
samuel: Dog = ammonite.$sess.cmd44$Helper$Dog@15f44dcf
res44_2: String = "Samuel"
```

However, in this case, we would like to treat the `Dog` just like a specialized tuple and use things like pattern matching, but we can't

```scala
val Dog(name, _, _) = samuel
```

```
cmd45.sc:1: object Dog is not a case class, nor does it have a valid unapply/unapplySeq membe
val Dog(name, _, _) = samuel
    ^Compilation Failed

:
Compilation Failed
```

We can do so by making `Dog` a `case class`:

```scala
case class Dog(name: String, breed: String, age: Int)
val samuel = Dog("Samuel", "Alsatian", 11)
samuel.name
val Dog(_, breed, _) = samuel
```

```
defined class Dog
samuel: Dog = Dog(name = "Samuel", breed = "Alsatian", age = 11)
res45_2: String = "Samuel"
breed: String = "Alsatian"
```

We can think of a `case class` as a "functional class" used for storing immutable data. We can define methods on such classes as well, but that is somewhat secondary.

6.3 Algebraic Data Types

In addition to a tuples grouping associated data together, we want to be able to define alternatives:

```scala
trait Pet
case class Dog(name: String, breed: String) extends Pet
case class Cat(name: String, breed: String) extends Pet

def greet(pet: Pet): String = pet match {
  case Dog(name, _) => s"Woof, $name!"
  case Cat(name, _) => s"Meow, $name!"
}

greet(Dog("Samuel", "Altsatian"))
greet(Cat("Jenkins", "Siamese"))
```

```
defined trait Pet
defined class Dog
defined class Cat
defined function greet
res46_4: String = "Woof, Samuel!"
res46_5: String = "Meow, Jenkins!"
```

A `trait` introduces a new type name and is roughly a `class` interface. In the above, `Dog` and `Cat` can be both be a `Pet`. The `greet` function takes as input a `Pet` and uses pattern matching to distinguish whether it is a `Dog` or a `Cat`.

`Pet` in the above example is a very simple algebraic data type. As an aside, an algebraic data type is named such because it combines *products* of data (i.e., tuples) and *sums* of data (i.e., cases).

6.3.1 Option

The built-in collection types `Option` and `List` described above are both algebraic data types. Consider the following `Option`-like definition:

```scala
sealed trait MyOption
case object MyNone extends MyOption
case class MySome(i: Int) extends MyOption

val none: MyOption = MyNone
val some: MyOption = MySome(42)

def getOrElse(o: MyOption, default: Int): Int = o match {
  case MyNone => default
  case MySome(i) => i
}

getOrElse(none, 0)
getOrElse(some, 0)
```

```
defined trait MyOption
defined object MyNone
defined class MySome
none: MyOption = MyNone
some: MyOption = MySome(i = 42)
defined function getOrElse
res47_6: Int = 0
```

```
res47_7: Int = 42
```

One new thing to note is the `sealed` qualifier, which says that all the classes that derive from `MyOption` (i.e., the alternatives) are defined here, so the programmer and compiler do not need to worry about other possible cases for `MyOption`.

Algebraic data types can also be recursive. Recursive data types is exemplified by `List`s, which we revisit subsequentlyin **?@sec-recursion**.

6.3.2 Parametric Polymorphism

One difference between the built-in `Option` and `MyOption` in the above is that `Option` is parametrized by a type of the value that may or may not exist. We can extend our definition of `MyOption` with a type parameter `A` as follows:

```
sealed trait MyOption[A]
case class MyNone[A]() extends MyOption[A]
case class MySome[A](v: A) extends MyOption[A]

val none: MyOption[Int] = MyNone()
val some: MyOption[Int] = MySome(42)

def getOrElse[A](o: MyOption[A], default: A): A = o match {
  case MyNone() => default
  case MySome(v) => v
}

getOrElse(none, 0)
getOrElse(some, 0)
```

```
defined trait MyOption
defined class MyNone
defined class MySome
none: MyOption[Int] = MyNone()
some: MyOption[Int] = MySome(v = 42)
defined function getOrElse
res48_6: Int = 0
res48_7: Int = 42
```

Observe that `getOrElse` (as well as the `MyNone` and `MySome` constructors) have a type parameter list (written brackets `[]`) and a value parameter list (written in parentheses `()`).

The `getOrElse` function is *generic* in the parametrized type `A`. Being generic, the `getOrElse` function is also called *parametric polymorphic*.

Note that `MyOption` is not quite the same definition as `Option`, but it is close.

7 Exercise: Expressions and Data Types

The purpose of this assignment is to warm-up with Scala.

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

- thinking in terms of types, values, and expressions; and
- imperative iteration.

Instructions

This assignment asks you to write Scala code. There are restrictions associated with how you can solve these problems. Please pay careful heed to those. If you are unsure, ask the course staff.

Note that `???` indicates that there is a missing function or code fragment that needs to be filled in. In Scala, it is also an expression that throws a `NotImplementedError` exception. Make sure that you remove the `???` and replace it with the answer.

Use the test cases provided to test your implementations. You are also encouraged to write your own test cases to help debug your work. However, please delete any extra cells you may have created lest they break an autograder.

7.1 Type Checking

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. In this question, we will reason for ourselves that `g` is indeed well-typed.

```
def g(x: Int) = /*e1*/({
  // env1
  val (a, b) = /*e2*/(
    // env2
    (1, (x, 3))
  )
  /*e3*/(
    // env3
    if (x == 0) (b, 1) else (b, a + 2)
  )
})
```

```
defined function g
```

We have added parentheses around 3 key sub-expressions of the body of `g` (e.g., `/*e1*/(...)`) and noted that there are corresponding environments (e.g., `// env1` for each sub-expression).

Exercise 7.1 (2 points)**.** What is the type environment `env1`? Briefly explain.

Use either format shown in Section 4.1.1 (e.g., $[x_1 : \tau_1, \ldots, x_n : \tau_n]$).

???

Exercise 7.2 (2 points)**.** What is the type environment `env2`? Briefly explain.

???

Exercise 7.3 (10 points)**.** Derive the type of expression `e2`.

Showing the type for each sub-expression of `e2`; stop when you reach literals or variable uses.

???

Notes

Use the format shown in Section 3.2.1. . Here's such a derivation for the type of the expression `(1 + 2) + (3 + 4)`.

```
(1 + 2) + (3 + 4): Int
- if 1 + 2: Int
  - if 1: Int
  - if 2: Int
- if 3 + 4: Int
  - if 3: Int
  - if 4: Int
```

Exercise 7.4 (2 points)**.** What is the type environment `env3`? Briefly explain.

???

Exercise 7.5 (2 points)**.** Derive the type of expression `e3`. {.unnumbered}

???

Exercise 7.6 (9 points)**.** Confirm your derivations by adding type assertions to each sub-expression of `g` and adding the return type of `g`.

That is, replace sub-expressions e of `g` with expressions e `:` τ. You may need to add some parentheses—`(` e `:` τ `)`—to preserve the syntactic structure. Skip adding typing assertions for literals and variable uses.

Edit this cell:

```
def g(x: Int) = /*e1*/({
  // env1
  val (a, b) = /*e2*/(
    // env2
    (1, (x, 3))
  )
  /*e3*/(
    // env3
    if (x == 0) (b, 1) else (b, a + 2)
  )
})
```

```
defined function g
```

???

Hint: There are 8 sub-expressions that are not literals nor variable uses that need typing assertions (i.e., e : τ), plus 1 more annotation for the return type of `g`.

7.2 Unit Testing

When starting to program in the large, it is useful to use a testing framework to manage tests and integrate with IDEs. One that is commonly used in Scala is ScalaTest.

While it is somewhat overkill for testing small exercises like the ones to come, we practice here writing tests using ScalaTest.

To load the ScalaTest library, run the following cell:

```
// RUN this cell FIRST before testing
import $ivy.`org.scalatest::scalatest:3.2.19`, org.scalatest._, events._, flatspec._
def report(suite: Suite) = suite.execute(stats = true)
def assertPassed(suite: Suite) =
  suite.run(None, Args(new Reporter {
    def apply(e: Event) = e match {
      case e @ (_: TestFailed) => assert(false, s"${e.message} (${e.testName})")
      case _ => ()
    }
  }))
def test(suite: Suite) = {
  report(suite)
  assertPassed(suite)
}
```

```
import $ivy.$                                , org.scalatest._, events._, flatspec._

defined function report
defined function assertPassed
defined function test
```

Exercise 7.7 (3 points)**.** Unit Test `plus`. For this question, **edit the next two code cells** to fix the implementation of `plus` and add the appropriate assertion for the third test case `"add (3,4) == 7"`.

Our goal is unit test the following complicated function (that we've gotten wrong!):

```
def plus(n: Int, m: Int): Int =
  ???
```

```
defined function plus
```

To use ScalaTest, we create "Spec" objects using ScalaTest methods like `should` that define an embedded domain-specific language (DSL) for defining tests:

```scala
val plusSuite = new AnyFlatSpec {
  // Define a *subject* to test (e.g., "plus").
  // After `should`, name a test (e.g, "add (1, 1) == 2").
  // After `in`, specify assertions (e.g., `assert(plus(1,1) == 2))
  "plus" should "add 1 + 1 == 2" in {
    // Specify assertions here.
    assert(plus(1,1) == 2)
  }

  it should "add 2 + 2 == 4" in {
    // It is convenient to distinguish the expected result from the code that
    // you're testing, which affects the error messages when the test fails.
    assertResult(2 + 2) {
      plus(2,2)
    }
  }

  it should "add 3 + 4 == 7"  in {
    ???
  }
}
report(plusSuite)
```

```
Run starting. Expected test count is: 3
cmd4$Helper$$anon$1:
plus
- should add 1 + 1 == 2 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd3$Helper.plus(cmd3.sc:2)
  at ammonite.$sess.cmd4$Helper$$anon$1.$anonfun$new$1(cmd4.sc:7)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
```

```
- should add 2 + 2 == 4 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd3$Helper.plus(cmd3.sc:2)
  at ammonite.$sess.cmd4$Helper$$anon$1.$anonfun$new$2(cmd4.sc:14)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should add 3 + 4 == 7 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd4$Helper$$anon$1.$anonfun$new$3(cmd4.sc:19)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  at org.scalatest.TestSuite.withFixture$(TestSuite.scala:195)
  ...
Run completed in 71 milliseconds.
Total number of tests run: 3
Suites: completed 1, aborted 0
Tests: succeeded 0, failed 3, canceled 0, ignored 0, pending 0
*** 3 TESTS FAILED ***

plusSuite: AnyFlatSpec = cmd4$Helper$$anon$1
```

???

7.3 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 programming.

Exercise 7.8 (4 points)**.** Write and test a function `abs` {.unnumbered}

Edit this cell:

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`.

Notes

- Do not use any Scala library functions.

Tests

Edit this cell:

```
val absSuite = new AnyFlatSpec {
  "abs" should "abs(2) == 2" in {
     assert(abs(2) == 2)
  }
  it should "abs(-2) == 2" in {
     assert(abs(-2) == 2)
  }
  it should "abs(0) == 0" in {
     assert(abs(0) === 0)
  }
  it should "???1" in {
    ???
  }
}
report(absSuite)
```

```
Run starting. Expected test count is: 4
cmd7$Helper$$anon$1:
abs
- should abs(2) == 2 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd6$Helper.abs(cmd6.sc:2)
  at ammonite.$sess.cmd7$Helper$$anon$1.$anonfun$new$1(cmd7.sc:3)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should abs(-2) == 2 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd6$Helper.abs(cmd6.sc:2)
  at ammonite.$sess.cmd7$Helper$$anon$1.$anonfun$new$2(cmd7.sc:6)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should abs(0) == 0 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd6$Helper.abs(cmd6.sc:2)
  at ammonite.$sess.cmd7$Helper$$anon$1.$anonfun$new$3(cmd7.sc:9)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should ???1 *** FAILED ***
```

```
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd7$Helper$$anon$1.$anonfun$new$4(cmd7.sc:12)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  at org.scalatest.TestSuite.withFixture$(TestSuite.scala:195)
  ...
Run completed in 3 milliseconds.
Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 0, failed 4, canceled 0, ignored 0, pending 0
*** 4 TESTS FAILED ***
```

```
absSuite: AnyFlatSpec = cmd7$Helper$$anon$1
```

???

Exercise 7.9 (4 points)**.** Write and test a function `xor`

Edit this cell:

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`**.

???

Notes

- For practice, do not use the Boolean operators. Instead, only use the **`if`**- expression and the Boolean literals (i.e., **`true`** or **`false`**).

Tests

Edit this cell:

```scala
val xorSuite = new AnyFlatSpec {
  "xor" should "!xor(true, true)" in {
    assert(!xor(true, true))
  }
  it should "xor(true, false)" in {
    assert(xor(true, false))
  }
  it should "???1" in {
    ???
  }
  it should "???2" in {
    ???
  }
}
report(xorSuite)
```

```
Run starting. Expected test count is: 4
cmd10$Helper$$anon$1:
xor
- should !xor(true, true) *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd9$Helper.xor(cmd9.sc:2)
  at ammonite.$sess.cmd10$Helper$$anon$1.$anonfun$new$1(cmd10.sc:3)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should xor(true, false) *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd9$Helper.xor(cmd9.sc:2)
  at ammonite.$sess.cmd10$Helper$$anon$1.$anonfun$new$2(cmd10.sc:6)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
```

```
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  ...
- should ???1 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd10$Helper$$anon$1.$anonfun$new$3(cmd10.sc:9)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  at org.scalatest.TestSuite.withFixture$(TestSuite.scala:195)
  ...
- should ???2 *** FAILED ***
  scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:345)
  at ammonite.$sess.cmd10$Helper$$anon$1.$anonfun$new$4(cmd10.sc:12)
  at org.scalatest.OutcomeOf.outcomeOf(OutcomeOf.scala:85)
  at org.scalatest.OutcomeOf.outcomeOf$(OutcomeOf.scala:83)
  at org.scalatest.OutcomeOf$.outcomeOf(OutcomeOf.scala:104)
  at org.scalatest.Transformer.apply(Transformer.scala:22)
  at org.scalatest.Transformer.apply(Transformer.scala:20)
  at org.scalatest.flatspec.AnyFlatSpecLike$$anon$5.apply(AnyFlatSpecLike.scala:1832)
  at org.scalatest.TestSuite.withFixture(TestSuite.scala:196)
  at org.scalatest.TestSuite.withFixture$(TestSuite.scala:195)
  ...
Run completed in 4 milliseconds.
Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 0, failed 4, canceled 0, ignored 0, pending 0
*** 4 TESTS FAILED ***

xorSuite: AnyFlatSpec = cmd10$Helper$$anon$1
```

7.4 Imperative Iteration and Complexity

Exercise 7.10 (5 points)**.** Write a function `filterPairsByBound`.

Edit this cell:

that given a list of pairs of integers, for example,

```
val input1_l = List( (1, 5), (2, 7), (15, 14), (18, 19), (14, 28), (0,0), (35, 24) )
```

```
input1_l: List[(Int, Int)] = List(
  (1, 5),
  (2, 7),
  (15, 14),
  (18, 19),
  (14, 28),
  (0, 0),
  (35, 24)
)
```

output a list consisting of just those pairs (n_1, n_2) in the original list wherein $|n_1 - n_2| \leq k$ where k is an integer given as input. Ensure that the order of the elements in the output list is the same as that in the input list.

For the list `input1`, the expected output, with `k == 1`, the expected output is as follows:

With `k == 4`, the expected output is as follows:

???

Notes

- Your function must be called `filterPairsByBound` with two arguments: (1) a list of pairs of integers, and (2) the k value. It must return a list of pairs of integers.
- You can use `for`-loops (or `foreach`) and the following operators for concatenating elements to a list:
  - `:::` appends two lists together.
  - `::` puts an element on the front of a list.
- You can use the `List` API method `reverse`. You may also use the `Int abs` method to obtain the absolute value of an integer (or use your `abs` function above).
- You should not use any other `List` API functions including `filter`, `map`, `foldLeft`, `foldRight`, etc. Plenty of time to learn them properly later on.
- Do not try to convert your list to an array or vector so that you can mutate it.
- If you are unable to solve the problem without violating the restrictions or unsure, ask us.
- You will need to use `var` given the above restrictions.

Hints

- In Scala, pairs of integers have the type `(Int, Int)`.
- A list containing pairs of integers has the type `List[(Int, Int)]`.
- Recall from the notes, here is how one iterates over the elements of a list in Scala:

```scala
val list = List(1, 2, 3)

for (elt <- list) {
  // do stuff with elt
  println(elt)
}
```

```
1
2
3


list: List[Int] = List(1, 2, 3)
```

- Append an element to the end of a list and update a `var`:

```scala
var resultList = List(1, 2, 3)
val elt = 42

resultList = resultList :+ elt
```

```
resultList: List[Int] = List(1, 2, 3, 42)
elt: Int = 42
```

- Or, append an element to the end of a list using list concatenation and update a `var`:

```scala
var resultList = List(1, 2, 3)
val elt = 42

resultList = resultList ::: List(elt)
```

```
resultList: List[Int] = List(1, 2, 3, 42)
elt: Int = 42
```

- Prepend an element and update a `var`:

```
var resultList = List(1, 2, 3)
val elt = 42

resultList = elt :: resultList
```

```
resultList: List[Int] = List(42, 1, 2, 3)
elt: Int = 42
```

- *Warning*: The `:::` or other operations appending operations take linear $O(n)$ time where n is the length of the (left) list. Thus, we will often try to avoid using these operations, but it is ok to use it for this particular part.

Tests

Exercise 7.11 (7 points)**.** Write a function `filterPairsByBoundLinearTime`

Edit this cell:

If you followed the hint and ignored the linear-time warning in the previous part Exercise 7.10, then you would have used the `:::` operation to append an element to the end of a list at each step.

```
for (... <- list) {
  // iterate over a loop
  ...
  newList = newList ::: List(newElement) // This takes O(length of newList).
}
```

Each `:::` operation requires a full list traversal to find the end of `newList` and then append to it. The overall algorithm thus requires $O(n^2)$ time where is the length of the original `list` (also the number of loop iterations).

To illustrate, cut-and-paste and then run this code in a new test cell. Just remember to delete that cell before you submit. It will take a long time to run.

```
// Create a list of 1,000,000 pairs
val longTestList = (1 to 1000000).map(x => (x, x - 1)).toList
// Run the function you wrote
filterPairsByBound(longTestList, 1)
// This will take a long time to finish.
```

In this problem, we wish to implement a function `filterPairsByBoundLinearTime` that solves the exact same problem as the previous part Exercise 7.10 but takes time linear in the size of the input list.

To do so, we would like you to use the `::` (read "cons") operator on a list that prepends an element to the front of a list, instead of `:+` or `:::` that appends to the back of a list.

You will want to use the `List reverse` API method:

```
val list = List(1, 2, 5, 6, 7, 8)
val r = list.reverse
```

```
list: List[Int] = List(1, 2, 5, 6, 7, 8)
r: List[Int] = List(8, 7, 6, 5, 2, 1)
```

The `r` has the reverse of `list`, and it works in linear time in the length of `list`.

The restrictions remain the same as the previous part Exercise 7.10, but we would like you to focus on ensuring that your solution runs in linear time.

???

Tests

Submission

Submission Instructions

If you are a University of Colorado Boulder student, we use Gradescope for assignment submission. In summary,

- ☐ Work on a copy of this Jupyter notebook.
- ☐ Submit it to the corresponding Gradescope assignment entry for grading.

GitHub and Gradescope

We use GitHub Classroom for assignment distribution, which gives you a private GitHub repository to work on your assignment. While using GitHub is perhaps overkill for this assignment, it does give you the ability to version and save incremental progress on GitHub (lest your laptop fails) and makes it easier to get help from the course staff. It will also become particularly useful when more files are involved, and it is never too early to get used to the workflow professional software engineers use with Git.

To use use GitHub and Gradescope,

- ☐ Create a private GitHub repository by clicking on a GitHub Classroom link from the corresponding Canvas assignment entry.
- ☐ Clone your private GitHub repository to your development environment (using the `<> Code` button on GitHub to get the repository URL).
- ☐ Work on the copy of this Jupyter notebook from your cloned repository. Use Git to save versions on GitHub (e.g., `git add`, `git commit`, `git push` on the command line, via JupyterLab, or via VSCode).
- ☐ Submit to the corresponding Gradescope assignment entry for grading by choosing GitHub as the submission method.

You need to have a GitHub identity and must have your full name in your GitHub profile in case we need to associate you with your submissions.

8 Recursion, Induction, and Iteration

Thus far in our programming, 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 $n \geq 1$:

$$n! = n \cdot (n-1)!$$

Based on this equation, we can define a Scala function to compute factorial as follows:

```scala
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
factorial(3)
```

```
defined function factorial
res0_1: Int = 6
```

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

`factorial(3)`	$\longrightarrow^*$	`if (3 == 0) 1 else 3 * factorial(3 - 1)`
	$\longrightarrow^*$	`3 * factorial(2)`
	$\longrightarrow^*$	`3 * 2 * factorial(1)`
	$\longrightarrow^*$	`3 * 2 * 1 * factorial(0)`
	$\longrightarrow^*$	`3 * 2 * 1 * (if (0 == 0) 1 else 1 * factorial(0 - 1))`
	$\longrightarrow^*$	`3 * 2 * 1 * 1`
	$\longrightarrow^*$	`6`

where the sequence above is shorthand for expressing that each successive pair of expressions is related by the multi-step evaluation relation $\longrightarrow^*$ written between them.

Observe that the variable `factorial` needs to be in scope in the function body (i.e., the expression after =) to enable the recursive definition. To define a recursive function, the return type `: Int` has to be given for `factorial` to be in scope in the function body. (Why? To enable static type checking.)

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

```
cmd1.sc:1: recursive method factorial needs result type
def factorial(n: Int) = if (n == 0) 1 else n * factorial(n - 1)
                                               ^Compilation Failed


:
Compilation Failed
```

8.1 Induction: Reasoning about Recursive Programs

Induction is important proof technique for reasoning about recursively-defined objects that you might recall from a discrete mathematics course. Here, we consider 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 (i.e., $\llcorner n \lrcorner$: `Int` as long as n is representable as an `Int`).

Theorem 8.1. *For all integers n such that $n \geq 0$,*

$$\mathtt{factorial}(\llcorner n \lrcorner) \longrightarrow^* \llcorner n! \lrcorner \;.$$

Proof. By mathematical induction on n.

Case $n = 0$: Note that $\llcorner 0 \lrcorner = \mathtt{0}$. Taking a few steps of evaluation, we have that

$$\mathtt{factorial(0)} \longrightarrow^* \mathtt{1} \;.$$

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

Case $n = n' + 1$ for some $n' \geq 0$: The induction hypothesis is as follows:

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

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

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

because we know that $n \neq 0$.

Applying the induction hypothesis (observing that $n - 1 = n'$), we have that

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

By further evaluation, we have that

$$\llcorner n \lrcorner \texttt{ * } \llcorner n'! \lrcorner \longrightarrow \llcorner n \cdot n'! \lrcorner \,.$$

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

$\square$

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.

8.2 Pattern Matching

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

Listing 8.1 Factorial: With Pattern Matching

```scala
def factorial(n: Int): Int = n match {
  case 0 => 1
  case _ => n * factorial(n - 1)
}
factorial(3)
```

```
defined function factorial
res1_1: Int = 6
```

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 `cases`. 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 detail in **?@sec-data-structures-and-pattern-matching**. For the moment, simply recognize that patterns in general bind names (like seen previously in Section 4.4.4). In Listing 8.1, we use the "wildcard" pattern `_` to match anything that is non-zero.

8.3 Function Preconditions

The definitions of `factorial` given aboe and implicitly assume that they are called with non-negative integer values. Consider evaluating `factorial(-2)`:

`factorial(-2)`	$\longrightarrow^*$	`-2 * factorial(-3)`
	$\longrightarrow^*$	`-2 * -3 * factorial(-4)`
	$\longrightarrow^*$	`-2 * -3 * -4 * factorial(-5)`
	$\longrightarrow^*$	`-2 * -3 * -4 * -5 * factorial(-5)`
	$\longrightarrow^*$	...

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

```scala
def factorial(n: Int): Int = {
  require(n >= 0)
  n match {
    case 0 => 1
```

```
    case _ => n * factorial(n - 1)
  }
}
factorial(-2)
```

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

```
def factorial(n: Int): Int = {
  require(n >= 0)
  def f(n: Int): Int = n match {
    case 0 => 1
    case _ => n * f(n - 1)
  }
  f(n)
}
factorial(3)
```

```
defined function factorial
res3_1: Int = 6
```

Here, the `f` function is local to the `factorial` function. The `f` does not do any checking on its argument, but the `require` check in `factorial` will ensure that `f` always terminates.

8.4 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 .

```
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)
}
factorial(3)
```

```
defined function factorial
res4_1: Int = 6
```

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

`factorial(3)`	$\longrightarrow^*$	`loop(1, 3)`
	$\longrightarrow^*$	`loop(1 * 3, 2)`
	$\longrightarrow^*$	`loop(3 * 2, 1)`
	$\longrightarrow^*$	`loop(6 * 1, 0)`
	$\longrightarrow^*$	`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 there is 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.

A tail-recursive function corresponds closely to a loop (e.g., a `while` loop) but does not require mutation. For example, consider the following imperative version of `factorial`:

```
def factorial(n: Int): Int = {
  require(n >= 0)
  println(s"factorial(n = $n)")
  var acc = 1
  var i = n
  while (i != 0) {
```

```scala
    println(s"acc -> $acc, i -> $i")
    acc = acc * i
    i = i - 1
  }
  println(s"acc -> $acc, i -> $i")
  acc
}
factorial(3)
```

```
factorial(n = 3)
acc -> 1, i -> 3
acc -> 3, i -> 2
acc -> 6, i -> 1
acc -> 6, i -> 0
```

```
defined function factorial
res5_1: Int = 6
```

Conceptually, each iteration of the `while` loop corresponds to a call to `loop`. The value of `acc` and `i` in each iteration of the `while` loop correspond to the values bound to `acc` and `n` on each tail-recursive call to `loop`. We see this by comparing the instrumentation to print the values of `acc` and `i` on each loop iteration and the values of `acc` and `n` in each tail-recursive call.

```scala
def factorial(n: Int): Int = {
  require(n >= 0)
  println(s"factorial(n = $n)")
  def loop(acc: Int, n: Int): Int = {
    println(s"-->* loop(acc = $acc, n = $n)")
    n match {
      case 0 => acc
      case _ => loop(acc * n, n - 1)
    }
  }
  val r = loop(1, n)
  println(s"-->* $r")
  r
}
factorial(3)
```

```
factorial(n = 3)
```

```
-->* loop(acc = 1, n = 3)
-->* loop(acc = 3, n = 2)
-->* loop(acc = 6, n = 1)
-->* loop(acc = 6, n = 0)
-->* 6
```

```
defined function factorial
res6_1: Int = 6
```

8.5 Exercise: Exponentiation

Exercise 8.1. A very similar example to `factorial` is to define the exponentiation function `exp` that computes x^n for $n \geq 0$.

```
def exp(x: Int, n: Int): Int = {
  require(n >= 0)
  ???
}
assert(exp(2,4) == 16)
```

8.6 Exercise: Tail-Recursive Fibonacci

Let us consider the `fibonacci` function that computes the n^{th} Fibonacci number:

```
def fibonacci(n: Int): Int = {
  require(n >= 0)
  n match {
    case 0 | 1 => 1
    case _ => fibonacci(n - 1) + fibonacci(n - 2)
  }
}
```

```
defined function fibonacci
```

The `fibonacci` function is more interesting than `factorial` because it makes two recursive calls. Is it terminating on all input n? Yes, we can reason by induction just like with `factorial`.

Is it tail recursive? Most definitely not, as each recursive call awaits the result of the other recursive call to then apply + on the results. This is potentially problematic because each call requires an allocation of a stack frame.

For `fibonacci(` n `)`, how many recursive calls are made? Let's consider an instrumented version that records the stack `depth` of and the `count` on total calls to `f`:

```
def fibonacci(n: Int): Int = {
  require(n >= 0)
  println(s"factorial($n)")
  def f(n: Int, depth: Int, count: Int): (Int, Int) = {
    val r = n match {
      case 0 | 1 => (1, count)
      case _ => {
        val (b, countb) = f(n - 1, depth + 1, count + 1)
        val (a, counta) = f(n - 2, depth + 1, countb + 1)
        (a + b, counta)
      }
    }
    println(s"${"  " * depth}- f(n = $n, depth = $depth, count = $count) = $r")
    r
  }
  val (r, _) = f(n, 0, 1)
  r
}
fibonacci(0)
fibonacci(1)
fibonacci(2)
fibonacci(3)
fibonacci(4)
fibonacci(5)
```

```
factorial(0)
- f(n = 0, depth = 0, count = 1) = (1,1)
factorial(1)
- f(n = 1, depth = 0, count = 1) = (1,1)
factorial(2)
  - f(n = 1, depth = 1, count = 2) = (1,2)
  - f(n = 0, depth = 1, count = 3) = (1,3)
- f(n = 2, depth = 0, count = 1) = (2,3)
factorial(3)
    - f(n = 1, depth = 2, count = 3) = (1,3)
    - f(n = 0, depth = 2, count = 4) = (1,4)
```

```
  - f(n = 2, depth = 1, count = 2) = (2,4)
  - f(n = 1, depth = 1, count = 5) = (1,5)
- f(n = 3, depth = 0, count = 1) = (3,5)
factorial(4)
      - f(n = 1, depth = 3, count = 4) = (1,4)
      - f(n = 0, depth = 3, count = 5) = (1,5)
    - f(n = 2, depth = 2, count = 3) = (2,5)
    - f(n = 1, depth = 2, count = 6) = (1,6)
  - f(n = 3, depth = 1, count = 2) = (3,6)
    - f(n = 1, depth = 2, count = 8) = (1,8)
    - f(n = 0, depth = 2, count = 9) = (1,9)
  - f(n = 2, depth = 1, count = 7) = (2,9)
- f(n = 4, depth = 0, count = 1) = (5,9)
factorial(5)
        - f(n = 1, depth = 4, count = 5) = (1,5)
        - f(n = 0, depth = 4, count = 6) = (1,6)
      - f(n = 2, depth = 3, count = 4) = (2,6)
      - f(n = 1, depth = 3, count = 7) = (1,7)
    - f(n = 3, depth = 2, count = 3) = (3,7)
      - f(n = 1, depth = 3, count = 9) = (1,9)
      - f(n = 0, depth = 3, count = 10) = (1,10)
    - f(n = 2, depth = 2, count = 8) = (2,10)
  - f(n = 4, depth = 1, count = 2) = (5,10)
      - f(n = 1, depth = 3, count = 13) = (1,13)
      - f(n = 0, depth = 3, count = 14) = (1,14)
    - f(n = 2, depth = 2, count = 12) = (2,14)
    - f(n = 1, depth = 2, count = 15) = (1,15)
  - f(n = 3, depth = 1, count = 11) = (3,15)
- f(n = 5, depth = 0, count = 1) = (8,15)

defined function fibonacci
res9_1: Int = 1
res9_2: Int = 1
res9_3: Int = 2
res9_4: Int = 3
res9_5: Int = 5
res9_6: Int = 8
```

Unfortunately, the growth of the number of recursive calls is exponential in n. Thus, to compute `fibonacci(40)` requires us to make more than a billion calls.

However, we can see from above that there is a wasted work in repeatedly computing smaller Fibonacci numbers. We can define a tail-recursive version of the `fibonacci` function by computing the n^{th} Fibonacci number "bottom up" starting from the 0^{th}, 1^{st}, 2^{nd}, 3^{rd}, ... (using what you might remember from other classes as *dynamic programming*).

Exercise 8.2. Give a tail-recursive definition `fib` of the Fibonacci function:

```
def fib(n: Int): Int = {
  require(n >= 0)
  ???
}
```

```
defined function fib
```

See that you can compute much larger Fibonacci numbers using your linear-time tail-recursive implementation `fib` compared to the direct recursive `fibonacci`.

9 Inductive Data Types

9.1 Lists

As we saw in Section 6.1.1, the `List` type constructor from the Scala library is defined with two constructors `Nil` and `::` (pronounced "cons"). A list is a basic inductive data type:

```
object MyList {
  sealed trait List[A]
  case class Nil[A]() extends List[A]
  case class ::[A](head: A, tail: List[A]) extends List[A]
}
```

```
defined object MyList
```

The `List` type constructor from Scala library is very close to the above. Observe that `List[A]` is a recursive type with the `tail` field of `::`.

Thus, the most direct way to implement functions on `List`s is using recursion and pattern matching. For example, defining a function to compute the length of a list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case _ :: t => 1 + length(t)
}
```

```
defined function length
```

The definition above using pattern matching very directly follows the inductive structure of the type. Observe that in a definition of `length` using an `if` expression

```
def length[A](l: List[A]): Int =
  if (l == Nil) 0
  else 1 + length(l.tail)
```

```
defined function length
```

it takes, for example, a bit of extra thought to realize that `l.tail` will never throw an exception.

We can also see why a function `append` (i.e., the `:::` method in the Scala library) that appends one list to another must necessarily be a linear-time operation over the left list `xl`:

```
def append[A](xl: List[A], yl: List[A]): List[A] = xl match {
  case Nil => yl
  case xh :: xt => xh :: append(xt, yl)
}
val xlyl_append = append(List(1, 2, 3), List(4, 5, 6))
val xlyl_::: = List(1, 2, 3) ::: List(4, 5, 6)
xlyl_append == xlyl_:::
```

```
defined function append
xlyl_append: List[Int] = List(1, 2, 3, 4, 5, 6)
xlyl_:::: List[Int] = List(1, 2, 3, 4, 5, 6)
res3_3: Boolean = true
```

Now, observing that `append` is not tail recursive, we might try to implement the following:

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

```
defined function buggyAppend
```

This is is not quite `append`. What does `buggyAppend` do?

```
val xlyl_buggyAppend = buggyAppend(List(1, 2, 3), List(4, 5, 6))
```

```
xlyl_buggyAppend: List[Int] = List(3, 2, 1, 4, 5, 6)
```

It reverses the first list `xl` and appends the second list `yl` to it. While a somewhat strange operation, it is tail recursive and in the standard library:

```scala
val xlyl_reverse_::: = List(1, 2, 3) reverse_::: List(4, 5, 6)
xlyl_buggyAppend == xlyl_reverse_:::
```

```
xlyl_reverse_:::: List[Int] = List(3, 2, 1, 4, 5, 6)
res6_1: Boolean = true
```

To define the `reverse` of a list `l`, we can use **`append`** to take an element on the head and append it to the `reverse` of the tail:

```scala
def reverse[A](l: List[A]): List[A] = l match {
  case Nil => Nil
  case h :: t => append(reverse(t), h :: Nil)
}
reverse(List(1, 2, 3, 4, 5))
```

```
defined function reverse
res7_1: List[Int] = List(5, 4, 3, 2, 1)
```

But what is the complexity of this function? It is $O(n^2)$ where n is the length of `l`!

Can we write a linear-time `reverse`? Looking at `buggyAppend`, we see how:

```scala
def reverse[A](l: List[A]): List[A] = {
  def rev(l: List[A], acc: List[A]): List[A] = l match {
    case Nil => acc
    case h :: t => rev(t, h :: acc)
  }
  rev(l, Nil)
}
reverse(List(1, 2, 3, 4, 5))
```

```
defined function reverse
res8_1: List[Int] = List(5, 4, 3, 2, 1)
```

Let's instrument this linear-time `reverse` to see it in action:

```
def reverse[A](l: List[A]): List[A] = {
  println(s"reverse($l)")
  def rev(l: List[A], acc: List[A]): List[A] = {
    println(s"-->* loop($l, $acc)")
    l match {
      case Nil => acc
      case h :: t => rev(t, h :: acc)
    }
  }
  val r = rev(l, Nil)
  println(r)
  r
}
reverse(List(1, 2, 3, 4, 5))
```

```
reverse(List(1, 2, 3, 4, 5))
-->* loop(List(1, 2, 3, 4, 5), List())
-->* loop(List(2, 3, 4, 5), List(1))
-->* loop(List(3, 4, 5), List(2, 1))
-->* loop(List(4, 5), List(3, 2, 1))
-->* loop(List(5), List(4, 3, 2, 1))
-->* loop(List(), List(5, 4, 3, 2, 1))
List(5, 4, 3, 2, 1)

defined function reverse
res9_1: List[Int] = List(5, 4, 3, 2, 1)
```

Observe that `rev` is exactly `buggyAppend`. The specification of `rev` (or `buggyAppend`) is that it returns the reverse of the its first argument followed by its second argument appended.

Previously, our discussion about tail recursion (Section 8.4) was simply about efficiency because the operators we considered were commutative (e.g., + or $*$ on `Int`s). Now, with a non-commutative operator like ::, we see that there is something more.

The intuition is that the accumulator parameter `acc` in `rev` enables us to "do something" as we "recurse down" the list. And the stack in a non-tail recursive function enables us to "do something" as we "return up".

9.2 Persistent Data Structures

Lists are special case of trees with one recursive parameter, so we see that values of user-defined inductive data types are in general trees. For example, we can define a binary tree of `Int`s:

```scala
sealed trait BinaryTree
case object Empty extends BinaryTree
case class Node(l: BinaryTree, d: Int, r: BinaryTree) extends BinaryTree

Node(Node(Empty, 2, Empty), 10, Node(Empty, 14, Node(Empty, 17, Empty)))
```

```
defined trait BinaryTree
defined object Empty
defined class Node
res10_3: Node = Node(
  l = Node(l = Empty, d = 2, r = Empty),
  d = 10,
  r = Node(l = Empty, d = 14, r = Node(l = Empty, d = 17, r = Empty))
)
```

One key application of immutable trees are for representing maps and sets with logarithmic lookup, insertion, and deletion using balanced search trees. First, consider making the `BinaryTree` type generic:

```scala
sealed trait BinaryTree[K,V]
case class Empty[K,V]() extends BinaryTree[K,V]
case class Node[K,V](l: BinaryTree[K,V], kv: (K, V), r: BinaryTree[K,V]) extends BinaryTree[

Node(Node(Empty(), 2 -> List("two", "dos", " "), Empty()), 10 -> List("ten", "diez", " "), No
```

```
defined trait BinaryTree
defined class Empty
defined class Node
res11_3: Node[Int, List[String]] = Node(
  l = Node(l = Empty(), kv = (2, List("two", "dos", "\u4e8c")), r = Empty()),
  kv = (10, List("ten", "diez", "\u5341")),
  r = Node(
    l = Empty(),
    kv = (14, List("fourteen", "catorce", "\u5341\u56db")),
    r = Node(
      l = Empty(),
      kv = (17, List("seventeen", "diecisiete", "\u5341\u4e03")),
      r = Empty()
    )
  )
)
```

Now, we do not want to directly construct such trees. Instead, we design an API for lookup, insertion, and deletion to maintain search (i.e., ordering) and balance invariants. Lookup, insertion, and deletion are logarithmic when the search and balance invariants are maintained.

The Scala `Map` and `Set` libraries are such search tree data structures.

```scala
val m = Map(2 -> List("two", "dos", " "), 10 -> List("ten", "diez", " "))
val newm = m + (14 -> List("fourteen", "catorce", " "))
```

```
m: Map[Int, List[String]] = Map(
  2 -> List("two", "dos", "\u4e8c"),
  10 -> List("ten", "diez", "\u5341")
)
newm: Map[Int, List[String]] = Map(
  2 -> List("two", "dos", "\u4e8c"),
  10 -> List("ten", "diez", "\u5341"),
  14 -> List("fourteen", "catorce", "\u5341\u56db")
)
```

The map `newm` is the map `m` with an additional key-value pair `14 -> List("fourteen", "catorce", " ")` inserted. Note that both the old version `m` and the new version `newm` exist:

```scala
val mOf10 = m(10)
val newmOf10 = newm(10)
```

```
mOf10: List[String] = List("ten", "diez", "\u5341")
newmOf10: List[String] = List("ten", "diez", "\u5341")
```

By checking reference equality (i.e., using `eq`)

```scala
mOf10 eq newmOf10
mOf10 eq List("ten", "diez", " ")
mOf10 == List("ten", "diez", " ")
```

```
res14_0: Boolean = true
res14_1: Boolean = false
res14_2: Boolean = true
```

we see that the above is one tree with both two versions on top of each other, leveraging immutability. Such data structures are called *persistent* because multiple versions can persist at the same time. In contrast, imperative data structures are *ephemeral* because only one version can exist at a time.

9.3 Abstract Syntax Trees (ASTs)

9.3.1 Mini Programming Languages

It is difficult to build an interpreter for any substantial language all at once. In this book, we will make some simplifications. We consider small subsets that isolate the essence of a language feature and incrementally examine more and more complex subsets.

For concreteness, let us consider variants of JavaScript as our primary object language of study, and we affectionately call the language that we implement in this course JavaScripty. However, note that the various subsets we consider could mimic just about any other language. In fact, this course has used other object languages in the past (e.g., a mini-OCaml called Lettuce, a mini-Scala called Smalla).

Because we do not yet have the mathematical tools to specify the semantics of a language, 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.

For example, let us consider the JavaScripty expression with +:

```
3 + 7 + 4.2
```

that results in `14.2`. That is,

When we have the tools to specify the semantics of a language, we may choose to make JavaScripty to have different semantics than JavaScript.

In actuality, there is not one language called JavaScript (officially, ECMAScript) but a set of closely related languages that may have slightly different semantics. In deciding how a JavaScripty program should behave, we consult a reference implementation (that we fix to be Google's open source V8 JavaScript Engine). We can run V8 through various engine interfaces (e.g., `node` and `deno`), and thus, we can write little test JavaScript programs and run it through to the engine to see how the test should behave.

9.3.2 Representing Abstract Syntax

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*.

While parsing seems like the place to start an implementation, the theory and implementation of parsers are surprising subtle. Instead, we can directly start our study of programming languages from abstract syntax assuming the JavaScripty input programs of interest come directly as abstract syntax trees.

9.3.2.1 JavaScripty: Number Literals and Addition

We represent abstract syntax trees in our meta/implementation language Scala using inductive, algebraic data types. Let us consider representing the most tiny JavaScripty language with number literals and + expressions. Here's one possible representation:

```scala
sealed trait Expr
case class N(n: Double) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr

val three = N(3)
val seven = N(7)
val four_point_two = N(4.2)
val three_plus_seven = Plus(three, seven)
val three_plus_seven_plus_four_point_two = Plus(three_plus_seven, four_point_two)
```

```
defined trait Expr
defined class N
defined class Plus
three: N = N(n = 3.0)
seven: N = N(n = 7.0)
four_point_two: N = N(n = 4.2)
three_plus_seven: Plus = Plus(e1 = N(n = 3.0), e2 = N(n = 7.0))
three_plus_seven_plus_four_point_two: Plus = Plus(
  e1 = Plus(e1 = N(n = 3.0), e2 = N(n = 7.0)),
  e2 = N(n = 4.2)
)
```

Here, we let a Scala value of type `Expr` (i.e., the meta language) represent a JavaScripty expression (i.e., the object language). A parser implementation (e.g., `parse: String => Expr` function) would take as input a JavaScripty expression (i.e., the object language) in concrete syntax (i.e., as a string) and convert into a Scala value of type `Expr` (i.e., in the meta language) as a tree (i.e., abstract syntax).

An `N` node represents a number literal where we represent JavaScripty numbers n as a Scala `Double`, and `Plus` is an AST node representing the JavaScripty $e_1 + e_2$.

Once we have a Scala value of type `Expr`, we can define functions that manipulate JavaScripty expressions. For example, we can define evaluation of JavaScripty expressions as an `eval` function in Scala:

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

eval( N(1.66) )
eval( Plus(N(2.1), N(3.5)) )
eval( three_plus_seven_plus_four_point_two )
```

```
defined function eval
res16_1: Double = 1.66
res16_2: Double = 5.6
res16_3: Double = 14.2
```

10 Lab: Recursion, Inductive Data Types, and Abstract Syntax Trees

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

Functional Programming Skills Representing data structures using algebraic data types.
Programming Languages Ideas Representing programs as abstract syntax.

Instructions

A version of project files for this lab resides in the public pppl-lab1 repository. Please follow separate instructions to get a private clone of this repository for your work.

You will be replacing `???` in the `Lab1.scala` file with solutions to the coding exercises described below. Make sure that you remove the `???` and replace it with the answer.

You may add additional tests to the `Lab1Spec.scala` file. In the `Lab1Spec.scala`, there is empty test class `Lab1StudentSpec` that you can use to separate your tests from the given tests in the `Lab1Spec` class. You are also likely to edit `Lab1.worksheet.sc` for any scratch work.

Single-file notebooks are convenient when experimenting with small bits of code, but they can become unwieldy when one needs a multiple-file project instead. In this case, we use standard build tools (e.g., `sbt` for Scala), IDEs (e.g., Visual Studio Code with Metals), and source control systems (e.g., `git` with GitHub). While it is almost overkill to use these standard software engineering tools for this lab, we get practice using these tools in the small.

If you like, you may use this notebook for experimentation. However, **please make sure your code is in `Lab1.scala`; this notebook will not graded.**

10.1 Recursion

10.1.1 Repeat String

Exercise 10.1. Write a recursive function `repeat` where `repeat(s, n)` returns a string with `n` copies of `s` concatenated together. For example, `repeat("a",3)` returns `"aaa"`. Implement by this function by direct recursion. Do not use any Scala library methods.

10.1.2 Square Root

In this exercise, we will implement the square root function. 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:

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

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 $\sqrt{c}$:

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

Exercise 10.2. First, implement a function `sqrtStep`

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

```
defined function sqrtStep
```

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

Exercise 10.3. Next, implement a function `sqrtN`

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

```
defined function sqrtN
```

that computes the nth approximation x_n from an initial guess x_0. You will want to call sqrtStep implemented in the previous part.

You need to implement this function using recursion and no mutable variables (i.e., vars)—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.

Exercise 10.4. Now, implement a function sqrtErr

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

```
defined function sqrtErr
```

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.

You need to implement this function using recursion, though it is useful to compare your recursive solution to one with a while loop.

10.2 Data Structures Review: Binary Search Trees

In this question, we 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 simplicity, we do 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 left child is l, data value is d, 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 classes:

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

```
defined trait Tree
defined object Empty
defined class Node
```

A `Tree` 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.

Exercise 10.5. The function `repOk`

```scala
def repOk(t: Tree): Boolean = {
  def check(t: Tree, min: Int, max: Int): Boolean = t match {
    case Empty => true
    case Node(l, d, r) => ???
  }
  check(t, Int.MinValue, Int.MaxValue)
}
```

```
defined function repOk
```

checks that an instance of `Tree` is valid binary search tree. In other words, it checks using a traversal of the tree the ordering invariant described above. This function is useful for testing your implementation.

Exercise 10.6. The function `insert`

```scala
def insert(t: Tree, n: Int): Tree = ???
```

```
defined function insert
```

inserts an integer into the binary search tree. Observe that the return type of `insert` is a `Tree`. 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.

Exercise 10.7. The function `deleteMin`

```scala
def deleteMin(t: Tree): (Tree, Int) = {
  require(t != Empty)
  (t: @unchecked) match {
    case Node(Empty, d, r) => (r, d)
    case Node(l, d, r) =>
      val (l1, m) = deleteMin(l)
      ???
  }
}
```

```
defined function deleteMin
```

deletes the smallest data element in the search tree (i.e., the leftmost 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.

Exercise 10.8. The function `delete`

```scala
def delete(t: Tree, n: Int): Tree = ???
```

```
defined function delete
```

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.

10.3 Interpreter: JavaScripty Calculator

In this question, we consider the arithmetic sub-language of JavaScripty (i.e., a basic calculator). We represent the abstract syntax for this sub-language in Scala using the following inductive data type:

```
defined trait Expr
defined class N
defined class Unary
defined class Binary
defined trait Uop
defined object Neg
defined trait Bop
```

Listing 10.1 Representing in Scala the abstract syntax of the arithmetic sub-language of JavaScripty (see `ast.scala`).

```
sealed trait Expr                                         // e ::=
case class N(n: Double) extends Expr                      //   n
case class Unary(uop: Uop, e1: Expr) extends Expr         // | uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr // | e1 bop e2

sealed trait Uop              // uop ::=
case object Neg extends Uop   //   -

sealed trait Bop              // bop ::=
case object Plus extends Bop  //   +
case object Minus extends Bop // | -
case object Times extends Bop // | *
case object Div extends Bop   // | /
```

```
defined object Plus
defined object Minus
defined object Times
defined object Div
```

In comments, we give a grammar that connects the abstract syntax with the concrete syntax of the language. We consider grammars in more detail subsequentlyin **?@sec-grammars**. For now, it is fine to ignore the concrete syntax or use your intuition for the connection. ow, given the inductive data type `Expr` defining the abstract syntax:

Exercise 10.9. Implement the `eval` function

```
def eval(e: Expr): Double = e match {
  case N(n) => ???
  case _   => ???
}
```

```
defined function eval
```

that evaluates the Scala representation of a JavaScripty expression `e` to the Scala double-precision floating point number corresponding to the Scala representation of the JavaScripty *value* of `e`. At this point, you have implemented your first language interpreter!

To go in more detail, consider a JavaScripty expression e, and imagine e to be concrete syntax. 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.

To see what a JavaScripty expression e should evaluate to, you may want to run e through a JavaScript interpreter to see what the value should be. For example,

```
3 + 4
```

```
1 / 7
```

```
6 * 4 - 2 + 10
```

Experiment in a Worksheet

Scala worksheets provide an interactive interface in the context of a multi-file project. A worksheet is a good place to start for experimenting with an implementation, whether on existing code or code that you are in the process of writing. A scratch worksheet `Lab1.worksheet.sc` is provided for you in the code repository.

To test and experiment with your `eval` function, you can write JavaScripty expressions directly in abstract syntax like above. You can also make use of a parser that is provided for you: it reads in a JavaScripty program-as-a-`String` and converts into an abstract syntax tree of type `Expr`.

For your convenience, we have also provided in the template `Lab1.scala` file, an overloaded `eval: String => Double` function that calls the provided parser and then delegates to your `eval: Expr => Double` function.

Test-Driven Development and Regression Testing

Once you have experimented enough in your worksheet to have some tests, it is useful to save those tests to run over-and-over again as you work on your implementation. The idea behind test-driven development is that we first write a test for what we expect our implementation to do. Initially, we expect our implementation to fail the test, and then we work on our implementation until the test succeeds. IDEs have features to support this workflow. While a test suite can never be exhaustive, we have provided a number of initial tests for you in `Lab1Spec.scala` to partially drive your test-driven development of the functions in this assignment.

Additional Notes

While you may not need them in this assignment, the `ast.scala` file also includes some basic helper functions for working with the AST, such as

```
def isValue(e: Expr): Boolean = e match {
  case N(_) => true
  case _ => false
}

val e_minus4_2 = N(-4.2)
isValue(e_minus4_2)

val e_neg_4_2 = Unary(Neg, N(4.2))
isValue(e_neg_4_2)
```

```
defined function isValue
e_minus4_2: N = N(n = -4.2)
res12_2: Boolean = true
e_neg_4_2: Unary = Unary(uop = Neg, e1 = N(n = 4.2))
res12_4: Boolean = false
```

the defines which expressions are values. In this case, literal number expressions `N(` n `)` are values where n is the meta-variable for JavaScripty numbers. We represent JavaScripty numbers in Scala with Scala values of type `Double`.

We also define functions to pretty-print, that is, convert abstract syntax to concrete syntax:

```
def prettyNumber(n: Double): String =
  if (n.isWhole) "%.0f" format n else n.toString

def pretty(v: Expr): String = {
  require(isValue(v))
  (v: @unchecked) match {
    case N(n) => prettyNumber(n)
  }
}

pretty(N(4.2))
pretty(N(10))
```

```
defined function prettyNumber
defined function pretty
res13_2: String = "4.2"
res13_3: String = "10"
```

We only define `pretty` for values, and we do not override the `toString` method so that the abstract syntax can be printed as-is.

```
e_minus4_2.toString
e_neg_4_2.toString
```

```
res14_0: String = "N(-4.2)"
res14_1: String = "Unary(Neg,N(4.2))"
```

The `@unchecked` annotation tells the Scala compiler that we know the pattern match is non-exhaustive syntactically, so we do not want to be warned about it. However, we see that our definition of `isValue` rules out the potential for a match error at run time (right?).

Submission

If you are a University of Colorado Boulder student, we use Gradescope for assignment submission. In summary,

- ☐ Create a private GitHub repository by clicking on a GitHub Classroom link from the corresponding Canvas assignment entry.
- ☐ Clone your private GitHub repository to your development environment (using the `<> Code` button on GitHub to get the repository URL).
- ☐ Work on this lab from your cloned repository. Use Git to save versions on GitHub (e.g., `git add`, `git commit`, `git push` on the command line or via VSCode).
- ☐ Submit to the corresponding Gradescope assignment entry for grading by choosing GitHub as the submission method.

You need to have a GitHub identity and must have your full name in your GitHub profile in case we need to associate you with your submissions.

Part III

Approaching a Programming Language

11 Concrete Syntax

We have studied programming languages like 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.

11.1 Concrete versus Abstract Syntax

Stated informally, the syntax of a language is concerned with the form of programs, such as, 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 consider defining the semantics of programming languages subsequentlyin **?@sec-operational-semantics**.

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 `{ ... }` 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 as trees (as in Section 9.3) used by language implementations and thus an important mental model for language implementers and language users. We draw out the relationship between concrete and abstract syntax here.

11.2 Context-Free Grammars

Formal language theory considers the study of describing sets of strings and the relative computational power of their recognizers called *automata*. We consider the formalisms from formal language theory only to the extent to be able to describe the syntax of a programming language. In particular, we introduce *grammars* that are formalisms for defining sets inductively.

A *formal language* $\mathcal{L}$ is a set of strings composed of characters drawn from some *alphabet* Σ (i.e., $\mathcal{L} \subseteq \Sigma^*$). 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*. In formal language theory, context-free languages are a proper superset of regular languages, and context-free grammars are the notational analogue of regular expressions.

A context-free grammar defines a language inductively and consists of *terminals*, *non-terminals*, 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. Non-terminals are defined via productions and conceptually recognize a sequence of symbols belonging to a sub-language. A production has the form

$$N ::= \alpha$$

where N is a non-terminal from the set of non-terminals $\mathcal{N}$ and α is a sequence of symbols (i.e., $\alpha \in (\Sigma \cup \mathcal{N})^*$). We write ε for the empty sequence. Note that $::=$ is sometimes written using different styles of arrows (e.g., $\rightarrow$).

A set of of productions with the same non-terminal, such as

$$\{N ::= \alpha_1, \dots, N ::= \alpha_n\}$$

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. This particular notation for context-free grammars is often called BNF (for Backus-Naur Form).

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

$$\begin{array}{rrcl} \text{integers} & i & ::= & \texttt{-}n \mid n \\ \text{numbers} & n & ::= & d \mid d\;n \\ \text{digits} & d & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \end{array}$$

with the alphabet

$$\Sigma \stackrel{\text{def}}{=} \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{-}\} \ .$$

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.

11.2.1 Deriving a Sentence in a Grammar

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

$$\alpha \quad \Longrightarrow \quad \beta$$

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 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 $\mathcal{L}(G)$ 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, $\mathcal{L}(G)$ is defined as follows:

$$\mathcal{L}(G) \stackrel{\text{def}}{=} \{\, s \mid s \in \Sigma^* \text{ and } S \Longrightarrow^* s \,\} \ .$$

11.2.2 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:

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

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 lexer is also sometimes called a *scanner*.

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:

$$\text{expressions} \quad \mathit{expr} \quad ::= \quad \mathit{num} \mid \mathit{expr}\, \texttt{+}\, \mathit{expr}$$

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.

As context-free languages include regular languages, one can also define parsers without lexers, typically called *lexer-less parsers* or *scanner-less parsers*.

11.2.3 Ambiguous Grammars

Consider the following arithmetic expression:

```
100 / 10 / 5
```

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

11.2.3.1 Associativity

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

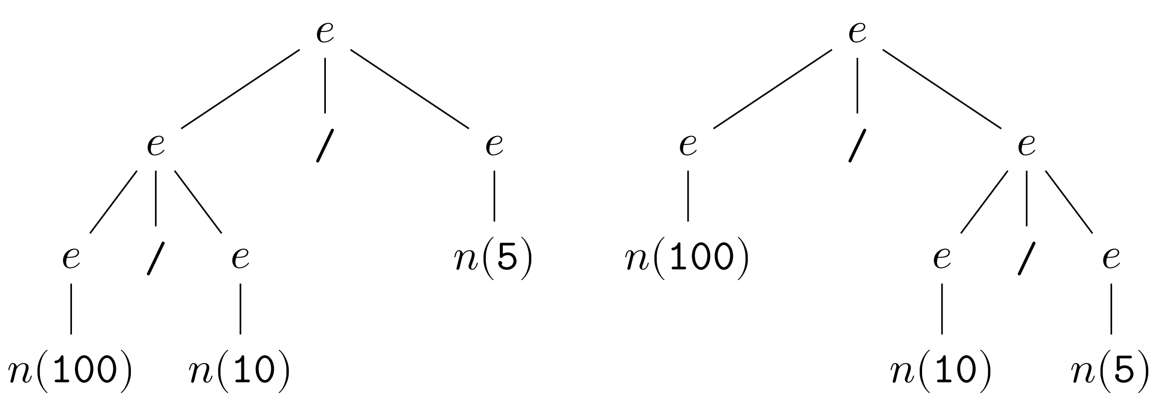
`100 / 10 / 5`

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

$$\text{expressions} \quad e \quad ::= \quad n \mid e \;\texttt{/}\; e$$

where n is the terminal for numbers.

We can diagram the two ways of reading the string '`100 / 10 / 5`' as shown in Figure 11.1a and Figure 11.1b where we write the lexemes for the n tokens in parentheses for clarity.



(a) The left-associative parse tree corresponding to (`100 / 10`) `/ 5`.

(b) The right-associative parse tree corresponding to `100 /` (`10 / 5`).

Figure 11.1: An ambiguous grammar is exhibited by two parse trees for a string in the language described by the grammar.

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. Note that the parentheses (...) in the captions are not part of sentences of the grammar but rather at the meta-level to convey the particular parse tree.

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 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 Table 11.1. One grammar is *left recursive*, that is, the production

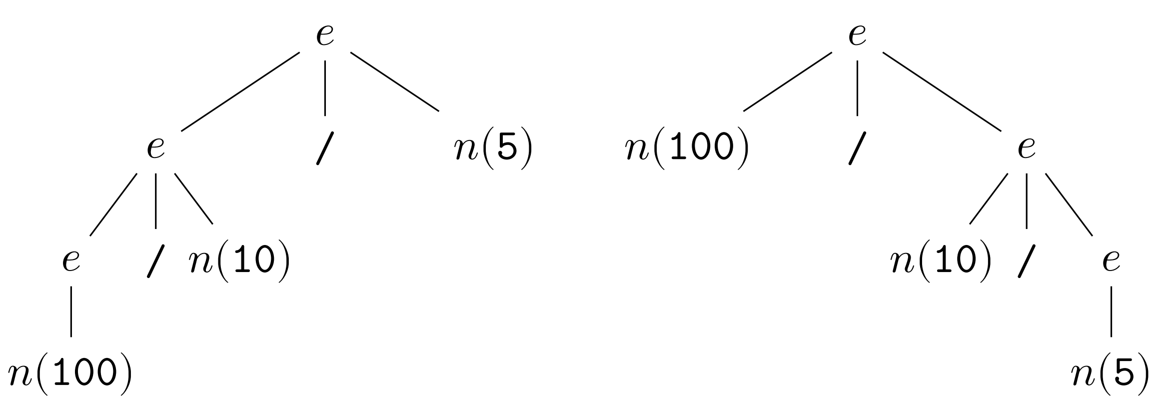
$$e ::= e \texttt{ / } n$$

is recursive only on the left of the binary operator token `/`. Analogously, we can write a *right recursive* grammar that accepts the same strings.

Table 11.1: Rewriting a grammar to eliminate ambiguity with respect to associativity.

Ambiguous	Unambiguous	
	Left-Recursive	Right-Recursive
$e ::= n \mid e \texttt{ / } e$	$e ::= n \mid e \texttt{ / } n$	$e ::= n \mid n \texttt{ / } e$

Intuitively, these grammars enforce a particular linearization of the possible parse trees: either to the left or to the right as shown in Figure 11.2. 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 11.2a. Analogously, a binary operator is *right associative* means expression trees involving that operator are linearized to the right, as in Figure 11.2b.



(a) The one possible parse tree for `100 / 10 / 5` corresponding to the left-recursive grammar in Table 11.1.

(b) The one possible parse tree for `100 / 10 / 5` corresponding to the right-recursive grammar in Table 11.1.

Figure 11.2: Grammars that enforce a particular associativity.

11.2.3.2 Precedence

A related syntactic issue appears when we consider multiple operators, such as the ambiguous grammar in Table 11.2.

Table 11.2: Rewriting a grammar to eliminate ambiguity and enforce a particular associativity and precedence. Both operators are left associative and the `/` operator has higher precedence than `-`.

Ambiguous	Unambiguous
expressions $e ::= n \mid e \texttt{ / } e \mid e \texttt{ - } e$	expressions $e ::= f \mid e \texttt{ - } f$ factors $f ::= n \mid f \texttt{ / } n$

For example, the string

`10 - 10 / 10`

has two parse trees corresponding to the following two readings:

`(10 - 10) / 10` or `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 Table 11.2. We layer the grammar by introducing a new non-terminal f that describes expressions with only `/` operators. 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, *lower precedence* means "binding looser" or "higher in the parse tree."

11.2.3.3 Syntactic and Semantic

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:

```
10 - 10 / 10
```

```
res0: Int = 9
```

12 Abstract Syntax and Parsing

Recall from Chapter 11 that the concrete syntax of a programming language is a set of *strings* (i.e., sequences of characters in an alphabet). A grammar is an inductive definition for describing a inductive set of strings.

A grammar is ambiguous when there exists at least one sentence in the language that can be generated by the grammar in more than one way. What this means is that the string has multiple distinct parse trees or derivations, leading to different interpretations of the program's tree structure.

The *abstract syntax* of a programming language makes explicit a program's *tree* structure (sometimes also called *terms*).

A *parser* converts concrete syntax into abstract syntax, which has deal with ambiguity. A common (though not only) source of ambiguity are infix operators, which can be disambiguated by making explicit associativity and precedence.

12.1 Abstract Syntax

Consider again the grammar of expressions involving the `/` and `-` operators in Table 11.2, with subscripts to make explicit the instances of the symbols:

$$\text{expressions} \quad e \quad ::= \quad n \mid e_1 \texttt{ / } e_2 \mid e_1 \texttt{ - } e_2 \tag{12.1}$$

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

```scala
sealed trait Expr                                   // e ::=
case class N(n: Int) extends Expr                   //   n
case class Divide(e1: Expr, e2: Expr) extends Expr // | e1 / e2
case class Minus(e1: Expr, e2: Expr) extends Expr  // | e1 - e2
```

```
defined trait Expr
defined class N
defined class Divide
defined class Minus
```

We define a new type `Expr` (i.e., a `trait`). 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 (Equation 12.1) to use these constructor names in each production, we get the following:

$$\begin{array}{rrcl} \text{expressions } \texttt{Expr} & e & ::= & \texttt{N(}n\texttt{)} \\ & & \mid & \texttt{Divide(}e_1\texttt{,}\ e_2\texttt{)} \\ & & \mid & \texttt{Minus(}e_1\texttt{,}\ e_2\texttt{)} \\ \text{integers} & n & & \end{array} \tag{12.2}$$

An example sentence in this language is

```
Minus(N(10), Divide(N(10), N(10)))
```

```
res1: Minus = Minus(e1 = N(n = 10), e2 = Divide(e1 = N(n = 10), e2 = N(n = 10)))
```

which corresponds to the following sentence in the first grammar:

`10 - 10 / 10`

Observe that a different sentence in the second grammar (Equation 12.2)

```
Divide(Minus(N(10), N(10)), N(10))
```

```
res2: Divide = Divide(
  e1 = Minus(e1 = N(n = 10), e2 = N(n = 10)),
  e2 = N(n = 10)
)
```

also corresponds to the sentence `10 - 10 / 10` in the first grammar (Equation 12.1) (with a different parse tree). Thus, while the first grammar is ambiguous, the second one is unambiguous.

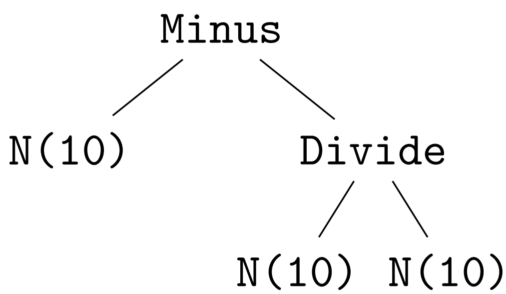
In a language implementation, we do not want to be constantly worrying about the "grouping" or parsing of a string (i.e., resolving ambiguity), so we prefer to work with *terms* in this second grammar. We call this second grammar, *abstract syntax*, where the tree structure is evident. Observe that parentheses around each sub-expression avoids ambiguity.

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

```scala
Minus(N(10), Divide(N(10), N(10)))
```

```
res3: Minus = Minus(e1 = N(n = 10), e2 = Divide(e1 = N(n = 10), e2 = N(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 (i.e., trees).

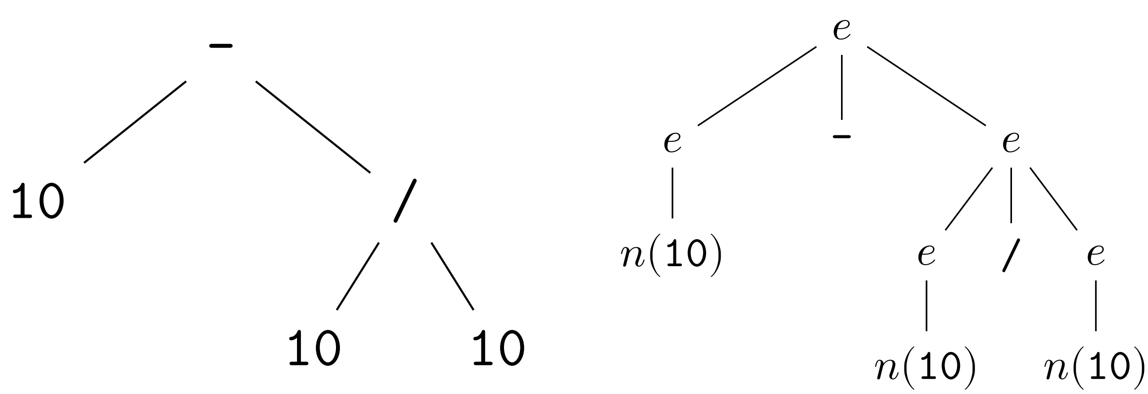
Because the concrete syntax is more concise visually and human friendly, it is standard practice to give (ambiguous) grammars like the first grammar above (Equation 12.1) and treat them as the corresponding abstract syntax specification given in the second grammar (Equation 12.2). In other words, we 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 in Figure 12.1a.

12.2 Parsing

Parsing is a large topic in terms of both deep theory and innovative tools. Thus, the theoretical and practical aspects are covered in more depth in theory of computation and compiler construction courses, respectively.

We use parsers daily, translating text that we can read and write into data structures tha machines can understand. The range of kinds of parsers is also enormous: from simple regular expression-based pattern matchers that run inside packet filters on the Internet to complex natural language parsers that take in extract structure out of natural language sentences. As such, parsing is arguably one of most successful applications of theoretical computer science into practice.

Parsers for programming languages are usually somewhere in between: they have inductive structure (e.g., matching parentheses) that require more than regular-expression parsers but not as complicated as natural language with all its inherent ambiguities and context-sensitivity



(a) An abstract syntax tree using concrete syntax operators.

(b) The corresponding parse tree using the ambiguous grammar in Equation 12.1.

Figure 12.1: Consider the abstract syntax tree `Minus(N(10), Divide(N(10), N(10)))`. We show an abstract syntax tree and the corresponding parse tree with the ambiguous grammar shown in Equation 12.1.

(though the syntax of real-world programming languages do sometimes extend beyond context-free).

12.2.1 Top-Down Parsing

There are numerous parsing algorithms with different tradeoffs that are better studied in a compiler construction course. However, all tools or libraries for creating parsers for programming languages generally involve specifying a BNF-like grammar.

Building parsers can get complex quickly even with a parsing library, so we focus here simply on restricted uses of such libraries to build intuitions for context-free grammars.

We consider a kind of library for parsers called *combinator parsers*. A *combinator* is a kind of higher-order function (e.g., a function that takes another function as input). A combinator-parsing library is one where the user specifies the grammar simply as calls to the library to build a parser from other parsers, along with callback functions. Combinator parsing libraries generally help us implement some form of *recursive-descent parsing*, which we can think of simply automating the top-down leftmost parsing derivation and trying productions left-to-right until finding a prefix match.

Thus, they work best (1) when the grammar is unambiguous so that the top-down derivation is deterministic and (2) when there is no left recursion so that each top-down leftmost derivation step makes progress consuming some prefix of the input string.

Scala Parser Combinators Library

Run the following cell to load the Scala Parser Combinators library.

Listing 12.1 `scala.util.parsing.combinator._`

```
import $ivy.`org.scala-lang.modules::scala-parser-combinators:2.4.0`
```

```
import $ivy.$
```

Let us consider our object language JavaScripty with number literals and addition from our abstract syntax tree discussion (Section 9.3.2.1).

```
sealed trait Expr                                  // e ::=
case class N(n: Double) extends Expr               //   n
case class Plus(e1: Expr, e2: Expr) extends Expr // | e1 + e2
```

```
defined trait Expr
defined class N
defined class Plus
```

We give a grammar corresponding to the abstract syntax with concrete syntax operators:

$$\text{expressions} \quad e \quad ::= \quad n \mid e_1 \,\texttt{+}\, e_2 \tag{12.3}$$

Note that we gave this same grammar as comments in the Scala code above.

Observe that this grammar given above (Equation 12.3) for JavaScripty with number literals and + expressions is ambiguous. Recall that an ambiguous grammar means there is a sentence in the language described by the grammar with more than one parse tree (or equivalently, more than one parsing derivation). For example, the sentence `1 + 2 + 3` (i.e., $n(\texttt{1})$ + $n(\texttt{2})$ + $n(\texttt{3})$ with lexical analysis) has two parse trees with this grammar. It also has left recursion in that the production

$$e ::= e \,\texttt{+}\, e$$

has the e non-terminal expanding to a sentential form with itself on the left.

12.2.1.1 Left Recursion and Top-Down Parsing

From Section 11.2.3.1, we know how to refactor the grammar to disambiguate for associativity. However, to make the infix binary + operator left associative, the resulting grammar has still left recursion.

> **!** Left Recursion and Top-Down Parsing
>
> To use simple recursive-descent parsing and the Scala Parser Combinator library, we cannot use a grammar with left recursion.

Intuitively, left recursion causes an infinite recursion in a simple recursive descent parser because we do not know how far we have to "lookahead" to choose between expanding with a recursive production or a base case production.

$$
\begin{array}{rcl}
e & \Longrightarrow & e \mathbin{\texttt{+}} e \\
 & \Longrightarrow & e \mathbin{\texttt{+}} e \mathbin{\texttt{+}} e \\
 & \Longrightarrow & e \mathbin{\texttt{+}} e \mathbin{\texttt{+}} e \mathbin{\texttt{+}} e \\
 & \Longrightarrow & \ldots
\end{array}
$$

12.2.1.2 Restricting the Concrete Syntax

We consider subsequentlyin **?@sec-ebnf** how we can parse left-associative operators with a recursive-descent parser. Here, we simply restrict the concrete syntax to simplify parsing (i.e., we "cheat" by changing the concrete syntax of the language). For example, for JavaScripty with number literals and addition, we consider the following grammar for the concrete syntax:

$$
\begin{array}{rrcl}
\text{terms} & t & ::= & n \mid \texttt{(}\ e\ \texttt{)} \\
\text{expressions} & e & ::= & t_1 \mathbin{\texttt{+}} t_2
\end{array}
\tag{12.4}
$$

with t as the start symbol. Take special note that the parentheses here `( )` are part of the concrete syntax of the object language. Compare and contrast this restricted, unambiguous grammar with the ambiguous grammar corresponding directly to the abstract syntax (Equation 12.3). Observe that it is similar to the ambiguous grammar in Equation 12.3 but *does not* accept the same language. Essentially, we have eliminated ambiguity by "forcing" the JavaScripty programmer to write enough parentheses `( )` to state what "grouping" they want. But also note that there is no left recursion in this restricted grammar.

A t here is what's called an *s-expression* (restricted to these particular terminal symbols). S-expressions are commonly used as a serialization format because it is easy to parse. They are used as the concrete syntax for Lisp and Lisp-derived programming languages (e.g., Scheme, Racket), though all operators are written in prefix notation.

12.2.1.3 Implementing a Parser

To connect this restricted grammar to a parser implementation, let us give alternative names to the non-terminal symbols:

$$
\begin{array}{rrcl}
\text{terms} & t, term & ::= & num \mid \texttt{(}\ expr\ \texttt{)} \\
\text{expressions} & e, expr & ::= & term\ \texttt{+}\ term \\
\text{numbers} & n, num & &
\end{array}
\tag{12.5}
$$

We can now implement the restricted grammar (Equation 12.4) directly using the Scala Parsing Combinator Library:

```
object ExprParser extends scala.util.parsing.combinator.RegexParsers {
  def term: Parser[Expr] =
    num ^^ { (n: String) => N(n.toDouble) } |
    "(" ~ expr ~ ")" ^^ { case _ ~ e ~ _ => e }

  def expr: Parser[Expr] =
    term ~ "+" ~ term ^^ { case e1 ~ _ ~ e2 => Plus(e1, e2) }

  def num: Parser[String] =
    """-?(\d+(\.\d*)?|\d*\.\d+)([eE][+-]?\d+)?""".r

  def parse(str: String): Either[String, Expr] = parseAll(term, str) match {
    case Success(e, _) => Right(e)
    case Failure(msg, _) => Left(s"Failure: $msg")
    case Error(msg, _) => Left(s"Error: $msg")
  }
}
```

```
defined object ExprParser
```

First, observe that we have translated each non-terminal *term*, *expr*, *num* into a method `term`, `expr`, and `num` that returns a value of type `Parser[A]` for some type `A`, respectively. This structure is indicative of a recursive-descent parser where we define a set of mutually-recursive parsing methods—one for each non-terminal in the grammar of interest.

Then, the `scala.util.parsing.combinator.RegexParsers` trait provides a number of library methods that we use to define the `expr`, `term`, and `num` methods. These methods, like `|` and `~`, make it look like we are writing a BNF grammar. We see that the `|` method calls separate grammar productions and the `~` method calls correspond to sequencing symbols on the right-hand side of a production. Ignore the `^^` method calls with the function arguments in `{ ... }` on the right for the moment.

The parameter `A` to the `Parser[A]` type is the result type of the parser. For example, the method `term: Parser[Expr]` returns a parser whose result type is an abstract syntax tree value `Expr`, while the `num: Parser[String]` returns a parser whose result type is a `String` value.

12.2.1.4 *term* ::= *num*

Let us focus on the implementation on the `term` method and the first part of the implementation:

```
    num ^^ { (n: String) => N(n.toDouble) } |
```

corresponding to the first production:

$$term ::= num$$

To implement this production, this code calls the `num` method. The `num` method returns a `Parser[String]` that recognizes its input as a number, returning the matched `String`. The `^^` library method call enables specifying a *semantic action* that translates a `Parser[String]` to a `Parser[Expr]` using the function

```
            { (n: String) => N(n.toDouble) }
```

of type `String => Expr`. In this case, the `String` given in `n` is converted to a `Double` using Scala's `toDouble` method and then passed to the `N` constructor defined above (that is an `Expr`).

We define an "interface" method `parse: String => Either[String, Expr]` that does the parsing by calling `parseAll` from the library, passing `expr` as the start symbol and `str` as the input string to parse.

```
ExprParser.parse("1")
ExprParser.parse("<should not parse>")
```

```
res7_0: Either[String, Expr] = Right(value = N(n = 1.0))
res7_1: Either[String, Expr] = Left(
  value = "Failure: '(' expected but '<' found"
)
```

We have defined the `parse` to return an `Either[String, Expr]`. A `Either` is used similarly to an `Option` except when we want to have something more in the `None` case. In this case, we either give a error message using the `Left` case or return resulting `Expr` using the `Right` case. It is a standard programming convention that the `Left` case of an `Either` is generally used for the "error case," while `Right` is used for "success."

12.2.1.5 *term* ::= **(** *expr* **)** and *expr* ::= *term* **+** *term*

Now focus on the second part of implementation of the `term` implementation:

```
    "(" ~ expr ~ ")" ^^ { case _ ~ e ~ _ => e }
```

corresponding to the second production:

$$term ::= \texttt{(}\ expr\ \texttt{)}$$

The `~` method produces a `Parser[A ~ B]` sequencing a `Parser[A]` and a `Parser[B]`. In this case, we have a a `Parser[String ~ Expr ~ String]` that we translate to a `Parser[Expr]` using this function:

```
{ case _ ~ e ~ _ => e }
```

Note that the `~` type constructor is a `case class` defined by the Scala Combinator Parser Library that we can see as simply a custom tuple type.

Also note that the `"("` and `")"` expressions in the above have type `Parser[String]`. The Scala Combinator Parsing Library (specifically in `RegexParsers`) defines implicit conversions that creates a `Parser[String]` that accepts a single `String`. Thus, this more specific pattern matching in the semantic action function would be behave the same:

```
    "(" ~ expr ~ ")" ^^ { case "(" ~ e ~ ")" => e }
```

The `expr` method definition is now relatively straightforward:

```
  def expr: Parser[Expr] =
    term ~ "+" ~ term ^^ { case e1 ~ _ ~ e2 => Plus(e1, e2) }
```

in that it creates a `Plus` node out of two `Expr`s.

We can now test out our parser with + expressions in concrete syntax:

```scala
ExprParser.parse("((1 + 2) + 3)")
ExprParser.parse("(1 + (2 + 3))")
```

```
res8_0: Either[String, Expr] = Right(
  value = Plus(e1 = Plus(e1 = N(n = 1.0), e2 = N(n = 2.0)), e2 = N(n = 3.0))
)
res8_1: Either[String, Expr] = Right(
  value = Plus(e1 = N(n = 1.0), e2 = Plus(e1 = N(n = 2.0), e2 = N(n = 3.0)))
)
```

And we can see that a + expression without parentheses fails to parse (as expected):

```scala
ExprParser.parse("1 + 2 + 3")
```

```
res9: Either[String, Expr] = Left(value = "Failure: end of input expected")
```

12.2.1.6 *num*

In the parser grammar from Equation 12.5, we consider *num* a terminal and never specified what sentences are in the language of *num* (i.e., $\mathcal{L}(num)$). In this implementation, we use the following regular expression:

```scala
  def num: Parser[String] =
    """-?(\d+(\.\d*)?|\d*\.\d+)([eE][+-]?\d+)?""".r
```

(i.e., a value of type `Regex` in Scala). The `RegexParsers` trait also defines an implicit conversion that creates a `Parser[String]` out of a `Regex` value, which is what we use here.

The following strings are matched by this regular expression:

```scala
ExprParser.parse("1")
ExprParser.parse("-1")
ExprParser.parse(".1")
ExprParser.parse("2e2")
ExprParser.parse("2e-10")
ExprParser.parse("2e+10")
ExprParser.parse("2E10")
```

```
res10_0: Either[String, Expr] = Right(value = N(n = 1.0))
res10_1: Either[String, Expr] = Right(value = N(n = -1.0))
res10_2: Either[String, Expr] = Right(value = N(n = 0.1))
res10_3: Either[String, Expr] = Right(value = N(n = 200.0))
res10_4: Either[String, Expr] = Right(value = N(n = 2.0E-10))
res10_5: Either[String, Expr] = Right(value = N(n = 2.0E10))
res10_6: Either[String, Expr] = Right(value = N(n = 2.0E10))
```

13 Exercise: Syntax

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

- working with abstract syntax trees;
- how grammars are used to specify the syntax of programming languages; and
- the distinction between concrete and abstract syntax.

Instructions

This assignment asks you to write Scala code. There are restrictions associated with how you can solve these problems. Please pay careful heed to those. If you are unsure, ask the course staff.

Note that `???` indicates that there is a missing function or code fragment that needs to be filled in. Make sure that you remove the `???` and replace it with the answer.

Use the test cases provided to test your implementations. You are also encouraged to write your own test cases to help debug your work. However, please delete any extra cells you may have created lest they break an autograder.

Imports

```
import $ivy.$                                   , org.scalatest._, events._, flatspec._
```

```
defined function report
defined function assertPassed
defined function passed
defined function test
```

```
import $ivy.$
```

Listing 13.1 `org.scalatest._`

```
// Run this cell FIRST before testing.
import $ivy.`org.scalatest::scalatest:3.2.19`, org.scalatest._, events._, flatspec._
def report(suite: Suite): Unit = suite.execute(stats = true)
def assertPassed(suite: Suite): Unit =
  suite.run(None, Args(new Reporter {
    def apply(e: Event) = e match {
      case e @ (_: TestFailed) => assert(false, s"${e.message} (${e.testName})")
      case _ => ()
    }
  }))
def passed(points: Int): Unit = {
  require(points >=0)
  if (points == 1) println("***   Tests Passed (1 point) ***")
  else println(s"***   Tests Passed ($points points) ***")
}
def test(suite: Suite, points: Int): Unit = {
  report(suite)
  assertPassed(suite)
  passed(points)
}
```

Listing 13.2 `scala.util.parsing.combinator._`

```
// Run this cell FIRST before building your parser.
import $ivy.`org.scala-lang.modules::scala-parser-combinators:2.4.0`
```

13.1 Abstract Syntax Trees

Let us consider the (abstract) syntax of a language of Boolean expressions:

$$
\begin{array}{lrcl}
\text{Boolean expressions} & e & ::= & x \mid \neg e_1 \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \\
\text{variables} & x & &
\end{array}
$$

13.1.1 Defining an Inductive Data Type

We can write explicitly the above grammar with abstract syntax tree nodes:

$$
\begin{array}{llll}
\text{Boolean expressions } \texttt{BExpr} & e & ::= & \texttt{Var(}x\texttt{)} \\
& & \mid & \texttt{Not(}e_1\texttt{)} \\
& & \mid & \texttt{And(}e_1\texttt{, }e_2\texttt{)} \\
& & \mid & \texttt{Or(}e_1\texttt{, }e_2\texttt{)}
\end{array}
$$

Exercise 13.1 (5 points)**.** Define an inductive data type `BExpr` in Scala following the above grammar. Use the the names given above for the constructors, and use the Scala type `String` to represent variable names.

Edit this cell:

???

Tests

13.1.2 Converting to Negation Normal Form

Exercise 13.2 (10 points)**.** Write a function `nnf` to convert Boolean formulas from the above grammar to their Negation Normal Form (NNF). A Boolean formula is said to be in NNF iff the negation operator $\neg$ (i.e., `Not`) is only applied to the variables. For example, the following formula is in NNF because negation is only applied to A or B, or both of which are variables:

$$(A \land \neg B) \lor (B \land \neg A) \;.$$

On the other hand, the following formula is not in NNF as negation is applied to a complex expression:

$$\neg(A \lor \neg B) \land (\neg B \lor C) \;.$$

The negation normal form of the above formula is as follows:

$$(\neg A \land \neg B) \land (\neg B \lor C) \;.$$

Edit this cell:

???

Notes

In general, a formula can be converted to NNF by applying three rules:

Rule 1 Double negation is cancelled: $\neg(\neg e) = e$ for any Boolean formula e.
Rule 2 De Morgan's Law for conjunction: $\neg(e_1 \wedge e_2) = \neg e_1 \vee \neg e_2$ for any two Boolean formulas e_1 and e_2.
Rule 3 De Morgan's Law for disjunction: $\neg(e_1 \vee e_2) = \neg e_1 \wedge \neg e_2$ for any two Boolean formulas e_1 and e_2.

The `nnf` function will have a case for each of the above rule. In addition, the function will also need to handle the following cases:

- A variable x or its negation $\neg x$ is already in NNF.
- An expression $e_1 \wedge e_2$ is in NNF iff e_1 and e_2 are in NNF.
- An expression $e_1 \vee e_2$ is in NNF iff e_1 and e_2 are in NNF.

If you handle all the 7 cases described above, then your `nnf` function will likely be correct, though of course, you will want to test it.

Tests

13.1.3 Substitution

Exercise 13.3 (5 points)**.** Write a function `subst` that substitutes *in* a given Boolean expression *with* another expression *for* a given variable name. For example, substituting in

$$\neg(A \vee B) \wedge (\neg B \vee C)$$

with D for B yields

$$\neg(A \vee D) \wedge (\neg D \vee C) \; .$$

Edit this cell:

???

Tests

13.2 Concrete Syntax

13.2.1 Precedence Detective

Consider the Scala binary expressions

$$\begin{array}{rcl} \text{expressions} & e & ::= & n \mid e_1 \texttt{ - } e_2 \mid e_1 \texttt{ << } e_2 \\ \text{integers} & n & & \end{array}$$

Exercise 13.4 (5 points)**.** Write Scala expressions to determine if `-` has higher precedence than `<<` or vice versa. To do, write an expression bound to `e_no_parens` that uses no parentheses. Then, bind to `e_higher_-` the expression that adds parentheses to the `e_no_parens` expression corresponding to the case if `-` has higher precedence than `<<`, and bind to `e_higher_<<` the expression adds parentheses corresponding to the other case.

Edit this cell:

Make sure that you are checking for precedence and not for left or right associativity.

???

val e_higher_- = ???

val e_higher_« = ???

```
- Partial implementation, but there are significant issues in the Scala expressions.
- Major issues:
  - Doesn't check for operator precedence between `-` and `<<`.
  - Incorrect or missing parentheses in `e_higher_-` or `e_higher_<<`.
  - Fail most test cases.

**Approaching (A)**
```

val e_no_parens = ???

val e_higher_- = ???

val e_higher_« = ???

```
- Mostly correct implementation, but might fail some test cases or have minor errors.
- Attempts to check for relative precedence of `-` and `<<` with parentheses but might mispla

**Proficient (P) or Exceeding (E)**
```

val e_no_parens = ???

val e_higher_- = ???

val e_higher_« = ???

```
- Fully correct implementation of all expressions.
- Correctly binds `e_no_parens`, `e_higher_-`, and `e_higher_<<` to appropriately test for pr
- Passes all test cases, including:
  - Ensuring that e_no_parens == e_higher_- or e_no_parens == e_higher_<<.
  - Ensuring that e_higher_- != e_higher_<<.

// END SOLUTION
:::

#### Assertions {.unnumbered}

:::: {.content-hidden when-format="pdf"}

::: {.cell nbgrader='{"grade":true,"grade_id":"detective_tests","locked":true,"points":5,"sch
``` {.scala .cell-code}
assert(e_no_parens == e_higher_- || e_no_parens == e_higher_<<)
assert(e_higher_- != e_higher_<<)
passed(5)
```

```
cmd4.sc:1: not found: value e_no_parens
val res4_0 = assert(e_no_parens == e_higher_- || e_no_parens == e_higher_<<)
                    ^cmd4.sc:1: not found: value e_higher_-
val res4_0 = assert(e_no_parens == e_higher_- || e_no_parens == e_higher_<<)
                                   ^cmd4.sc:1: not found: value e_no_parens
val res4_0 = assert(e_no_parens == e_higher_- || e_no_parens == e_higher_<<)
                                                 ^cmd4.sc:1: not found: value e_higher_<<
val res4_0 = assert(e_no_parens == e_higher_- || e_no_parens == e_higher_<<)
                                                                ^cmd4.sc:2: not found: value
val res4_1 = assert(e_higher_- != e_higher_<<)
                    ^cmd4.sc:2: not found: value e_higher_<<
```
```

```
val res4_1 = assert(e_higher_- != e_higher_<<)
                                  ^Compilation Failed
```

```
:
Compilation Failed
```

::::

Explanation

Exercise 13.5 (5 points)**.** Explain how you arrived at the relative precedence of `-` and `<<` based on the output that you saw in the Scala interpreter.

Edit this cell:

???

13.3 Parse Trees

Consider the following grammar:

$$\begin{array}{rrcl} \text{expressions} & e & ::= & x \mid e \; \texttt{?} \; e \; \texttt{:} \; e \\ \text{variables} & x & ::= & \texttt{a} \mid \texttt{b} \end{array} \tag{13.1}$$

Consider the following data type for representing parse trees:

```scala
sealed trait ParseTree
case class Leaf(term: String) extends ParseTree
case class Node(nonterm: String, children: List[ParseTree]) extends ParseTree
```

```
defined trait ParseTree
defined class Leaf
defined class Node
```

Observe that a `ParseTree` is just an n-ary tree containing `String`s. A `Leaf` is a terminal containing the lexemes (i.e., letters from the alphabet), while a `Node` is represents a non-terminal with a `String` for the name of the non-terminal a list of `children` `ParseTree`s.

For example, the following are `ParseTree`s for this grammar (Equation 13.1):

```scala
val p1 = Node("x", Leaf("a") :: Nil)
val p2 = Node("e", Node("x", Leaf("a") :: Nil) :: Nil)
```

```
p1: Node = Node(nonterm = "x", children = List(Leaf(term = "a")))
p2: Node = Node(
  nonterm = "e",
  children = List(Node(nonterm = "x", children = List(Leaf(term = "a"))))
)
```

We provide a function that pretty-prints a `ParseTree` for *this grammar* (Equation 13.1).

```scala
def pretty(t: ParseTree): Option[String] = {
  val alphabet = Set("a", "b", "?", ":")
  t match {
    // e ::= x
    case Node("e", (x @ Node("x", _)) :: Nil) => pretty(x)
    // e ::= e ? e : e
    case Node("e", (e1 @ Node("e", _)) :: Leaf("?") :: (e2 @ Node("e", _)) :: Leaf(":") :: (e
      for { s1 <- pretty(e1); s2 <- pretty(e2); s3 <- pretty(e3) }
      yield s"$s1 ? $s2 : $s3"
    // x ::= a
    case Node("x", Leaf("a") :: Nil) => Some("a")
    // x ::= b
    case Node("x", Leaf("b") :: Nil) => Some("b")
    // failure
    case _ => None
  }
}

pretty(p1)
pretty(p2)
```

```
defined function pretty
res6_1: Option[String] = Some(value = "a")
res6_2: Option[String] = Some(value = "a")
```

Since it is possible to have `ParseTree`s that are not recognized by this grammar, the `pretty` function has return type `Option[String]`. Calling `pretty` on `ParseTree`s that are not in this grammar return `None`:

```scala
pretty(Leaf("a"))
pretty(Node("x", Leaf("c") :: Nil))
pretty(Node("y", Leaf("a") :: Nil))
```

```
res7_0: Option[String] = None
res7_1: Option[String] = None
res7_2: Option[String] = None
```

Note that the `pretty` function makes use of Scala's **`for-yield`** expressions, which you do not need to understand for this exercise.

Exercise

Exercise 13.6 (5 points)**.** Give a parse tree for the sentence in the grammar `a ? b : a`.

Edit this cell:

???

Assertion

Exercise

Exercise 13.7 (10 points)**.** Show that this grammar (Equation 13.1) is ambiguous by giving two parse trees for the sentence in the grammar `a ? b : a ? b : a`.

Edit this cell:

???

Assertions

13.4 Defining Grammars

In this question, we define a BNF grammar for *floating-point numbers* that are made up of a fraction (e.g., `5.6` or `3.123` or `-2.5`) followed by an optional exponent (e.g., `E10` or `E-10`).

More precisely for this exercise, our floating-point numbers

- ☐ must have a decimal point,

- ☐ do not have leading zeros,
- ☐ can have any number of trailing zeros,
- ☐ non-zero exponents if it exists,
- ☐ must have non-zero fraction to have an exponent, and
- ☐ cannot have a '-' in front of a zero number; also,
- ☐ the exponent cannot have leading zeros.

The exponent, if it exists, is the letter `E` followed by an integer. For example, the following are floating-point numbers: `0.0`, `3.5E3`, `3.123E30`, `-2.5E2`, `-2.5E-2`, `3.50`, and `3.01E2`.

The following are examples of strings that are *not* floating-point numbers by our definition: `0`, `-0.0`, `3.E3`, `E3`, `3.0E4.5`, and `4E4`.

For this exercise, let us assume that the tokens are characters in the following alphabet Σ:

$\Sigma \stackrel{\text{def}}{=} \{$ `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `E`, `-`, `.` $\}$

Exercise 13.8 (10 points)**.** Translate a grammar into a parser `FloatParser.float: Parser[String]` using the Scala Parsing Combinator Library that recognizes the language of floating-point numbers described above. Since we do not care about the correctness of the grammar, we let parse result simply be the input string.

We suggest that you first use the scratch cell below to give a grammar in BNF notation. Your grammar should be completely defined using the alphabet above as the terminals (i.e., it should not count on a non-terminal that it does not itself define).

SCRATCH CELL

Edit this cell:

???

Notes

- The `success[A](a: A): Parser[A]` method corresponds to an ε production in BNF. It returns a parser that is successful without consuming any input yielding the parse result `a`. If you use it in your parser, make sure it is the *last* production for the non-terminal.
- We provide some helper functions `concat2: String ~ String => String`, `concat3: String ~ String` etc. that simply concatenate their input strings together. These helper functions are the only semantic actions you need.
- We have provided some basic parsers `sign`, `anyOneToNine`, `digit`, `zeroOrMoreDigits` that you may use if you like.
- You may edit the `parse` interface function that we have provided to start from a different non-terminal while you're developing, but make sure it is `float` in the end and that your starting non-terminal is `float`. Alternatively, you may add additional such interface functions with a different name for your testing.

Tests

14 Static Scoping

Let us consider our object language JavaScripty with number literals and addition from our abstract syntax tree discussion (Section 9.3.2.1). Now, let's extend it with variable uses and binding.

What makes a language go beyond what we might consider a calculator language is adding variable uses and binding. In the following, we show variable binding `const` in JavaScript, `let` in OCaml, and `val` in Scala. We have intentionally aligned them so that their syntactic differences are superficial (i.e., essentially keywords that introduce binding).

14.1 JavaScripty (JavaScript)

$$
\begin{array}{lrcll}
\text{expressions} & e & ::= & n \mid e_1 \texttt{+} e_2 & \text{number literals and addition} \\
 & & \mid & x \mid \textbf{const}\ x \texttt{=} e_1 \texttt{;}\ e_2 & \text{variable uses and binding}
\end{array}
\tag{14.1}
$$

14.2 Lettuce (OCaml)

$$
\begin{array}{lrcll}
\text{expressions} & e & ::= & n \mid e_1 \texttt{+} e_2 & \text{number literals and addition} \\
 & & \mid & x \mid \textbf{let}\ x \texttt{=} e_1\ \textbf{in}\ e_2 & \text{variable uses and binding}
\end{array}
$$

14.3 Smalla (Scala)

$$
\begin{array}{lrcll}
\text{expressions} & e & ::= & n \mid e_1 \texttt{+} e_2 & \text{number literals and addition} \\
 & & \mid & x \mid \textbf{val}\ x \texttt{=} e_1 \texttt{;}\ e_2 & \text{variable uses and binding}
\end{array}
$$

14.4 JavaScripty: Variable Uses and Binding

Let us consider extending the representation of the abstract syntax of JavaScripty in Scala with variable uses and bindings:

```scala
sealed trait Expr                                                  // e ::=
case class N(n: Double) extends Expr                               //   n
case class Plus(e1: Expr, e2: Expr) extends Expr                   // | e1 + e2
case class Var(x: String) extends Expr                             // | x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr // | const x = e1; e2
```

```
defined trait Expr
defined class N
defined class Plus
defined class Var
defined class ConstDecl
```

As we discuss in Section 4.2 regarding scoping in Scala, the scope of a variable binding is the part of the program where that variable can be used and refers to that particular binding. Static scoping is where the scope of a variable can be determined by looking directly at the source code.

We have structured our abstract syntax so that the scope of a variable binding is apparent. In particular, the `ConstDecl(` x `,` e_1 `,` e_2 `)` is the AST node that represents binding a JavaScripty variable x (whose name is stored in Scala as `x: String`) to the JavaScripty value obtained by evaluating expression e_1 and whose *scope* is exactly the JavaScripty expression e_2. Note that in particular, x is not in scope in e_1.

14.5 Free Variables

We can thus define a function to compute the free variables of a JavaScripty expression (represented in Scala) as follows:

```scala
def freeVars(e: Expr): Set[Var] = e match {
  case N(_) => Set.empty
  case Plus(e1, e2) => freeVars(e1) union freeVars(e2)
  case x @ Var(_)  => Set(x)
  case ConstDecl(x, e1, e2) => freeVars(e1) union (freeVars(e2) - Var(x))
}
```

```
defined function freeVars
```

For the `N(_)` case, there are no free-variable uses. For the `Plus(e1, e2)` case, the `Plus` node does that change the set of free variables, so it is union of the free variables of `e1` and `e2`.

The `x @ Var(_)` case is a free-variable use, so the singleton set `Set(x)` is the set of free variables. The `@` pattern in Scala enables binding a variable and matching a specific pattern.

The `ConstDecl(x, e1, e2)` case shows that it is a binding construct where the variable named by `x` is not in scope in `e1` but is in scope in `e2`. Uses of the variable named by `x` in `e2` thus must be removed, as they are no longer free outside of this `ConstDecl` expression.

To see `freeVars` in action, let us consider a JavaScripty expression in concrete syntax:

```
const four = (2 + 2); (four + four)
```

We can represent the above JavaScripty expression as an abstract syntax tree in Scala as follows, and let us bind Scala variables to all sub-expressions:

```scala
val e_n = N(2)
val e_plusnn = Plus(e_n, e_n)
val e_var = Var("four")
val e_plusvarvar = Plus(e_var, e_var)
val e_constdecl = ConstDecl("four", e_plusnn, e_plusvarvar)
```

```
e_n: N = N(n = 2.0)
e_plusnn: Plus = Plus(e1 = N(n = 2.0), e2 = N(n = 2.0))
e_var: Var = Var(x = "four")
e_plusvarvar: Plus = Plus(e1 = Var(x = "four"), e2 = Var(x = "four"))
e_constdecl: ConstDecl = ConstDecl(
  x = "four",
  e1 = Plus(e1 = N(n = 2.0), e2 = N(n = 2.0)),
  e2 = Plus(e1 = Var(x = "four"), e2 = Var(x = "four"))
)
```

We then compute the free variables with `freeVars` for each of these JavaScripty expressions:

```scala
val fv_n = freeVars(e_n)
val fv_plusnn = freeVars(e_plusnn)
val fv_var = freeVars(e_var)
val fv_plusvarvar = freeVars(e_plusvarvar)
val fv_constdecl = freeVars(e_constdecl)
```

```
fv_n: Set[Var] = Set()
fv_plusnn: Set[Var] = Set()
fv_var: Set[Var] = Set(Var(x = "four"))
fv_plusvarvar: Set[Var] = Set(Var(x = "four"))
fv_constdecl: Set[Var] = Set()
```

Note it is simply one software engineering decision for the `freeVars` function here to have type `Expr => Set[Var]`, that is, it returns a set of values with data-class type `Var`. Another possible choice is `Expr => Set[String]`, which instead returns a set of the strings within the `Var` uses in the given `Expr`. The former does convey a more specific type constraint; however, the latter has the same information. A good exercise is to rewrite `freeVars` to have type `Expr => Set[String]` to see the difference.

Exercise 14.1. Rewrite the `freeVars` function above to have the following type:

```
def freeVarsAlt(e: Expr): Set[String] = ???
```

```
defined function freeVarsAlt
```

14.6 Value Environments and Evaluation

As we discuss in Section 4.1.1 regarding value bindings in Scala, the meaning of an expression depends on the meaning of the free variables of an expression. One way to give meaning to free-variable uses is by referencing an environment that specifies the assumed meaning of each variable.

Let us consider a value environment for JavaScripty represented in Scala as a `Map[Var, Double]`:

```
type Env = Map[Var, Double]
```

```
defined type Env
```

Note that like with `freeVars` above in Section 14.5, it would also be reasonable to choose `type Env = Map[String, Double]` for the value environment mapping the variables names to their values.

As compared to the `eval` for number literals and addition in Section 9.3.2.1, our `eval` also takes a value environment `env`:

```
def eval(env: Env, e: Expr): Double = e match {
  case N(n) => n
  case Plus(e1, e2) => eval(env, e1) + eval(env, e2)
  case x @ Var(_)  => env(x)
  case ConstDecl(x, e1, e2) => {
    val v1 = eval(env, e1)
    eval(env + (Var(x) -> v1), e2)
  }
}
```

```
defined function eval
```

The `x @ Var(_)` case looks up the variable in the value environment `env`, while the `ConstDecl(x, e1, e2)` case extends the environment with a binding for evaluating `e2`.

Let us a define a "public-facing interface" function that calls `eval` with an empty environment (with some informational logging):

```
def evalExpr(e: Expr): Double = {
  print(s"$e   ")
  val v = eval(Map.empty, e)
  println(s"$v")
  v
}
```

```
defined function evalExpr
```

It works fine for number literals and addition:

```
val v_n = evalExpr(e_n)
val v_plusnn = evalExpr(e_plusnn)
```

```
N(2.0)   2.0
Plus(N(2.0),N(2.0))   4.0
```

```
v_n: Double = 2.0
v_plusnn: Double = 4.0
```

However, it fails unexpectedly for any expression with a free-variable use

```
val v_var = evalExpr(e_var)
```

```
val v_plusvarvar = evalExpr(e_plusvarvar)
```

as variables in scope must have a binding in the environment.

Let's make our requirement that `evalExpr` can only evaluate *closed* expressions explicit:

```
def evalExpr(e: Expr): Double = {
  require(freeVars(e).isEmpty, s"Expression $e is not closed.")
  print(s"$e   ")
  val v = eval(Map.empty, e)
  println(s"$v")
  v
}
```

```
defined function evalExpr
```

```
val v_plusvarvar = evalExpr(e_plusvarvar)
```

```
val v_constdecl = evalExpr(e_constdecl)
```

14.7 Renaming Bound Variables

Consider again the JavaScripty expression, along with two rewrites:

14.8 JavaScripty (JavaScript)

```
const four = (2 + 2); (four + four)
```

```
const x = (2 + 2); (x + x)
```

```
const fuzz = (2 + 2); (fuzz + fuzz)
```

14.9 Lettuce (OCaml)

```
let four = (2 + 2) in (four + four)
```

```
let x = (2 + 2) in (x + x)
```

```
let fuzz = (2 + 2) in (fuzz + fuzz)
```

14.10 Smalla (Scala)

```
val four = (2 + 2); (four + four)
```

```
val x = (2 + 2); (x + x)
```

```
val fuzz = (2 + 2); (fuzz + fuzz)
```

Even though these expressions are different in the concrete syntax and quite different for a human user, they are effectively the same for a language implementation. They certainly have the same meaning according to the `evalExpr` function defined above in Section 14.6.

Just like with the "grouping" structure as in Section 12.1 above, we want to make evident the binding structure of an expression. Therefore, we generally consider terms equivalent up to the renaming of *bound* variables (e.g., we "see" the three expressions given above as the "same" expression).

Renaming *bound* variables consistently is also called α*-renaming* (alpha-renaming) for historical reasons from the λ-calculus (lambda-calculus). Similarly, this equivalence relation on expressions is also called α-equivalence.

14.10.1 Higher-Order Abstract Syntax

One way to encode the binding structure into the abstract syntax representation is encode the binding structure of the object language using the variable binding in the meta language:

```
object HOAS {
  sealed trait Expr                                               // e ::=
  case class N(n: Double) extends Expr                            //   n
  case class Plus(e1: Expr, e2: Expr) extends Expr                // | e1 + e2
                                                                  // | x
  case class ConstDecl(e1: Expr, e2: Double => Expr) extends Expr  // | const x = e1; e2
}
```

```
defined object HOAS
```

Observe that there is no `Expr` AST node for variable uses in the object language, and instead they are represented by variable uses in the meta language in the `ConstDecl` AST node. This representation is called *higher-order abstract syntax*.

14.11 JavaScripty: Concrete Syntax: Declarations

Note that the JavaScripty grammar with **const** above (Equation 14.1) specifies the abstract syntax using notation borrowed from the concrete syntax. The actual concrete syntax of JavaScripty is less flexible than this abstract syntax to match the syntactic structure of JavaScript. For example,

```
Plus(N(1), ConstDecl("a", N(2), Var("a")))
```

```
res14: Plus = Plus(
  e1 = N(n = 1.0),
  e2 = ConstDecl(x = "a", e1 = N(n = 2.0), e2 = Var(x = "a"))
)
```

is an abstract syntax tree that would never be produced by the parser. That is,

```
1 + const a = 2; a
```

results in a parse error.

The JavaScripty grammar with **const** above (Equation 14.1) read as concrete syntax is ambiguous in multiple ways, including the relative precedence of **const**-bindings and **+**-expressions:

```
const b = 3; b + 4
```

JavaScript uses additional syntactic categories for "declarations" and "statements" layered on top of expressions. Variable bindings with **const** are declarations (and not expressions).

Thus, we give a more restrictive grammar for JavaScripty with declarations and statements matching the syntactic structure of JavaScript as follows:

$$
\begin{array}{rrcl}
\text{declarations} & d & ::= & \mathbf{const}\ x\ \texttt{=}\ e\texttt{;} \mid s \mid d_1\ d_2 \mid \varepsilon \\
\text{statements} & s & ::= & e\texttt{;} \mid \texttt{\{}\ d\ \texttt{\}} \mid \texttt{;} \\
\text{expressions} & e & ::= & \texttt{(}e\texttt{)} \mid \cdots \\
\text{variables} & x & &
\end{array}
$$

A declaration can be a **const** binding (with a trailing `;`), a statement s, or a sequence of declarations (i.e., $d_1\ d_2 \mid \varepsilon$ where we consider sequencing declarations right associative). A statement can be an expression e (with a trailing `;`), a block `{` d `}`, or an empty statement `;`. Note that JavaScript parsers (like Scala's) have some rules for semi-colon `;` inference to be a bit more flexible than this grammar. In the concrete syntax, expressions e can be parenthesized `(`e`)` and otherwise are value literals, n-ary expressions, etc.

To make JavaScripty variable declarations simpler, we also deviate slightly with respect to static scoping rules. Whereas JavaScript (like Scala) considers all bindings to be in the same scope in the same declaration list d, our JavaScripty `ConstDecl` bindings each introduce their own scope. Essentially, we consider **const** x `=` e_1 `;` d_2 in JavaScripty as **const** x `=` e_1 `;` `{` d_2 `}` in JavaScript.

15 Judgments

A *judgment* is a statement about syntactic objects, that is, asserts a relation on a set of objects. The form of 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. As parameters are meta-variables for syntactic objects, they 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*.

15.1 Grammars and Inference Rules

15.1.1 Example: Syntax

Recall from that a grammar defines inductively a set of syntactic objects. For example, we can describe the natural numbers using a unary notation. We give an explicit name to the set of syntactic objects describing natural numbers (i.e., we call the language of natural numbers here **Nat**).

$$\text{natural numbers} \quad \mathbf{Nat} \quad n \quad ::= \quad \mathsf{z} \mid \mathsf{s}(n) \tag{15.1}$$

We now define the language of natural numbers using judgments and inference rules. Let $n \in \mathbf{Nat}$ be the (unary) judgment form that says, "Syntactic object n is an element of the set we call **Nat**."

$$\boxed{n \in \mathbf{Nat}} \qquad \frac{}{\mathsf{z} \in \mathbf{Nat}}\ \text{Zero} \qquad \frac{n \in \mathbf{Nat}}{\mathsf{s}(n) \in \mathbf{Nat}}\ \text{Successor}$$

Figure 15.1: Defining the judgment form $n \in \mathbf{Nat}$ that says, "Syntactic object n is an element of the set we call **Nat**."

We define the judgment form $n \in \mathbf{Nat}$ with two inference rules named Zero and Successor. Rule Zero is an axiom that says that z is an element of the set we call **Nat**. Rule Successor says that $\mathsf{s}(n)$ is an element in the set we call **Nat** *if* n is an element in the set we call **Nat**. The italicized *if* in the previous sentence corresponds to the horizontal line of the inference rule. As a convention, we list the judgment form we are defining with a set of inference rules as a header in a box.

In Section 12.1, we see a grammar as an inductive data type. For example, we might translate the above grammar (Equation 15.1) into the following inductive data type in Scala:

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

```
defined class Nat
defined object Z
defined class S
```

We can see the inference rules defining the judgment form $n \in \mathbf{Nat}$ (Figure 15.1) as defining the same thing as the grammar (Equation 15.1) — an inductively-defined set **Nat**. However, we can also see the inference rules as defining a unary relation that judges when a n is the set named **Nat**, which we might translate into the following function in Scala:

```scala
def isNat(n: Nat): Boolean = n match {
  // Zero
  case Z => true
  // Successor
```

```
  case S(n) => isNat(n)
}

isNat(Z)
isNat(S(Z))
isNat(S(S(Z)))
```

```
defined function isNat
res1_1: Boolean = true
res1_2: Boolean = true
res1_3: Boolean = true
```

Of course, this function is a silly one to write, as it will always return true. That is, we have this trivial meta-theorem:

Proposition 15.1 (All ns are elements of the set we call **Nat**)**.** *For all n, $n \in \mathbf{Nat}$.*

A standard shorthand is that when we write a judgment "$n \in \mathbf{Nat}$" in this context, we mean, "the judgment $n \in \mathbf{Nat}$ holds."

The `isNat` function is called the characteristic function of the set **Nat**.

With this trivial meta-theorem, we see grammars and this form of inference rules interchangeable for defining syntax. However, as inference rules are a more general form of inductive definitions (in that they are n-ary relations), we generally use grammars to define syntax and instead use inference rules to define semantics.

15.1.2 Key Intuition

While the function `isNat` function is silly, we see that the meta-language of judgment forms, inference rules, and judgments in mathematical specification corresponds to function signatures, function bodies, and function calls in code. Just like the that the meta-language of meta-variables, grammars, and terms corresponds to types, inductive data type definitions, and values.

15.2 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 of a judgment* or sometimes simply 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 derivation:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{z} \in \mathbf{Nat}}\ \textsc{Zero}}{\mathsf{s}(\mathsf{z}) \in \mathbf{Nat}}\ \textsc{Successor}}{\mathsf{s}(\mathsf{s}(\mathsf{z})) \in \mathbf{Nat}}\ \textsc{Successor}$$

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

Note the same term *derivation* is also used in the context of a *parsing derivation* that witnesses when a given string is a sentence in a given grammar (cf. Section 11.2.1). Observe that in both cases, the term *derivation* refers to witnessing an instance of an inductive definition.

Given that the meta-language of judgment forms, inference rules, and judgments in mathematical specification corresponds to function signatures, function bodies, and function calls in code, the notion of derivations of a judgment corresponds to the execution of a function call.

We instrument `isNat` to show the correspondence between the derivation of the judgment $n \in \mathbf{Nat}$ and an execution trace of `isNat(n)` of some test cases for `n`, specifically `Z`, `S(Z)`, and `S(S(Z))`.

```
def isNat(n: Nat): Boolean = {
  val r = n match {
    // Zero
    case Z => {
      val r = true
      println("------------- Zero")
      r
    }
    // Successor
    case S(n) => {
      val r = isNat(n)
      println("------------- Successor")
      r
```

```
    }
  }
  println(s"$n   Nat")
  r
}
```

```
defined function isNat
```

```
isNat(Z)
```

```
------------- Zero
Z   Nat

res3: Boolean = true
```

```
isNat(S(Z))
```

```
------------- Zero
Z   Nat
------------- Successor
S(Z)   Nat

res4: Boolean = true
```

```
isNat(S(S(Z)))
```

```
------------- Zero
Z   Nat
------------- Successor
S(Z)   Nat
------------- Successor
S(S(Z))   Nat

res5: Boolean = true
```

15.3 Inductively-Defined

15.3.1 Example: Structural Equality

As another example, let us define when two natural numbers n_1, n_2 are structurally equal. That is, we define the judgment form $n_1 =_{\textbf{Nat}} n_2$ that we intend to mean, "Natural number n_1 is structurally equal to natural number n_2." as the least relation closed under the inference rules specified in Figure 15.2.

$$\boxed{n_1 =_{\textbf{Nat}} n_2} \qquad \frac{}{\mathsf{z} =_{\textbf{Nat}} \mathsf{z}}\ \textsc{ZeroEq} \qquad \frac{n_1 =_{\textbf{Nat}} n_2}{\mathsf{s}(n_1) =_{\textbf{Nat}} \mathsf{s}(n_2)}\ \textsc{SuccessorEq}$$

Figure 15.2: Defining structural equality on natural numbers n. The judgment form $n_1 =_{\textbf{Nat}} n_2$ says, "Natural number n_1 is structurally equal to natural number n_2."

What it means to be the *least* relation is that we read the inference rules inductively. Intuitively, this means that a judgment holds if and only if there is a derivation for it.

```
def eqNat(n1: Nat, n2: Nat): Boolean = {
  val r = (n1, n2) match {
    // ZeroEq
    case (Z, Z) => {
      println("----------------------- ZeroEq")
      true
    }
    // SuccessorEq
    case (S(n1), S(n2)) => {
      val r = eqNat(n1, n2)
      println("----------------------- SuccessorEq")
      r
    }
    // No Rules
    case _ => false
  }
  println(s"$n1 =Nat $n2")
  r
}
```

```
defined function eqNat
```

We see using the instrumented function `eqNat` corresponding to the judgment form $n_1 =_{\mathbf{Nat}} n_2$ that we get complete derivations (i.e., end in applications of axioms) for the judgments $\mathsf{z} =_{\mathbf{Nat}} \mathsf{z}$ and $\mathsf{s}(\mathsf{z}) =_{\mathbf{Nat}} \mathsf{s}(\mathsf{z})$ that should hold:

```
eqNat(Z, Z)
```

```
----------------------- ZeroEq
Z =Nat Z

res7: Boolean = true
```

```
eqNat(S(Z), S(Z))
```

```
----------------------- ZeroEq
Z =Nat Z
----------------------- SuccessorEq
S(Z) =Nat S(Z)

res8: Boolean = true
```

And we cannot complete the derivation for the judgment $\mathsf{s}(\mathsf{s}(\mathsf{z})) =_{\mathbf{Nat}} \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$ that should not hold:

```
eqNat(S(S(Z)), S(S(S(Z))))
```

```
Z =Nat S(Z)
----------------------- SuccessorEq
S(Z) =Nat S(S(Z))
----------------------- SuccessorEq
S(S(Z)) =Nat S(S(S(Z)))

res9: Boolean = false
```

15.4 Functions versus Relations

Mathematically, judgment forms are inductively-defined relations. Thus far, when we translate them to functional programs, we have translated them into their characteristic functions (i.e., has return type `Boolean`). In some cases, the relations we define judgmentally are more naturally read as functions. And as such, we want to translate them to functions in Scala.

15.4.1 Example: Semantics

Let us write i as the meta-variable for a mathematical integer (i.e., $\mathbb{Z}$). In particular, we do not define syntax for i. We define an interpretation of syntactic natural numbers n into integers i with the judgment form $n \Downarrow i$, which we read as, "Natural number n evaluates to integer i."

$$\boxed{n \Downarrow i} \qquad \overset{\textsc{EvalZero}}{\dfrac{}{\mathsf{z} \Downarrow 0}} \qquad \overset{\textsc{EvalSuccessor}}{\dfrac{n \Downarrow i}{\mathsf{s}(n) \Downarrow i + 1}}$$

Figure 15.3: Defining structural equality on natural numbers n. The judgment form $n_1 =_{\mathbf{Nat}} n_2$ says, "Natural number n_1 is structurally equal to natural number n_2."

In defining a Scala implementation, let us choose the Scala type `Int` to represent i. Then, we see the judgment form $n \Downarrow i$ defines an `eval` function with which we are already familiar:

```scala
def eval(n: Nat): Int = {
  val i = n match {
    case Z => {
      println("------------ EvalZero")
      0
    }
    case S(n) => {
      val i = eval(n)
      println("------------ EvalSuccessor")
      i + 1
    }
  }
  println(s"$n   $i")
  i
}
```

```
defined function eval
```

```scala
eval(Z)
```

```
------------ EvalZero
Z   0
```

```
res11: Int = 0
```

```scala
eval(S(Z))
```

```
------------ EvalZero
Z   0
------------ EvalSuccessor
S(Z)   1

res12: Int = 1
```

```scala
eval(S(S(Z)))
```

```
------------ EvalZero
Z   0
------------ EvalSuccessor
S(Z)   1
------------ EvalSuccessor
S(S(Z))   2

res13: Int = 2
```

We are able to translate the judgment form $n \Downarrow i$ into the `eval` function in Scala because we can show that it indeed defines a function:

Proposition 15.2 (Deterministic Evaluation)**.** *If* $n \Downarrow i_1$ *and* $n \Downarrow i_2$*, then* $i_1 = i_2$*.*

16 Lab: Basic Values, Variables, and Judgments

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

- the distinction between concrete and abstract syntax;
- the relationship between judgment forms/inference rules/judgments and implementation code;
- using a reference implementation as a definition of semantics;
- variable binding and variable environments.

Functional Programming Skills Recursion over abstract syntax. Representation invariants.
Programming Language Ideas Inductive definitions (grammars/productions/sentences and judgment forms/inference rules/judgments). Semantics (via detective work).

Instructions

A version of project files for this lab resides in the public pppl-lab2 repository. Please follow separate instructions to get a private clone of this repository for your work.

You will be replacing ??? or `case _ => ???` in the `Lab2.scala` file with solutions to the coding exercises described below.

Your lab will not be graded if it does not compile. You may check compilation with your IDE, `sbt compile`, or with the "sbt compile" GitHub Action provided for you. Comment out any code that does not compile or causes a failing assert. Put in `???` as needed to get something that compiles without error.

You may add additional tests to the `Lab2Spec.scala` file. In the `Lab2Spec.scala`, there is empty test class `Lab2StudentSpec` that you can use to separate your tests from the given tests in the `Lab2Spec` class. You are also likely to edit `Lab2.worksheet.sc` for any scratch work. You can also use `Lab2.worksheet.js` to write and experiment in a JavaScript file that you can then parse into a JavaScripty AST (see `Lab2.worksheet.sc`).

If you like, you may use this notebook for experimentation. However, **please make sure your code is in `Lab2.scala`; this notebook will not graded.**

Recall that you need to switch kernels between running JavaScript and Scala cells.

16.1 Interpreter: JavaScripty Calculator

In this lab, we extend JavaScripty with additional value types and variable binding. That is, the culmination of the lab is to implement an interpreter for the subset of JavaScript with numbers, booleans, strings, the undefined value, and variable binding.

```
trait Expr                               // e ::=
case class N(n: Double) extends Expr //       n

type Env = Map[String, Expr]
val empty: Env = Map.empty
```

```
defined trait Expr
defined class N
defined type Env
empty: Env = Map()
```

```
def eval(env: Env, e: Expr): Expr = ???
```

```
defined function eval
```

We leave the `Expr` inductive data type mostly undefined for the moment to focus on the type of `eval`.

First, observe that the return type of `eval` is an `Expr` (versus `Double` in the previous lab), as we now have more value types. However, `eval` should return an `Expr` that is a JavaScripty value (i.e., is an e : `Expr` such that `isValue(` e `)` returns **`true`**). The need for the object-language versus meta-language distinction is even more salient here than in the previous lab. For example, it is critical to keep straight that `N(1.0)` is the Scala value representing the JavaScripty value `1.0`.

Second, observe that the `eval` function takes a JavaScripty value environment `env: Env` to give meaning to free JavaScripty variables in `e`.

These ideas take unpacking, so let us start from the arithmetic sub-language of JavaScripty:

```scala
case class Unary(uop: Uop, e1: Expr) extends Expr           // e ::= uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr //     | e1 bop e2

trait Uop                     // uop ::=
case object Neg extends Uop   //   -

trait Bop                     // bop ::=
case object Plus extends Bop  //   +
case object Minus extends Bop // | -
case object Times extends Bop // | *
case object Div extends Bop   // | /
```

```
defined class Unary
defined class Binary
defined trait Uop
defined object Neg
defined trait Bop
defined object Plus
defined object Minus
defined object Times
defined object Div
```

Thus far, we have considered only one type of value in our JavaScripty object language: numbers n (which we have considered double-precision floating-point numbers). Specifically, we have stated the following:

$$\begin{array}{rrcl} \text{expressions} & e & ::= & v \\ \text{values} & v & ::= & n \end{array}$$

Recall from our preliminary discussion about evaluation (Section 3.3) that the computational model is a rewriting or reduction of expressions until reaching values. A value is simply an expression that cannot be reduced any further. Thus, we can also consider a unary judgment form $e\ \mathsf{value}$ that judges when an expression is a value.

$$\textsc{NumVal}\quad \frac{}{n\ \mathsf{value}}$$

This judgment form corresponds to the `isValue` function:

```scala
def isValue(e: Expr): Boolean = e match {
  case N(_) => true
  case _ => false
}
```

```
defined function isValue
```

From our discussion on grammars and inference rules (Section 15.1), we can see this unary judgment form e value as defining a syntactic category values v. Thus, we freely write grammar productions for v to define the judgment form e value.

Exercise 16.1. For this part of the lab, implement `eval` for the calculator language above (which is the same language as in the previous lab) but now with type `(Env, Expr) => Expr`.

```scala
def eval(env: Env, e: Expr): Expr = ???
```

```
defined function eval
```

You will not use the `env` parameter yet. You will want to check that your implementation returns JavaScripty values. For example,

16.2 Coercions: Basic Values

16.2.1 Booleans, Strings, and Undefined

Like most other languages, JavaScript has other basic value types. Let us extend JavaScripty with Booleans, strings, and the **undefined** value:

$$
\begin{array}{rrcl}
\text{values} & v & ::= & b \mid \mathit{str} \mid \textbf{undefined} \\
\text{booleans} & b & & \\
\text{strings} & \mathit{str} & &
\end{array}
$$

Boolean values b are the literals **true** and **false**. String values str are string literals like `"hi"` that we do not explicitly define here.

The **undefined** literal is a distinguished value that is different than all other values. It is like the unit literal `()` in Scala.

```
case class B(b: Boolean) extends Expr  // e ::= b
case class S(str: String) extends Expr //      | str
case object Undefined extends Expr     //      | undefined
```

```
defined class B
defined class S
defined object Undefined
```

Examples

```
true
false
"Hello"
"Hola"
undefined
```

We update our `isValue` function appropriately:

```
def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) | S(_) | Undefined => true
  case _ => false
}
```

```
defined function isValue
```

A good exercise here is to reflect on what this code translates to in terms of additional inference rules for the e value judgment form.

16.2.2 Expressions

Each value type comes with some operations. Our abstract syntax tree has two constructors for unary and binary expressions that are parametrized by unary *uop* and binary *bop* operators, respectively:

$$\text{expressions} \quad e \quad ::= \quad \mid \mathit{uop}\, e_1 \mid e_1\, \mathit{bop}\, e_2$$

16.2.2.1 Numbers

For example, numbers has

$$
\begin{array}{rrcl}
\text{unary operators} & uop & ::= & \texttt{-} \\
\text{binary operators} & bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{array}
$$

that we considered previously.

16.2.2.2 Booleans

For booleans, we add unary negation `!` and binary conjunction `&&` and disjunction `||`.

$$
\begin{array}{rrcl}
\text{unary operators} & uop & ::= & \texttt{!} \\
\text{binary operators} & bop & ::= & \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

```
case object Not extends Uop // uop ::= !
case object And extends Bop // bop ::= &&
case object Or extends Bop  //       | ||
```

```
defined object Not
defined object And
defined object Or
```

ℹ Examples

```
!true
true && false
true || false
```

We also expect to be able to elimate booleans with a conditional if-then-else expression:

$$
\text{expressions} \quad e \quad ::= \quad e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3
$$

```
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr // e ::= e1 ? e2 : e3
```

```
defined class If
```

We also expect to be able to compare values for equality and disquality and numbers for inequality:

$$\text{binary operators} \quad \mathit{bop} \quad ::= \quad \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}$$

```
case object Eq extends Bop // bop ::= ===
case object Ne extends Bop //       | !==
case object Lt extends Bop //       | <
case object Le extends Bop //       | <=
case object Gt extends Bop //       | >
case object Ge extends Bop //       | >=
```

```
defined object Eq
defined object Ne
defined object Lt
defined object Le
defined object Gt
defined object Ge
```

16.2.2.3 Strings

The string operations we support in JavaScripty are string concatenation and string comparison. In JavaScript, string concatenation is written with the binary operator + and string comparison using <, <=, >, and >=, so we do not need to extend the syntax.

> **Examples**
>
> ```
> "Hello" + ", " + "World" + "!"
> ```

16.2.2.4 Undefined

As **undefined** corresponds to unit () in Scala and is uninteresting in itself, we add a side-effecting expression that prints to console.

$$\text{expressions} \quad e \quad ::= \quad \texttt{console.log}(e_1)$$

```
case class Print(e1: Expr) extends Expr // e ::= console.log(e1)
```

```
defined class Print
```

ℹ Examples

```
console.log("Hello, World!")
```

If we now have side-effecting expressions, then we would expect to have a way to sequence executing expressions for their effects.

$$\text{binary operators} \quad bop \quad ::= \quad ,$$

```
case object Seq extends Bop // bop ::= ,
```

```
defined object Seq
```

ℹ Examples

```
undefined, 3
```

16.2.3 Semantics Detective: JavaScript is Bananas

In the above, we have carefully specified the abstract syntax of the object language of interest and informally discussed its semantics. But if we are to implement an interpreter in the `eval` function, we also need to define its semantics! And we give a precise definition as follows:

! Important

In this lab, the semantics of a JavaScripty expression e is defined by the evaluation of it as a JavaScript program.

Given the careful specification of the abstract syntax, a natural question to ask is whether all abstract syntax trees of type `Expr` in the above are valid expressions and have a semantics in JavaScript (and hence JavaScripty). Is `3 + true` a valid expression?

```
//| filename: JavaScript
3 + true
```

We try it out and see that yes it is. One aspect that makes the JavaScript specification interesting is the presence of implicit coercions (e.g., non-numeric values, such as booleans or strings, may be implicitly converted to numeric values depending on the context in which they are used).

You might guess that defining coercions between value types can lead to some interesting semantics. It is because of these coercions that we have the meme that “JavaScript is bananas.”

```
//| filename: JavaScript
"b" + "a" + "n" + - "a" + "a" + "s"
```

Armed with knowledge that in JavaScript, numbers are floating-point numbers, the + operator in JavaScript is overloaded for strings and numbers, and coercions happen between value types, see if you can explain what is happening in the “bananas” expression above.

16.2.3.1 Coercions

Our `eval` function interpreter will need to make use of three helper functions for converting values to numbers, booleans, and strings:

```
def toNumber(v: Expr): Double = ???
def toBoolean(v: Expr): Boolean = ???
def toStr(v: Expr): String = ???
```

```
defined function toNumber
defined function toBoolean
defined function toStr
```

Exercise 16.2. Write at least 1 JavaScript expression that shows a coercion from a non-numeric value to a number and see what the result should be:

```
//| filename: JavaScript
// YOUR CODE HERE
undefined
```

Then, translate this JavaScript expression (written in concrete syntax) into an `Expr` value (i.e., a JavaScripty abstract syntax tree).

Finally, use this `Expr` as a unit test for `toNumber` in the `Lab2StudentSpec` class in the `Lab2Spec.scala` file.

Exercise 16.3. Do the same to create a `toBoolean` test. Write at least 1 JavaScript expression that shows a coercion from a non-boolean value to a boolean, see what the result should be, translate it to an `Expr` value, and add it as test in `Lab2StudentSpec`.

Listing 16.1 `Lab2Spec.scala`

```
val e_toNumber_test1 = ???
"toNumber" should s"${e_toNumber_test1}" in {
  assertResult(???) { toNumber(e_toNumber_test1) }
}
```

```
//| filename: JavaScript
// YOUR CODE HERE
undefined
```

Exercise 16.4. Do the same to create a `toStr` test. Write at least 1 JavaScript expression that shows a coercion from a non-string value to a string, see what the result should be, translate it to an `Expr` value, and add it as test in `Lab2StudentSpec`.

```
//| filename: JavaScript
// YOUR CODE HERE
undefined
```

Exercise 16.5. Implement `toNumber`, `toBoolean`, and `toStr` in `Lab2.scala` (using test-driven development with the test cases you have written above).

Exercise 16.6. For this part of the lab, extend your `eval` from Exercise 16.1 for booleans, strings, the undefined value, and printing.

```
def eval(env: Env, e: Expr): Expr = ???
```

```
defined function eval
```

You still will not use the `env` parameter yet. You again will want to check that your implementation returns JavaScripty values using the latest `isValue`. For example,

16.3 Interpreter: JavaScripty Variables

The final piece of this lab is to extend our interpreter with variable uses and binding (cf. Chapter 14).

$$\text{expressions} \quad e \quad ::= \quad x \mid \textbf{const}\ x \ \texttt{=}\ e_1 \ \texttt{;}\ e_2$$

```scala
case class Var(x: String) extends Expr                          // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr //     | const x = e1; e2
```

```
defined class Var
defined class ConstDecl
```

Note that the above is the abstract syntax we consider for `ConstDecl`, which is more flexible the concrete syntax for `const` allowed by JavaScript (cf. Section 14.11)

In this lab, we define a value environment as a map from variable names to JavaScripty values, which we represent in Scala as a value of type `Map[String, Expr]`. Note that representing variable names as Scala `String`s here, and it is a representation invariant that the `Expr`s must correspond to JavaScripty values.

```scala
type Env = Map[String, Expr]
val empty: Env = Map.empty

def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined type Env
empty: Env = Map()
defined function lookup
defined function extend
```

We provide the above value and function bindings to interface with the Scala standard library for `Map[String, Expr]` and maintain this representation invariant. You may use the Scala standard library directly if you wish, but we recommend that you just use these interfaces, as they are all that you need and give you the safety of enforcing the representation invariant. The `empty` value represents an empty value environment, the `lookup` function gets the value bound to the variable named by a given string, and the `extend` function extends a given environment with a new variable binding.

Exercise 16.7. For this part of the lab, extend your `eval` from Exercise 16.1 for variable uses (i.e., `Var`) and variable binding (i.e., `ConstDecl`). We suggest you start by adding tests with variables uses and `const` bindings to be able do test-driven development (see below).

```scala
def eval(env: Env, e: Expr): Expr = ???
```

```
defined function eval
```

Testing

In this lab, we have carefully defined the syntax of the JavaScripty variant of interest, and we have defined its semantics to be defined to be the same as JavaScript. Thus, you can write any JavaScript program within the syntax defined above to create test cases for your `eval` function. Any of the above JavaScript examples could be used as test cases. In some cases, you may want to write abstract syntax trees directly in Scala (i.e., values of type `Expr`). In other cases, you can use the provided JavaScripty parser to translate concrete syntax (i.e., values of type `String`) into abstract syntax (i.e., values of type `Expr`).

Exercise 16.8 (Optional)**.** We give some exercises below to explore this subset of JavaScripty that you can then use as test cases that you add to your `Lab2StudentSpec`.

16.3.0.1 Basic Arithmetic Operations

This program defines two constants, `x` and `y`, and calculates their sum. It then logs the result `sum` to the console.

```javascript
const x = 10;
const y = 5;
const sum =
  // YOUR CODE HERE (replace undefined)
  undefined
;
console.log(sum);
```

```javascript
console.assert(sum === 15)
```

16.3.0.2 Conditional Expressions

```
const a = 10;
const b = 20;
const max =
  // YOUR CODE HERE (replace undefined)
  undefined
;
```

```
console.assert(max === 20)
```

16.3.0.3 Unary and Binary Operations

This program checks if a number is positive using a unary negation `-` and a binary relational operator.

```
const num = -5;
const isPositive =
  // YOUR CODE HERE (replace undefined)
  undefined
;
```

```
console.assert(isPositive === false)
```

16.3.0.4 Undefined

This program demonstrates the correspondence between `undefined` in JavaScript and `()`.

```
const r = console.log("Hello");
```

```
console.assert(r === undefined)
```

Submission

If you are a University of Colorado Boulder student, we use Gradescope for assignment submission. In summary,

- ☐ Create a private GitHub repository by clicking on a GitHub Classroom link from the corresponding Canvas assignment entry.
- ☐ Clone your private GitHub repository to your development environment (using the `<> Code` button on GitHub to get the repository URL).

- ☐ Work on this lab from your cloned repository. Use Git to save versions on GitHub (e.g., `git add`, `git commit`, `git push` on the command line or via VSCode).
- ☐ Submit to the corresponding Gradescope assignment entry for grading by choosing GitHub as the submission method.

You need to have a GitHub identity and must have your full name in your GitHub profile in case we need to associate you with your submissions.

17 Review: Syntax

Instructions

This assignment is a review exercise in preparation for a subsequent assessment activity.

This is a peer-quizzing activity with two students. Each section has an even number of exercises. Student A quizzes Student B on the odd numbered exercises, and Student B quizzes Student A on the even numbered exercises.

To the best of your ability, give feedback using the learning-levels rubric below on where your peer is in reaching or exceeding Proficient (P) on each question live. Guidance of what a Proficient (P) answer looks like are given.

There may or may not be a member of the course staff assigned to your slot. It is expected that regardless of whether a member of the course staff is present, this is a peer-quizzing activity. If a member of the course staff is present, you may ask for their help and guidance on answering the questions and/or their assessment of where you are at in your learning level.

It is not expected that you can complete all exercises in the allotted time. You and your partner may pick and choose which sections you want to focus on and use the remaining questions as a study guide. You and your partner may, of course, continue working together after the scheduled session.

At the same time, most questions can be answered in a few minutes with a Proficient (P) level of understanding. Aim for 3–4 sections in 30 minutes.

Your submission for this session is an overall assessment of where your partner is in their reaching-or-exceeding-proficiency level. Be constructive and honest. **Neither your nor your partners grade will depend on your learning-level assessment.** Instead, your score for this assignment will be based on the thoughtfulness of your feedback to your partner.

Submit on Gradescope as a pair. That is, use Gradescope's group assignment feature to submit as a group. The submission form has a spot for each of you to provide your assessment and feedback for each other.

Please proactively fill slots with an existing sign-up to have a partner. In case your peer does not show up to the slot, try to join another slot happening at the same time from the course calendar. If that fails and a course staff member is present, you may do the exercise with the staff member and get credit. If there is no staff member present, you may try to find a slot at a later time if you like or else write to the Course Manager on Piazza timestamped during the slot.

Learning-Levels Rubric

4 - Exceeding (E) Student demonstrates synthesis of the underlying concepts. Student can go beyond merely describing the solution to explaining the underlying reasoning and discussing generalizations.

3 - Proficient (P) Student is able to explain the overall solution and can answer specific questions. While the student is capable of explaining their solution, they may not be able to confidently extend their explanation beyond the immediate context.

2 - Approaching (A) Student may able to describe the solution but has difficulty answering specific questions about it. Student has difficulty explaining the reasoning behind their solution.

1 - Novice (N) Student has trouble describing their solution or responding to guidance. Student is unable to offer much explanation of their solution.

17.1 Abstract Syntax Trees

Exercise 17.1. Consider the following buggy implementation of `nnf` that attempts to convert a boolean expression represented as a `BExpr` into negation normal form (Exercise 13.2). What's correct and what's buggy about this implementation?

```
sealed trait BExpr
case class Var(x: String) extends BExpr
case class Not(e: BExpr) extends BExpr
case class And(e1: BExpr, e2: BExpr) extends BExpr
case class Or(e1: BExpr, e2: BExpr) extends BExpr

def nnf(e: BExpr): BExpr = e match {
  case Not(Not(e1)) => e1 // Remove double negation
  case Not(And(e1, e2)) => Or(Not(e1), Not(e2)) // De Morgan's Law for conjunction
  case Not(Or(e1, e2)) => And(Not(e1), Not(e2)) // De Morgan's Law for disjunction
  case _ => e
}
```

```
defined trait BExpr
defined class Var
defined class Not
defined class And
```

```
defined class Or
defined function nnf
```

> A Proficient (P) answer recognizes that the pattern matching implementing the rules to convert to negation normal form is correct, but the following are missing: base cases, pass-through recursive cases for `And` and `Or`, and the recursive calls to normalize the appropriate sub-expressions in the `Not` cases. A Proficient (P) answer should be able articulate which test cases will work and which will not.

Exercise 17.2. Fix this buggy implementation of `nnf` and argue that it correctly converts any `BExpr` into negation normal form. It is sufficient to do this by writing on a sheet of paper or a whiteboard.

> A Proficient (P) answer will be able add the missing base cases, the cases for `And` and `Or`, and fix the `Not` cases. The correctness argument should state something about normalizing recursively or by induction.
>
> An Exceeding (E) answer should recognize that the induction hypothesis needs to be general enough for both expressions e and their negation `Not(` e `)`.

17.2 Ambiguity Detective

Exercise 17.3. Consider a programming language with some binary infix-operator expressions. When is it possible to test the relative precedence of those operators by evaluating example expressions? How would you do it? Can you give an example? Explain.

> A Proficient (P) answer should recognize this is what was asked in the Precedence Detective exercise (Exercise 13.4) with `<<` and `-`. It should state that if running different versions of an expression using `<<` and `-` corresponding to different precedence orders yields different values, then it is possible to test relative precedence.

Exercise 17.4. How about for associativity? Can you give a positive example an operator that you can test its parsing-associativity by evaluating expressions and a negative example where you cannot in Scala?

> A Proficient (P) answer should recognize that different versions of an expression corresponding to different associativities may yield the same answer. In this case, one cannot test. In math, an operation is called an *associative* operation where parsing expressions with the operator as left or right associative does not change its semantics. Note the difference in using two uses of “associative” in the last sentence. A standard example for an associative operation (in math) is $+$, while $-$ is not.

17.3 Grammars

Exercise 17.5. Consider the following grammar:

$$\begin{array}{rcl} S & ::= & A\ B\ B\ A \\ A & ::= & \mathtt{a} \mid \mathtt{a}\ A \\ B & ::= & \mathtt{b}\ B\ \mathtt{c} \mid B\ B \mid \varepsilon \end{array}$$

1. Describe the sentences of the language defined by this grammar.
2. Give two positive example sentences in the language described by this grammar and two negative example strings not in the language described by this grammar.
3. For each positive example sentence, give a leftmost derivation and a parse tree.
4. For each negative example string, argue why they are not by described the grammar (e.g., show getting stuck trying to construct parse trees).

A Proficient (P) answer sees that A describes the language of one-or-more `a`'s and B as the language of matching `b`'s and `c`'s. Thus, S must have one-or-more `a`'s followed by matching `b`'s and `c`'s followed by one-or-more `a`'s. A Proficient (A) answer will be able to give derivations, parse trees, etc. for 2–4.

Exercise 17.6. Consider the following two grammars for expressions e:

$$e \quad ::= \quad \mathit{operand} \mid e\ \mathit{operator}\ \mathit{operand} \tag{17.1}$$

$$\begin{array}{rcl} e & ::= & \mathit{operand}\ \mathit{esuffix} \\ \mathit{esuffix} & ::= & \mathit{operator}\ \mathit{operand}\ \mathit{esuffix} \mid \varepsilon \end{array} \tag{17.2}$$

In both grammars, *operator* and *operand* are the same; you do not need to know their productions for this question.

1. Describe the expressions generated by the two grammars.
2. Do these grammars generate the same or different expressions? Explain.

Hint: Think about both the concrete syntax (sentences or strings) and the abstract syntax (terms or trees).

A Proficient (P) answer will recognize that in both grammars, the language described by e is one-or-more *operand*'s separated by *operator*s. It should state they describe the same language, that is, they are the same in terms of strings and concrete syntax. They are both refactorings of a common binary *operator* grammar.

A Proficient (P) answer will recognize that neither grammar is ambiguous and in terms of parsing, produce different parse trees. The first grammar has left recursion in e, while the second grammar has right recursion in *esuffix*.

17.4 Concrete Syntax, Abstract Syntax, and Semantics

Consider the following grammar:

$$\begin{array}{rcl} A & ::= & B \mid \otimes A \oslash A \mid A \oplus B \\ B & ::= & \texttt{a} \mid \texttt{b} \end{array}$$

Exercise 17.7. Is the above grammar ambiguous? If so, prove that it is ambiguous. If not, argue informally why it isn't.

> A Proficient (P) answer will recognize that the grammar is ambiguous involving the second and third productions. An example sentence that shows the ambiguity is $\otimes\texttt{a} \oslash \texttt{a} \oplus \texttt{a}$. It will state a sentence like this one and give two different (valid) parse trees for it.

Exercise 17.8. Let us ascribe a semantics to the syntactic objects A specified in the above grammar. In particular, let us write

$$A \Downarrow n$$

for the judgment form that should mean A has a total number n of `a` symbols where n is the meta-variable for numbers. Define this judgment form via a set of inference rules. You may rely upon arithmetic operators over numbers.

Hint: There should be one inference rule for each production of the non-terminal A (called a syntax-directed judgment form).

> A Proficient (P) answer will define this judgment using three inference rules—one for each production of A. It is also a good answer to define a judgment form $B \Downarrow n$ with one inference rule.
>
> A Exceeding (E) answer will realize that one could give these two possible answers.

17.5 Interpreter Implementation

Exercise 17.9. Some binary operators *bop* are overloaded for numbers and strings in JavaScript(y), that is, they apply a different number or string operation depending on the type of the operands.

1. To implement your `eval` function, how did you discover which ones? Which ones did you discover are overloaded?
2. Give an example JavaScript(y) expression that performs a string operation after coercing a number to a string.

3. On paper or a whiteboard, trace through your `eval` implementation using your test case from 2. It is fine if you discover a bug in your `eval` implementation doing this exercise.

Use this following notation to show the key steps in running your Scala implementation:

`eval(` env `,` e `) =` v

- if `eval(` env_1 `,` e_1 `) =` v_1
  - if `eval(` env'_1 `,` e'_1 `) =` v'_1
  - ...
- if `eval(` env_2 `,` e_2 `) =` v_2
  - ...
- ...

where each "node" in the tree above is a recursive call to `eval`. You may write env, e, v as Scala values (i.e., the Scala representation of JavaScripty) or JavaScripty concrete syntax as you prefer.

> A Proficient (P) answer will say that number addition and string concatenation use the same operator `+` and that number comparison and lexicographic-string comparison is also overloaded with `<`, `<=`, `>`, and `>=`. The answer should include that they wrote and ran JavaScript expressions that should performing number addition versus string concatenation (and similarly for comparisons) to discover that **+** is considered string concatenation if *either* argument is a string, while `<`, `<=`, `>`, and `>=` are considered lexicographic-string comparison only if *both* arguments are strings. It should then use `+` to give a test case that coerces a number to a string.
>
> A Proficient (P) answer for the tracing should have the right number of `eval` calls on sub-expressions to reach the bases cases with the appropriate indentions showing recursive calls.

Exercise 17.10. Trace through your `eval` implementation for the JavaScripty test case

const `abc = 1 + 2; abc`

or equivalently, for

```
eval(Map(), ConstDecl("abc", Binary(Plus, N(1), N(2)), Var("abc")))
```

using the notation given above.

> A Proficient (P) answer will have exactly 5 calls to `eval`.

As noted above, it is fine to answer with concrete JavaScripty syntax in place of the Scala abstract syntax tree representation as long as it is understood that this is just for ease of writing it on the board and is not what “Scala sees”. It is a below Proficient (P) indicator if there’s confusion about what is concrete syntax for the board and what are abstract syntax trees.

Part IV

Language Design and Implementation

18 Operational Semantics

In the previous part, 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.

18.1 Big-Step Operational Semantics

18.1.1 JavaScript is Bananas

We might guess the semantics of particular expressions based on common conventions. For example, we might guess that expression

$$e_1 \texttt{+} e_2$$

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

As we have seen in the previous lab (Section 16.2.3), one aspect that makes the JavaScript specification "interesting" 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,

```
//| filename: JavaScript
true + 2
```

evaluates to `3`.

Then, the `+` operator in JavaScript is overloaded for strings and numbers with `+` on strings meaning string concatenation:

```
//| filename: JavaScript
"Hello, " + "World!"
```

So $e_1 + e_2$ may not be just adding two numbers!

You might guess that defining coercions between value types can lead to some interesting semantics. It is because of these coercions that we have the meme that "JavaScript is bananas."

```
//| filename: JavaScript
"b" + "a" + "n" + - "a" + "a" + "s"
```

How can we describe how to implement a interpreter for all programs?

18.1.2 An Evaluation Judgment

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 methodology for semantics 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. A set of inference rules defining such an evaluation judgment form is called a *big-step operational semantics* for expressions e because it describes evaluation from expressions in one "big step" to values. Another term for a big-step operational semantics is a *natural semantics*.

18.2 One Type of Values

Let us first consider an object language with only one type of values—numbers. In particular, we consider just numbers n and one arithmetic operator `+`:

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & v \mid e_1 \mathbin{bop} e_2 \\
\text{values} & v & ::= & n \\
\text{binary operators} & bop & ::= & \texttt{+} \\
\text{numbers} & n & &
\end{array}
$$

```scala
trait Expr // e
trait Bop  // bop

case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr // e ::= e1 bop e2

case class N(n: Double) extends Expr // e ::= n
case object Plus extends Bop         // bop ::= +

def isValue(e: Expr): Boolean = e match {
  case N(_) => true
  case _ => false
}

val e_oneplustwo = Binary(Plus, N(1), N(2))
```

```
defined trait Expr
defined trait Bop
defined class Binary
defined class N
defined object Plus
defined function isValue
e_oneplustwo: Binary = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0))
```

Back to the original example in this section, we are trying to specify how the expression $e_1 \texttt{+} e_2$ 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 rules defining the $e \Downarrow v$ judgment form:

$$
\textsc{EvalNum}\ \frac{}{n \Downarrow n}
\qquad
\textsc{EvalPlus}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{e_1 \texttt{+} e_2 \Downarrow n}
$$

The EvalNum rule is an axiom that states that an expression n evaluates to itself (as it is already a value).

The EvalPlus rule specifies how the expression $e_1 \texttt{+} e_2$ evaluates following our intuition above. Reading top-down, this rule says if we know that expression e_1 evaluates to a number n_1 and e_2 evaluates to n_2, then expression $e_1 \texttt{+} e_2$ evaluates to n where n is the addition of the n_1 and n_2.

Any evaluation rule can also be read bottom-up, which matches more closely to an implementation. For example, the above EvalPlus rule says, "To evaluate $e_1 \texttt{+} e_2$, evaluate e_1 to get a number n_1, evaluate e_2 to get a number n_2, and return the addition of those two numbers $n = n_1 + n_2$."

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 distinguished the meta-language "plus" for clarity, but often, the reader is asked to determine this distinction based on context. To be completely explicit, let us use an alternative notation for the abstract syntax:

$$\textsc{EvalPlus}\quad \frac{e_1 \Downarrow \texttt{N}(n_1) \qquad e_1 \Downarrow \texttt{N}(n_2) \qquad n = n_1 + n_2}{\texttt{Binary(Plus, } e_1\texttt{, } e_2\texttt{)} \Downarrow \texttt{N}(n)}$$

We see that these inference rules could translate to following `eval` implementation:

```
def eval(e: Expr): Expr = e match {
  // EvalNum
  case n @ N(_) => n
  // EvalPlus
  case Binary(Plus, e1, e2) => {
    val N(n1) = eval(e1)
    val N(n2) = eval(e2)
    val n = n1 + n2
    N(n)
  }
}

e_oneplustwo
val v_oneplustwo = eval(e_oneplustwo)
assert(v_oneplustwo == N(3))
```

```
defined function eval
```

```
res1_1: Binary = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0))
v_oneplustwo: Expr = N(n = 3.0)
```

As there could be slightly different code implementations that behave the same, the same is true for inference rules. For example, the following version of EvalPlus says the same thing without making an explicit "binding" of n:

$$\textsc{EvalPlus}\quad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \,\texttt{+}\, e_2 \Downarrow n_1 + n_2}$$

While we want a set of inference rules to define a semantics unambiguously, there are implementation choices. For example, defining `eval` as follows:

```
def eval(e: Expr): Double = e match {
  // EvalNum
  case N(n) => n
  // EvalPlus
  case Binary(Plus, e1, e2) => eval(e1) + eval(e2)
}

e_oneplustwo
val v_oneplustwo = eval(e_oneplustwo)
assert(v_oneplustwo == 3)
```

```
defined function eval
res2_1: Binary = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0))
v_oneplustwo: Double = 3.0
```

is also described by these inference rules.

We can imagine that we also add inference rules for other arithmetic operators in a similar manner (cf. Figure 18.2).

18.3 Dynamic Typing

Let us add boolean values to our JavaScripty variant:

$$\begin{array}{rrcl} \text{values} & v & ::= & b \\ \text{booleans} & b & & \end{array}$$

```
case class B(b: Boolean) extends Expr // e ::= b

def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) => true
  case _ => false
}

val e_true = B(true)
val e_trueplustwo = Binary(Plus, e_true, N(2))
```

```
defined class B
defined function isValue
e_true: B = B(b = true)
e_trueplustwo: Binary = Binary(bop = Plus, e1 = B(b = true), e2 = N(n = 2.0))
```

For the moment, we only add boolean literals and consider the following set of inference rules defining evaluation:

$$\textsc{EvalNum}\ \frac{}{n \Downarrow n} \qquad \textsc{EvalBool}\ \frac{}{b \Downarrow b} \qquad \textsc{EvalPlus}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \mathbin{\texttt{+}} e_2 \Downarrow n_1 + n_2}$$

```
def eval(e: Expr): Expr = e match {
  // EvalNum
  case n @ N(_) => n
  // EvalBool
  case b @ B(_) => b
  // EvalPlus
  case Binary(Plus, e1, e2) => {
    val N(n1) = eval(e1)
    val N(n2) = eval(e2)
    N(n1 + n2)
  }
}

e_oneplustwo
val v_oneplustwo = eval(e_oneplustwo)
assert(v_oneplustwo == N(3))

e_true
```

```
val v_true = eval(e_true)
assert(v_true == B(true))
```

```
defined function eval
res4_1: Binary = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0))
v_oneplustwo: Expr = N(n = 3.0)
res4_4: B = B(b = true)
v_true: Expr = B(b = true)
```

An alternative would be to have just one rule for values:

$$\textsc{EvalVal}\ \frac{}{v \Downarrow v} \qquad \textsc{EvalPlus}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \mathbin{\texttt{+}} e_2 \Downarrow n_1 + n_2}$$

where have rewritten EvalNum and EvalBool to EvalVal here to apply to all values v, including both numbers and booleans. We can reimplement `eval` to match these rules:

```
def eval(e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(v) => v
  // EvalPlus
  case Binary(Plus, e1, e2) => {
    val N(n1) = eval(e1)
    val N(n2) = eval(e2)
    N(n1 + n2)
  }
}

e_oneplustwo
val v_oneplustwo = eval(e_oneplustwo)
assert(v_oneplustwo == N(3))

e_true
val v_true = eval(e_true)
assert(v_true == B(true))
```

```
defined function eval
res5_1: Binary = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0))
v_oneplustwo: Expr = N(n = 3.0)
```

```
res5_4: B = B(b = true)
v_true: Expr = B(b = true)
```

Recall that a judgment form (e.g., $e \Downarrow v$) is an inductively-defined relation. A particular judgment holds, for example,

$$1 + 2 \Downarrow 3$$

when we can find a derivation for it (cf. Section 15.2).

It is essentially undefined behavior when a judgment does not hold. For example, with these rules, there is no derivation for the judgment

$$\mathbf{true} + 2 \Downarrow v$$

for any value v. In code, this might manifest as an exception:

```
e_trueplustwo
val v_trueplustwo = eval(e_trueplustwo)
```

We see that the particular issue is that $+$ can only apply to numbers in the EvalPlus rule: specifically, $n_1 + n_2$. When the operator does not apply to input values, this is called a *type error*. If we detect type error at run time, then this is called *dynamic typing*.

In particular, we do not fail haphazardly. Instead, we want to identify specifically the expression that has the type error:

```
case class DynamicTypeError(e: Expr) extends Exception {
  override def toString: String = s"TypeError: in expression $e"
}

def eval(e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(v) => v
  case Binary(Plus, e1, e2) => {
    (eval(e1), eval(e2)) match {
      // EvalPlus
      case (N(n1), N(n2)) => N(n1 + n2)
      // Otherwise, we have a type error.
      case _ => throw DynamicTypeError(e)
    }
  }
}
```

```
defined class DynamicTypeError
defined function eval
```

We introduce the exception type `DynamicTypeError` so that we can report the specific expression that has the type error.

```
e_trueplustwo
val v_trueplustwo = eval(e_trueplustwo)
```

18.4 Coercions

The EvalPlus rules above define semantics that does not match JavaScript because they require e_1 and e_2 in $e_1 \mathbin{\texttt{+}} e_2$ 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.

Suppose we want to extend the definition of the evaluation judgment form so that there is a derivation of the judgment

$$\mathbf{true} \mathbin{\texttt{+}} 2 \Downarrow v$$

for some value v. That is, we want to define type coercions so that we can apply $n_1 + n_2$ in EvalPlus.

Let us introduce a new judgment form for type coercions:

$$v \rightsquigarrow n$$

to say, "Value v coerces to number n." In the case that values are numbers or booleans (i.e., $v ::= n \mid b$), we define this judgment form for coercions as follows:

$$\boxed{v \rightsquigarrow n} \qquad \frac{}{n \rightsquigarrow n}\ \textsc{ToNumberNum} \qquad \frac{}{\mathbf{true} \rightsquigarrow 1}\ \textsc{ToNumberTrue} \qquad \frac{}{\mathbf{false} \rightsquigarrow 0}\ \textsc{ToNumberFalse}$$

that we implement with the `toNumber` function:

```
def toNumber(e: Expr): Double = {
  require(isValue(e))
  e match {
    // ToNumberNum
    case N(n) => n
    // ToNumberTrue
    case B(true) => 1
```

```
    // ToNumberFalse
    case B(false) => 0
  }
}
```

```
defined function toNumber
```

$$\frac{}{v \Downarrow v}\textsc{EvalVal} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \rightsquigarrow n_1 \quad v_2 \rightsquigarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2}\textsc{EvalPlus}$$

```
def eval(e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(v) => v
  // EvalPlus
  case Binary(Plus, e1, e2) => {
    val v1 = eval(e1)
    val v2 = eval(e2)
    N(toNumber(v1) + toNumber(v2))
  }
}

e_trueplustwo
val v_trueplustwo = eval(e_trueplustwo)
```

```
defined function eval
res10_1: Binary = Binary(bop = Plus, e1 = B(b = true), e2 = N(n = 2.0))
v_trueplustwo: Expr = N(n = 3.0)
```

We can imagine that we also add inference rules for other arithmetic and boolean operators in a similar manner, though we also need a judgment form for coercing into booleans $v \rightsquigarrow b$ (cf. Figure 18.2).

18.5 Variables

Let us consider extending the above language with variable uses and binding (as before in Chapter 14):

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & x \mid \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \\
\text{variables} & x, y & &
\end{array}
$$

```
case class Var(x: String) extends Expr                        // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr // e ::= const x = e1; e2

val e_const_i_two_trueplusi = ConstDecl("i", N(2), Binary(Plus, B(true), Var("i")))
```

```
defined class Var
defined class ConstDecl
e_const_i_two_trueplusi: ConstDecl = ConstDecl(
  x = "i",
  e1 = N(n = 2.0),
  e2 = Binary(bop = Plus, e1 = B(b = true), e2 = Var(x = "i"))
)
```

Because of variables, we need a slightly 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.

First, we need to "refactor" the EvalVal and EvalPlus rules to account for the additional value environment parameter:

$$
\textsc{EvalVal}\ \frac{}{E \vdash v \Downarrow v}
\qquad
\textsc{EvalPlus}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2}{E \vdash e_1 \texttt{+} e_2 \Downarrow n_1 + n_2}
$$

Reading the rules bottom-up, observe that we are simply passing the value environment E into recursive calls of $E \vdash e \Downarrow v$.

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

$$\text{value environments} \quad E, \mathit{env} \quad ::= \quad \cdot \mid E[x \mapsto v]$$

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 by induction on the structure of E:

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

While we give a syntax for value environments in the above to define them mathematically, we may choose to implement them in other ways. For example, we choose to represent value environments `Env` using the `Map[String, Expr]` data type from Scala standard library with `lookup` and `extend` functions:

```scala
type Env = Map[String, Expr]

val empty: Env = Map.empty

def lookup(env: Env, x: String): Expr = env(x)

def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined type Env
empty: Env = Map()
defined function lookup
defined function extend
```

Let us consider inference rules that define evaluating variable uses and variable binding:

$$
\frac{}{E \vdash x \Downarrow E(x)}\ \textsc{EvalVar}
\qquad
\frac{E \vdash e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{E \vdash \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \Downarrow v_2}\ \textsc{EvalConstDecl}
$$

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.

The EvalConstDecl rule is particularly interesting because we see explicitly that

$$\textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2$$

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

It is informative to see how the rules correspond to implementing an interpreter:

```
def eval(env: Env, e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(v) => v
  // EvalPlus
  case Binary(Plus, e1, e2) => {
    val v1 = eval(env, e1)
    val v2 = eval(env, e2)
    N(toNumber(v1) + toNumber(v2))
  }
  // EvalVar
  case Var(x) => lookup(env, x)
  // EvalConstDecl
  case ConstDecl(x, e1, e2) => {
    val v1 = eval(env, e1)
    eval(extend(env, x, v1), e2)
  }
}

e_const_i_two_trueplusi
val v_const_i_two_trueplusi = eval(empty, e_const_i_two_trueplusi)
assert(v_const_i_two_trueplusi == N(3))
```

```
defined function eval
res13_1: ConstDecl = ConstDecl(
  x = "i",
  e1 = N(n = 2.0),
  e2 = Binary(bop = Plus, e1 = B(b = true), e2 = Var(x = "i"))
)
v_const_i_two_trueplusi: Expr = N(n = 3.0)
```

18.6 JavaScripty: Variables, Numbers, and Booleans

Figure 18.1 describes the syntax of a JavaScripty variant with variables, numbers, and booleans 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, a variable binding expression, value literals, unary operator expressions, binary operator expressions, and a conditional if-then-else expression. Value literals v can be numbers (double-precision floating point) and booleans. This set of arithmetic and logic expressions is the usual core of any programming language. Strings, side effects, and functions are notably missing.

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & x \mid \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \mid v \mid uop\, e_1 \mid e_1\ bop\ e_2 \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \\
\text{values} & v & ::= & n \mid b \\
\text{unary operators} & uop & ::= & \texttt{-} \mid \texttt{!} \\
\text{binary operators} & bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\text{variables} & x & &
\end{array}
$$

Figure 18.1: Syntax of JavaScripty with variables, numbers, and booleans (i.e., binding, arithmetic, and logic).

We give in Figure 18.2, a big-step operational semantics for the JavaScripty variant defined above. That is, we give inference rules that define the evaluation judgment form: $E \vdash e \Downarrow v$.

In rule EvalArith, we lump all of the arithmetic operators `+`, `-`, `*`, and `/` together. We abuse notation here slightly by treating the *bop* as the corresponding meta-language operator in $n_1\ bop\ n_2$.

It is informative to study the complete set of inference rules and think about how the rules correspond to implementing an interpreter.

Observe that the EvalAndTrue, EvalAndFalse, EvalOrTrue, EvalOrFalse, EvalIfTrue, and EvalIfFalse rules use coercions to booleans but do not necessarily return boolean. The EvalEquality rule shows that equality `===` and disequality `!==` apply to any values without coercion, while the EvalInequality rule says that inequalites apply after coercing to numbers.

Does this reveal any bugs in your implementation in the previous lab?

18.7 JavaScripty: Strings

Let us consider extending our big-step semantics for JavaScripty with string values.

$$
\begin{array}{rrcl}
\text{values} & v & ::= & str \\
\text{strings} & str & &
\end{array}
$$

We have seen that in JavaScript some operators are overloaded for numbers and strings (e.g., + for string concatenation and <, <=, >, and >= for string comparisons).

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

$$\textsc{EvalVar}\ \frac{}{E \vdash x \Downarrow E(x)}$$

$$\textsc{EvalConstDecl}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{E \vdash \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \Downarrow v_2}$$

$$\textsc{EvalVal}\ \frac{}{E \vdash v \Downarrow v}$$

$$\textsc{EvalNeg}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow n_1}{E \vdash \texttt{-}\, e_1 \Downarrow -n_1}$$

$$\textsc{EvalNot}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow b_1}{E \vdash \texttt{!}\ e_1 \Downarrow \neg b_1}$$

$$\textsc{EvalArith}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2 \qquad bop \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{E \vdash e_1\ bop\ e_2 \Downarrow n_1\ bop\ n_2}$$

$$\textsc{EvalAndTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{true} \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ \texttt{\&\&}\ e_2 \Downarrow v_2}$$

$$\textsc{EvalAndFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{false}}{E \vdash e_1\ \texttt{\&\&}\ e_2 \Downarrow v_1}$$

$$\textsc{EvalOrTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{true}}{E \vdash e_1\ \texttt{||}\ e_2 \Downarrow v_1}$$

$$\textsc{EvalOrFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{false} \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ \texttt{||}\ e_2 \Downarrow v_2}$$

$$\textsc{EvalIfTrue}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{true} \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_2}$$

$$\textsc{EvalIfFalse}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad v_1 \rightsquigarrow \textbf{false} \qquad E \vdash e_3 \Downarrow v_3}{E \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_3}$$

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

$$\textsc{EvalInequality}\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{E \vdash e_1\ bop\ e_2 \Downarrow n_1\ bop\ n_2}$$

Figure 18.2: Big-step operational semantics of JavaScripty with variables, numbers, and booleans (i.e., binding, arithmetic, and logic).

Where we need to do some detective work is when these string operations apply with type coercions. When is a value coerced into a number versus a string?

Let's consider string concatenation +:

```
"hello" + 3
3 + "hello"
```

It appears that string concatenation applies if either operand is a string *str*.

How about string comparison?

```
0 < "1"
0 < "hello"
"0" < 1
"a" < "ab"
```

It appears that string comparison only applies if both operands are strings.

To capture these observations, we replace rules EvalArith and EvalInequality in Figure 18.2 with the following rules shown in Figure 18.3.

The rules EvalPlusString_1 and EvalPlusString_2 apply string concatenation if either e_1 or e_2 evaluate to strings, whereas EvalInequalityString applies if and only if both e_1 and e_2 evaluate to strings. The other rules carefully state that number operations apply in all other cases.

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

$$\textsc{EvalPlusNumber}\quad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \neq str_1 \qquad v_2 \neq str_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2}{E \vdash e_1 \texttt{+} e_2 \Downarrow n_1 + n_2}$$

$$\textsc{EvalPlusString}_1\quad \frac{E \vdash e_1 \Downarrow str_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_2 \rightsquigarrow str_2}{E \vdash e_1 \texttt{+} e_2 \Downarrow str_1 str_2}$$

$$\textsc{EvalPlusString}_2\quad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow str_2 \qquad v_1 \rightsquigarrow str_1}{E \vdash e_1 \texttt{+} e_2 \Downarrow str_1 str_2}$$

$$\textsc{EvalArith}\quad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2 \qquad bop \in \{\texttt{-}, \texttt{*}, \texttt{/}\}}{E \vdash e_1\ bop\ e_2 \Downarrow n_1\ bop\ n_2}$$

$$\textsc{EvalInequalityNumber}_1\quad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_1 \neq str_1 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{E \vdash e_1\ bop\ e_2 \Downarrow n_1\ bop\ n_2}$$

$$\textsc{EvalInequalityNumber}_2\quad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_2 \neq str_2 \qquad v_1 \rightsquigarrow n_1 \qquad v_2 \rightsquigarrow n_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{E \vdash e_1\ bop\ e_2 \Downarrow n_1\ bop\ n_2}$$

$$\textsc{EvalInequalityString}\quad \frac{E \vdash e_1 \Downarrow str_1 \qquad E \vdash e_2 \Downarrow str_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{E \vdash e_1\ bop\ e_2 \Downarrow str_1\ bop\ str_2}$$

Figure 18.3: Updating the big-step semantics of JavaScripty with variables, numbers, and booleans (Figure 18.2) to include strings.

19 Functions and Dynamic Scoping

19.1 Functions Are Values

A code abstraction mechanism like functions is essential to what we would consider a programming language. Let us consider our object language JavaScripty with variables and base types (Section 18.6)

```
trait Expr                                                        // e
case class N(n: Double) extends Expr                              // e ::= n
case class Var(x: String) extends Expr                            // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr // e ::= const x = e1; e2
```

```
defined trait Expr
defined class N
defined class Var
defined class ConstDecl
```

and extend it with function values:

$$
\begin{array}{rrcl}
\text{values} & v & ::= & \texttt{(}x\texttt{)} \Rightarrow e_1 \\
\text{expressions} & e & ::= & e_1\texttt{(}e_2\texttt{)} \\
\text{variables} & x & &
\end{array}
\tag{19.1}
$$

```
case class Fun(x: String, e1: Expr) extends Expr // e ::= (x) => e1
case class Call(e1: Expr, e2: Expr) extends Expr // e ::= e1(e2)
```

```
defined class Fun
defined class Call
```

A function literal $\texttt{(}x\texttt{)} \Rightarrow e_1$ has a *formal parameter* x and function body e_1. Note that a function literal is a value because it is an expression that cannot reduce any further until it is called. A function call expression $e_1\texttt{(}e_2\texttt{)}$ expects e_1 to evaluate to a function literal and e_2 to a value (called the *actual argument*) to use for the formal parameter in evaluating the function body.

```scala
def isValue(e: Expr): Boolean = e match {
  case N(_) | Fun(_, _) => true
  case _ => false
}

type Env = Map[String, Expr]
val empty: Env = Map.empty
def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined function isValue
defined type Env
empty: Env = Map()
defined function lookup
defined function extend
```

```scala
case class DynamicTypeError(e: Expr) extends Exception {
  override def toString: String = s"TypeError: in expression $e"
}
```

```
defined class DynamicTypeError
```

Functions as we have here are called *first-class functions* because they are values that can be passed and returned like any other type of values (e.g., numbers, booleans, strings).

For simplicity and to focus in on their essence, all functions have exactly one parameter and are anonymous and cannot be recursive. Since functions are first-class values, we can define multi-parameter functions via currying (i.e., functions that return functions).

19.2 Dynamic Scoping

We first try to implement function call in the most straightforward way. What we will discover is that we have made a historical mistake and have ended up with a form of dynamic scoping.

The evaluation judgment form $E \vdash e \Downarrow v$ says, “In value environment E, expression e evaluates to value v.” We extend the definition of this judgment form for function call $e_1\texttt{(}e_2\texttt{)}$ with the EvalCall rule as follows:

$$\boxed{E \vdash e \Downarrow v} \qquad \frac{\begin{array}{c}\textsc{EvalCall} \\ E \vdash e_1 \Downarrow (x) \Rightarrow e' \qquad E \vdash e_2 \Downarrow v_2 \qquad E[x \mapsto v_2] \vdash e' \Downarrow v'\end{array}}{E \vdash e_1(e_2) \Downarrow v'}$$

Figure 19.1: Defining evaluation of a function call expression that "accidentally" implements dynamic scoping.

This rule says that we evaluate e_1 to a function value $(x) \Rightarrow e'$ and evaluate e_2 to a value v_2. Then, we extend the environment to bind the formal parameter x to the actual argument v_2 to evaluate the function body expression e' to a value v'.

First, observe that we can only evaluate a call expression $e_1(e_2)$ if e_1 evaluates to a function. It is a type error if e_1 does not evaluate to a function. This is indeed one of the few run-time errors in JavaScript.

Let us implement this judgment form:

```scala
def eval(env: Env, e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(e) => v
  // EvalVar
  case Var(x) => lookup(env, x)
  // EvalConstDecl
  case ConstDecl(x, e1, e2) => {
    val v1 = eval(env, e1)
    eval(extend(env, x, v1), e2)
  }
  // EvalCall
  case Call(e1, e2) => eval(env, e1) match {
    case Fun(x, e) => {
      val v2 = eval(env, e2)
      eval(extend(env, x, v2), e)
    }
    case _ => throw DynamicTypeError(e)
  }
}
```

```
defined function eval
```

```scala
eval(empty, Call(N(1), N(2)))
```

Now, recall that the scope of a variable in most languages is a static property—for any variable use, the variable binding site it references does not depend on program execution. Dynamic scoping is thus when the binding site of the variable being used *does* depend on program execution.

If we study EvalCall closely (Figure 19.1), we can get a hint of the "accidental" appearance of dynamic scoping. The function body expression e' may have free variable uses that under static scoping should reference variables where the function is defined, but it is being evaluated in a value environment E that is potentially very different from the value environment when it was defined. Can we come up with example that exhibits dynamic scoping with this EvalCall rule?

Let us reimplement this judgment for with some instrumentation to show derivations:

```
def eval(level: Int, env: Env, e: Expr): Expr = {
  val indent = " " * level
  val v = e match {
    // EvalVal
    case v if isValue(e) => {
      println(s"\n${indent}------------------------ EvalVal")
      v
    }
    // EvalVar
    case Var(x) => {
      val v = lookup(env, x)
      println(s"\n${indent}------------------------ EvalVar")
      v
    }
    // EvalConstDecl
    case ConstDecl(x, e1, e2) => {
      val v1 = eval(level, env, e1)
      val v2 = eval(level + 6, extend(env, x, v1), e2)
      println(s"${indent}------------------------ EvalConstDecl")
      v2
    }
    // EvalCall
    case Call(e1, e2) => {
      eval(level, env, e1) match {
        case Fun(x, e) => {
          val v2 = eval(level + 4, env, e2)
          val v = eval(level + 8, extend(env, x, v2), e)
          println(s"${indent}------------------------ EvalCall")
          v
```

```
        }
        case _ => throw DynamicTypeError(e)
      }
    }
  }
  println(s"${indent}$env   $e   $v")
  v
}

def eval(e: Expr): Expr = eval(0, empty, e)
```

```
defined function eval
defined function eval
```

We use indention to indicate the different premises of a multi-premise rule:

```
eval( Call(Fun("x", Var("x")), N(2)) )
```

```
------------------------ EvalVal
Map()   Fun(x,Var(x))   Fun(x,Var(x))

    ------------------------ EvalVal
    Map()   N(2.0)   N(2.0)

        ------------------------ EvalVar
        Map(x -> N(2.0))   Var(x)   N(2.0)
------------------------ EvalCall
Map()   Call(Fun(x,Var(x)),N(2.0))   N(2.0)

res7: Expr = N(n = 2.0)
```

To construct an example that exhibits dynamic scoping, we define a function that under static scoping references an outer variable binding that gets shadowed by a variable when its body is later evaluated:

```
const x = 1;
const g = (y) => x;
((x) => g(2))(3)
```

Under static scoping, the variable use x in the function defined on line 2 references the variable binding of x on line 1 and should always return 1. However, using EvalCall in Figure 19.1, it ends up referencing the variable binding at line 3 and returning 3:

```
val e_dynamicScoping =
  ConstDecl("x", N(1),
  ConstDecl("g", Fun("y", Var("x")),
  Call(Fun("x", Call(Var("g"), N(2))), N(3))))

val v_dynamicScoping = eval(e_dynamicScoping)
```

```
------------------------ EvalVal
Map()   N(1.0)   N(1.0)

      ------------------------ EvalVal
      Map(x -> N(1.0))   Fun(y,Var(x))   Fun(y,Var(x))

            ------------------------ EvalVal
            Map(x -> N(1.0), g -> Fun(y,Var(x)))   Fun(x,Call(Var(g),N(2.0)))   Fun(x,Call(Va

                ------------------------ EvalVal
                Map(x -> N(1.0), g -> Fun(y,Var(x)))   N(3.0)   N(3.0)

                    ------------------------ EvalVar
                    Map(x -> N(3.0), g -> Fun(y,Var(x)))   Var(g)   Fun(y,Var(x))

                        ------------------------ EvalVal
                        Map(x -> N(3.0), g -> Fun(y,Var(x)))   N(2.0)   N(2.0)

                            ------------------------ EvalVar
                            Map(x -> N(3.0), g -> Fun(y,Var(x)), y -> N(2.0))   Var(x)   N(3.
                    ------------------------ EvalCall
                    Map(x -> N(3.0), g -> Fun(y,Var(x)))   Call(Var(g),N(2.0))   N(3.0)
            ------------------------ EvalCall
            Map(x -> N(1.0), g -> Fun(y,Var(x)))   Call(Fun(x,Call(Var(g),N(2.0))),N(3.0))   N
      ------------------------ EvalConstDecl
      Map(x -> N(1.0))   ConstDecl(g,Fun(y,Var(x)),Call(Fun(x,Call(Var(g),N(2.0))),N(3.0)))
------------------------ EvalConstDecl
Map()   ConstDecl(x,N(1.0),ConstDecl(g,Fun(y,Var(x)),Call(Fun(x,Call(Var(g),N(2.0))),N(3.0)))

e_dynamicScoping: ConstDecl = ConstDecl(
```

```
  x = "x",
  e1 = N(n = 1.0),
  e2 = ConstDecl(
    x = "g",
    e1 = Fun(x = "y", e1 = Var(x = "x")),
    e2 = Call(
      e1 = Fun(x = "x", e1 = Call(e1 = Var(x = "g"), e2 = N(n = 2.0))),
      e2 = N(n = 3.0)
    )
  )
)
v_dynamicScoping: Expr = N(n = 3.0)
```

19.3 Closures

The example that exhibits dynamic scoping suggests some possible fixes to implement static scoping. We observe that a free variable use in a function body references a variable binding at the time the function is defined, not when the function body is evaluated. This suggests that a function body should be evaluated in the value environment when it is defined.

A *closure* is exactly this $(x) \Rightarrow e_1[E]$ —a pair consisting of a function literal $(x) \Rightarrow e_1$ and its value environment at the time of its definition E. Function values are now closures:

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & (x) \Rightarrow e_1 \mid e_1(e_2) \\
\text{values} & v & ::= & (x) \Rightarrow e_1[E] \\
\text{variables} & x & &
\end{array}
$$

```scala
case class Closure(fun: Fun, env: Env) extends Expr // e ::= (x) => e1[E]

def isValue(e: Expr): Boolean = e match {
  case N(_) | Closure(_, _) => true
  case _ => false
}

type Env = Map[String, Expr]
val empty: Env = Map.empty
def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined class Closure
defined function isValue
defined type Env
empty: Env = Map()
defined function lookup
defined function extend
```

We add a rule EvalFun that says that evaluating a function literal creates a closure. Then, evaluating the function body e' in EvalCall uses the value environment from the closure E':

$$\boxed{E \vdash e \Downarrow v} \qquad \frac{\textsc{EvalFun}}{E \vdash (x) \Rightarrow e \Downarrow (x) \Rightarrow e[E]}$$

$$\textsc{EvalCall}\ \frac{E \vdash e_1 \Downarrow (x) \Rightarrow e'[E'] \qquad E \vdash e_2 \Downarrow v_2 \qquad E'[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1(e_2) \Downarrow v'}$$

```scala
def eval(env: Env, e: Expr): Expr = e match {
  // EvalVal
  case v if isValue(e) => v
  // EvalVar
  case Var(x) => lookup(env, x)
  // EvalConstDecl
  case ConstDecl(x, e1, e2) => {
    val v1 = eval(env, e1)
    eval(extend(env, x, v1), e2)
  }
  // EvalFun
  case f @ Fun(x, e) => Closure(f, env)
  // EvalCall
  case Call(e1, e2) => eval(env, e1) match {
    case Closure(Fun(x, e_), env_) => {
      val v2 = eval(env, e2)
      eval(extend(env_, x, v2), e_)
    }
    case _ => throw DynamicTypeError(e)
  }
}

val v_dynamicScopingFixed = eval(empty, e_dynamicScoping)
```

```
defined function eval
v_dynamicScopingFixed: Expr = N(n = 1.0)
```

19.4 Substitution

This observation about "accidental" dynamic scoping also suggests another strategy for implementing static scoping. We avoid the chance of dynamic scoping if we avoid free variables, that is, we maintain the invariant that we evaluate only closed expressions. It is possible to maintain this invariant by using *substitution*.

We write $[e_1/x_1]e$ for a scope-respecting substitution of expression e_1 for free variable uses of x_1 in expression e. This function can be defined by induction on the structure of e, though it does require some care to respect binding and scope. In particular, substitution applies to *free variable uses* of x_1 and must be *capture-avoiding* (i.e., avoiding the capture of any free variable uses in e_1).

Given a scope-respecting substitution, we define an evaluation judgment for only closed expressions again $e \Downarrow v$.

We return to the case where function literals are values (though they will be closed).

$$\begin{array}{rrcl}
\text{values} & v & ::= & (x) \Rightarrow e' \\
\text{expressions} & e & ::= & e_1(e_2) \\
\text{variables} & x & &
\end{array}$$

In EvalConstDecl and EvalCall, we can see that we use substitution to effectively "apply" the value environment eagerly one-binding-at-a-time to the expression so that we never need to reify it:

$$\boxed{e \Downarrow v} \qquad \text{no EvalVar rule} \qquad \frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v_2}{\textbf{const}\ x = e_1\text{;}\ e_2 \Downarrow v_2}\ \textsc{EvalConstDecl} \qquad \text{no EvalFun rule}$$

$$\frac{e_1 \Downarrow (x) \Rightarrow e' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e' \Downarrow v'}{e_1(e_2) \Downarrow v'}\ \textsc{EvalCall}$$

There is no EvalVar rule because variable uses are replaced by the values their bound when the binding site is evaluated. And there is no EvalFun because function literals are again function values.

19.5 Recursive Functions

Thus far we have considered anonymous function literals $(y) \Rightarrow e'$ that cannot be recursive. To allow for recursive function definitions, we enrich the function expression `Fun` with a parameter for an optional variable name to refer to itself:

```
case class Fun(xopt: Option[String], y: String, e1: Expr) extends Expr // e ::= xopt(y) => e1
```

```
defined class Fun
```

Correspondingly, let us extend our abstract syntax for JavaScripty as follows:

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & x^?(y) \Rightarrow e_1 \mid e_1(e_2) \\
\text{optional variables} & x^? & ::= & x \mid \varepsilon \\
\text{variables} & x & &
\end{array}
$$

Observe that we define $x^?$ as the non-terminal for an optional variable.

When a function expression has a name $x(y) \Rightarrow e'$, then it is can be recursive. In particular, variable x is an additional formal parameter, and the function body e' may have free variable uses of x. The variable x gets bound to itself (i.e., the function value for $x(y) \Rightarrow e'$) on a function call.

In terms of the `Expr` representation, the `xopt` can be `Some(` x `)` corresponding to $x(y) \Rightarrow e_1$ or `None` corresponding to $(y) \Rightarrow e_1$.

Note that we consider $x^?(y) \Rightarrow e_1$ abstract syntax. In particular, $x(y) \Rightarrow e_1$ is not valid concrete syntax in JavaScript, as we discuss next.

Exercise 19.1 (Big-Step Semantics for Potentially-Recursive Functions)**.** Give a rule EvalCallRec that defines function call to a named-function literal $x(y) \Rightarrow e_1$. Either extend the evaluation judgment form for potentially-open expressions with value environments $E \vdash e \Downarrow v$ using closures or the evaluation judgment form for closed expressions $e \Downarrow v$ or both.

19.6 JavaScripty: Concrete Syntax: Functions

Recall from Section 14.11 that in the concrete syntax, **const**-bindings are declarations (and not expressions).

$$
\begin{array}{rrcl}
\text{declarations} & d & ::= & \textbf{const}\ x\ \texttt{=}\ e\,\texttt{;} \mid s \mid d_1\ d_2 \mid \varepsilon \\
\text{statements} & s & ::= & e\texttt{;} \mid \texttt{\{}\ d\ \texttt{\}} \mid \texttt{;} \\
\text{expressions} & e & ::= & \texttt{(}e\texttt{)} \mid \cdots \mid e_1\texttt{(}e_2\texttt{)} \mid \texttt{(}x\texttt{)}\ \texttt{=>}\ e \\
 & & \mid & \texttt{(}x\texttt{)}\ \texttt{=>}\ \texttt{\{}\ \mathit{body}\ \texttt{\}} \mid \textbf{function}\ x^{?}\texttt{(}y\texttt{)}\ \texttt{\{}\ \mathit{body}\ \texttt{\}} \\
\text{variables} & x, y & &
\end{array}
$$

To accommodate declarations in function bodies, JavaScript has additional concrete syntax for function literals (in addition to $\texttt{(}x\texttt{)} \texttt{=>} e$):

$$\texttt{(}x\texttt{)}\ \texttt{=>}\ \texttt{\{}\ \mathit{body}\ \texttt{\}} \quad \text{and} \quad \textbf{function}\ x^{?}\texttt{(}y\texttt{)}\ \texttt{\{}\ \mathit{body}\ \texttt{\}}$$

In both of these variants, a function body *body* is surrounded by curly braces (i.e., `{ }`) and consists of a declaration d (e.g., for **const**-bindings) followed by a **return** keyword, a return-expression e, and a trailing `;`:

$$\text{function bodies} \quad \mathit{body} \quad ::= \quad d\ \textbf{return}\ e\,\texttt{;}$$

Note that JavaScript permits function bodies that leave out the **return** keyword or the return-expression e. When the **return** keyword is left out, the meaning of a function body is to implicitly return **undefined**.

The **function** keyword syntax may have a function name but whose definition must be a function body $\texttt{\{}\ \mathit{body}\ \texttt{\}}$.

20 Exercise: Big-Step Operational Semantics

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

- how to read a formal specification of a language semantics;
- how dynamic scoping arises; and
- big-step interpretation.

Instructions

This assignment asks you to write Scala code. There are restrictions associated with how you can solve these problems. Please pay careful heed to those. If you are unsure, ask the course staff.

Note that `???` indicates that there is a missing function or code fragment that needs to be filled in. Make sure that you remove the `???` and replace it with the answer.

Use the test cases provided to test your implementations. You are also encouraged to write your own test cases to help debug your work. However, please delete any extra cells you may have created lest they break an autograder.

Imports

```
import $ivy.$                                        , org.scalatest._, events._, flatspec._
```

```
defined function report
defined function assertPassed
defined function passed
defined function test
```

Listing 20.1 `org.scalatest._`

```
// Run this cell FIRST before testing.
import $ivy.`org.scalatest::scalatest:3.2.19`, org.scalatest._, events._, flatspec._
def report(suite: Suite): Unit = suite.execute(stats = true)
def assertPassed(suite: Suite): Unit =
  suite.run(None, Args(new Reporter {
    def apply(e: Event) = e match {
      case e @ (_: TestFailed) => assert(false, s"${e.message} (${e.testName})")
      case _ => ()
    }
  }))
def passed(points: Int): Unit = {
  require(points >=0)
  if (points == 1) println("***   Tests Passed (1 point) ***")
  else println(s"***   Tests Passed ($points points) ***")
}
def test(suite: Suite, points: Int): Unit = {
  report(suite)
  assertPassed(suite)
  passed(points)
}
```

20.1 A Big-Step Javascripty Interpreter

We now have the formal tools to specify exactly how a JavaScripty program should behave. Unless otherwise specified, we continue to try to match JavaScript semantics, though we are no longer beholden to it. Thus, it is still useful to write little test JavaScript programs and see how the test should behave.

In this exercise, we extend JavaScripty with functions. We try to implement the `eval` function in the most straightforward way. What we will discover is that we have made a historical mistake and have ended up with a form of *dynamic scoping*.

For the purpose of this exercise, we will limit the scope of JavaScripty by restricting expression forms and simplifying semantics as appropriate for pedagogical purposes. In particular, we simplify the semantics by no longer performing implicit type coercions.

20.1.1 Syntax

We consider the following *abstract syntax* for this exercise. Note that new constructs for functions are highlighted.

$$
\begin{array}{rcll}
\text{expressions} & e & ::= & n \mid b \mid e_1 \; bop \; e_2 \mid e_1 \; \texttt{?} \; e_2 \; \texttt{:} \; e_3 \mid x \mid \textbf{const} \; x \; \texttt{=} \; e_1 \texttt{;} \; e_2 \\
 & & \mid & \texttt{(}x\texttt{)} \Rightarrow e_1 \mid e_1 \texttt{(}e_2\texttt{)} \\
\text{values} & v & ::= & n \mid b \mid \texttt{(}x\texttt{)} \Rightarrow e_1 \\
\text{binary operators} & bop & ::= & \texttt{+} \mid \texttt{===} \mid \texttt{!==} \\
\text{variables} & x & & \\
\text{numbers} & n & & \\
\text{booleans} & b & &
\end{array}
$$

Observe that we consider only base values numbers n and booleans b and have significantly reduced the number of expression forms we consider.

Like in the book chapter, all functions are one argument functions for simplicity.

20.2 Dynamic Scoping Test

Exercise 20.1 (5 points)**.** Write a JavaScript program that behaves differently under dynamic scoping versus static scoping (and does not crash). This will get us used to the syntax, while providing a crucial test case for our interpreter.

Edit this cell:

```
const x = 10;
const f = (a) => {
  ???
}
const g = (b) => {
  ???
}
g(-1)
```

Explain in 1-2 sentences why you think this program would behave differently under dynamic scoping versus static scoping.

Edit this cell:

???

Notes

- We are using **const** to *name* functions, that is, we are binding an expression, which is a function, to a variable. This binding allows us to get it later, but it *does not* allow us call it inside the function definition (i.e., recursion).
- We are providing a throw-away parameter to our function because according to our syntax functions have exactly one parameter.
- As noted above, we are simplifying some semantics in this exercise compared with the previous lab: implicit type coercions work in JavaScript and in the previous lab, but you will not include them in your implementation on this homework. Therefore, *your test case cannot have any implicit type conversions.*
- In order to execute the program, you will need to switch your kernal to Deno, the Javascript kernel for Jupyter.

20.3 Reading an Operational Semantics

In this homework, we start to see specifications of programming language semantics. A big-step operational semantics of this small fragment of JavaScripty is given below. Except perhaps for the assigned reading, this figure (Figure 20.1) may be one of the first times that you are reading a formal semantics of a programming language. It may seem daunting at first, but it will be become easier with practice. This homework is such an opportunity to practice.

$$\boxed{E \vdash e \Downarrow v} \qquad \frac{}{E \vdash x \Downarrow E(x)}\;\textsc{EvalVar} \qquad \frac{E \vdash e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{E \vdash \textbf{const}\ x \texttt{ = } e_1 \texttt{;}\ e_2 \Downarrow v_2}\;\textsc{EvalConstDecl} \qquad \frac{}{E \vdash v \Downarrow v}\;\textsc{EvalVal}$$

$$\frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2}{E \vdash e_1 \texttt{ + } e_2 \Downarrow n_1 + n_2}\;\textsc{EvalPlusNumber} \qquad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{E \vdash e_1\ bop\ e_2 \Downarrow v_1\ bop\ v_2}\;\textsc{EvalEquality}$$

$$\frac{E \vdash e_1 \Downarrow \textbf{true} \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \Downarrow v_2}\;\textsc{EvalIfTrue} \qquad \frac{E \vdash e_1 \Downarrow \textbf{false} \qquad E \vdash e_3 \Downarrow v_3}{E \vdash e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \Downarrow v_3}\;\textsc{EvalIfFalse}$$

Figure 20.1: A big-step operational semantics of a fragment of JavaScripty with some arithmetic and logic expressions, as well as variable binding. We define the judgment form $E \vdash e \Downarrow v$, which says informally, “In value environment E, expression e evaluates to value v.” This relation has three parameters: E, e, and v. You can see the other parts of the judgment form as simply punctuation.

A value environment E is a finite map from variables x to values v that we write as follows:

$$\text{value environments} \quad E, \mathit{env} \quad ::= \quad \cdot \mid E[x \mapsto v]$$

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.

A formal semantics enables us to describe the semantics of a programming language clearly and concisely. The initial barrier is getting used to the meta-language of judgment forms and inference rules. However, once you cross that barrier, you will see that we are telling you exactly how to implement the interpreter—it will almost feel like cheating!

20.3.1 Strings

Exercise 20.2 (5 points)**.** Suppose that we extend the above language with strings *str* and a string concatenation e_1 + e_2 expression (like in JavaScript). Consider the following inference rule for the evaluation judgment form:

$$\frac{\textsc{EvalPlusString1} \atop E \vdash e_1 \Downarrow \mathit{str}_1 \qquad E \vdash e_2 \Downarrow v_2 \qquad v_2 \rightsquigarrow \mathit{str}_2}{E \vdash e_1 \texttt{+} e_2 \Downarrow \mathit{str}_1\mathit{str}_2}$$

Explain in 1-2 sentences what EvalPlusString1 is stating.

Edit this cell:

???

Notes

- The $v \rightsquigarrow \mathit{str}$ judgment form says that value v coerces to string *str*.

Exercise 20.3 (5 points)**.** Let us define rules that specify evaluation of the expression e_1 + e_2 just like in JavaScript. Give the other rule $\textsc{EvalPlusString}_2$ that concatenates strings in the case that e_2 evaluates to a string.

Edit this cell:

???

Explain in 1-2 sentences why you need EvalPlusString1 and EvalPlusString2 together for interpreting string concatenation like in JavaScript.

Edit this cell:

???

Notes

You may give the rule in LaTeX math or as plain text (ascii art) approximating the math rendering. For example,

```
EvalPlusString1
E |- e1 vv str1    E |- e2 vv v2    v2 ~~> str2
-----------------------------------------------
E |- e1 + e2 vv str1 str2
```

The LaTeX code for the rendered EvalPlusString1 rule above is as follows:

```
\inferrule[EvalPlusString1]{
  E \vdash e_1 \Downarrow \mathit{str}_1
  \and
  E \vdash e_2 \Downarrow v_2
  \and
  v_2 \rightsquigarrow \mathit{str}_2
}{
  E \vdash e_1 \mathbin{\texttt{+}} e_2 \Downarrow \mathit{str}_1 \mathit{str}_2
}
```

20.3.2 Functions

The inference rule defining evaluation of a function call (that accidentally results in dynamic scoping) is as follows:

Exercise 20.4 (5 points)**.** To continue this warm up and guide our implementation of these inference rules, write out what EvalCall is stating.

Edit this cell:

???

$$\frac{E \vdash e_1 \Downarrow (x) \Rightarrow e' \qquad E \vdash e_2 \Downarrow v_2 \qquad E[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1(e_2) \Downarrow v'}\ \textsc{EvalCall}$$

Figure 20.2

20.4 Implementing from Inference Rules

20.4.1 Abstract Syntax

In the following, we build up to implementing an `eval` function:

```
def eval(env: Env, e: Expr): Expr
```

This `eval` function directly corresponds the the evaluation judgment: $E \vdash e \Downarrow v$, which is the operational semantics defined above. It takes as input a value environment E and an expression e and returns a value v.

Below is the `Expr` type defining our abstract syntax tree in Scala. If you haven't already, switch back to the Scala kernel and then run the two cells below.

```
trait Expr // e ::=

case class Var(x: String) extends Expr                              // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr // e ::= const x = e1; e2

case class N(n: Double) extends Expr  // e ::= n
case class B(b: Boolean) extends Expr // e ::= b

trait Bop // bop ::=
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr // e ::= e1 bop b2

case object Plus extends Bop // bop ::= +
case object Eq extends Bop   // bop ::= ===
case object Ne extends Bop   // bop ::= !==

case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr // e ::= e1 ? e2 : e3

case class Fun(x: String, e1: Expr) extends Expr // e ::= (x) => e1
case class Call(e1: Expr, e2: Expr) extends Expr // e ::= e1(e2)
```

```
defined trait Expr
defined class Var
defined class ConstDecl
defined class N
defined class B
defined trait Bop
defined class Binary
defined object Plus
defined object Eq
defined object Ne
defined class If
defined class Fun
defined class Call
```

Numbers n, booleans b, and functions $(x) \Rightarrow e_1$ are values, and we represent a value environment E as a `Map[String, Expr]`:

```
def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) | Fun(_, _) => true
  case _ => false
}

type Env = Map[String, Expr]
val empty: Env = Map()
def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined function isValue
defined type Env
empty: Env = Map()
defined function lookup
defined function extend
```

Exercise 20.5 (5 points)**.** Now that we have the AST type `Expr` defined, take your JavaScripty test program from Exercise 20.1 and write out the AST it would parse to. This will serve as a test for your implementation.

Notes

- Recall the difference between concrete and abstract syntax. Your AST here will use the abstract syntax and be of type `Expr` defined above. Therefore, the AST nodes you write in can only be constructors of `Expr`. For example, the **return** keyword is in the concrete syntax but not a constructor of `Expr`.

20.4.2 Variables, Numbers, and Booleans

Exercise 20.6 (10 points)**.** Implement `eval` the evaluation judgment form $E \vdash e \Downarrow v$ for all rules except EvalCall shown in Figure 20.1. It should be noted that this implementation should very similar to your implementation of `eval` in previous lab.

Notes

- It is most beneficial to first implement `eval` from scratch by referencing the rules shown in Figure 20.1.
- After you implement `eval` here by following the rules, it may then be informative to compare with your implementation from the previous lab (that was for a larger language and with implicit type coercions).
- You will have unmatched cases (i.e., there are no corresponding rules), which you can leave unimplemented with `???` or the potential for `MatchError`.

Tests

20.4.3 Functions

Exercise 20.7 (10 points)**.** Extend your implementation with functions. On function calls, you need to extend the environment for the formal parameter. Begin with what you have from Exercise 20.6.

Notes

- This question is asking you to implement EvalCall.
- Do not worry yet about dynamic type errors, so this will still have some `???`s or have the possibility of `MatchError`s.

Tests

20.4.4 Dynamic Typing

In the previous lab, all expressions could be evaluated to something (because of conversions). With functions, we encounter one of the very few run-time errors in JavaScript: trying to call something that is not a function. In JavaScript and in JavaScripty, calling a non-function raises a run-time error. Such a run-time error is known as a dynamic type error. Languages are called *dynamically typed* when they allow all syntactically valid programs to run and check for type errors during execution.

We define a Scala exception

```
case class DynamicTypeError(e: Expr) extends Exception {
  override def toString = s"TypeError: in expression $e"
}
```

```
defined class DynamicTypeError
```

to signal this case. In other words, when your interpreter discovers a dynamic type error, it should throw this exception using the following Scala code:

```
throw DynamicTypeError(e)
```

The argument should be the input expression `e` to `eval` where the type error was detected. That is, the expression where there is no possible rule to continue. For example, in the case of calling a non-function, the type error should be reported on the `Call` node and not any sub-expression.

Exercise 20.8 (10 points)**.** Add support for checking for all dynamic type errors. You should have no possibility for a `MatchError` or a `NotImplementedError`. Start with what you have from Exercise 20.7.

Tests

20.4.5 Dynamic Scoping

Exercise 20.9 (5 points)**.** Below is a cell that runs the AST from the test case you wrote in Exercise 20.5 with your interpreter implementation.

Does it evaluate to what your excepted? The evaluation output above should be different from what it would evaluate to with a JavaScript interpreter, such as Deno. Ensure that the results are different and write below what each interpreter evaluates to.

Edit this cell:

???

It seems like we implemented dynamic scoping instead of static scoping. Explain the failed test case and how your interpreter behaves differently compared to a JavaScript interpreter. Furthermore, think about why this is the case and explain in 1-2 sentences *why* your interpreter behaves differently.

Edit this cell:

???

20.4.6 Closures

In order to fix our dynamic scoping issue, we will implement explicit closures. That is, when functions are evaluated, they will use the value environment in which they were defined.

Here are the updates to our abstract syntax.

$$\begin{array}{rrcl} \text{expressions} & e & ::= & \texttt{(}x\texttt{)} \Rightarrow e_1 \mid e_1\texttt{(}e_2\texttt{)} \\ \text{values} & v & ::= & \texttt{(}x\texttt{)} \Rightarrow e_1[E] \\ \text{variables} & x & & \end{array}$$

Notice that now, closures are values (and functions are not), while functions are still expressions.

We also add the EvalFun rule to our operational semantics, and edit the EvalCall rule, seen below.

$$\boxed{E \vdash e \Downarrow v} \qquad \text{EvalFun}\ \frac{}{E \vdash \texttt{(}x\texttt{)} \Rightarrow e \Downarrow \texttt{(}x\texttt{)} \Rightarrow e[E]}$$

$$\text{EvalCall}\ \frac{E \vdash e_1 \Downarrow \texttt{(}x\texttt{)} \Rightarrow e'[E'] \qquad E \vdash e_2 \Downarrow v_2 \qquad E'[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1\texttt{(}e_2\texttt{)} \Downarrow v'}$$

In order to implement this, we will add `Closure` to our `Expr` type, and edit other helper functions as needed.

```scala
case class Closure(fun: Fun, env: Env) extends Expr
def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) | Closure(_, _) => true
  case _ => false
}
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

```
defined class Closure
defined function isValue
defined function extend
```

Exercise 20.10 (10 points)**.** With the above, implement a new version of `eval` that uses closures to enforce static scoping. Begin with what you have from Exercise 20.8.

Tests

This code tests your new implementation against the dynamic scoping test case you wrote:

If you're implementation is correct, it should evaluate to what a JavaScript interpreter evaluates it to.

20.5 Implementing Recursive Functions (Accelerated)

The remaining exercises are for those who want to go deeper and take an "accelerated" version of this course.

We begin by extending our abstract syntax to allow for recursive functions. To call a function within itself, we permit functions to have a variable identifier to refer to itself. If the identifier is present, then it can be used for recursion.

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & x^{?}(y) \Rightarrow e_1 \mid e_1(e_2) \\
\text{optional variables} & x^{?} & ::= & x \mid \varepsilon \\
\text{variables} & x & &
\end{array}
$$

20.5.1 Defining Inference Rules

Exercise 20.11 (10 points)**.** To allow for recursion, at a function call, we must bind the function identifier to the function value when evaluating the function body. Give a inference rule called EvalCallRec that describes this semantics.

Edit this cell:

???

20.5.2 Writing a Test Case

Exercise 20.12 (1 point)**.** Write a function `sumOneToN` that computes the sum from `1` to `n` using the fragment of JavaScripty in this assignment.

Edit this cell:

In order to allow for recursive, we must edit our `Fun` constructor to accept an optional variable. (We must also re-run the other constructor and helper functions that rely on `Fun`.)

Exercise 20.13 (4 points)**.** Now that we have an abstract syntax tree node to write recursive functions, create an `Expr` that is a recursive function which computes the sum from `1` to a parameter `n`. This will be used in a test case for an updated version of `eval`. Write out the AST that your `sumOneToN` function will be parsed to.

Edit this cell:

Exercise 20.14 (10 points)**.** Rewrite your `eval` function to handle recursive functions.

This cell tests your implementation against your test case `sumOneToN`.

Tests

21 Evaluation Order

In defining a big-step operational semantics (Section 18.1), we have carefully specified several aspects of how the expression e_1 + e_2 should be evaluated. In essence, it says that it adds two numbers 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 JavaScripty expression:

```
(console.log(1), 1) + (console.log(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 console a representation of that value. If the left operand of `+` is evaluated first before the right operand, then the above expression 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 early on in discussing functional versus imperative computation (Section 3.1), 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. So far in JavaScripty, our only side-effecting expression is `console.log(` e_1 `)`. If we remove the `console.log`s from the above expression, then the evaluation order cannot be observed.

21.1 A Small-Step Operational Semantics

The big-step operational semantics (Section 18.1) 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 previous chapters, we will not extend this judgment form with value environments for free variables. 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:

$$\texttt{const one = 1; one + one} \quad\longrightarrow\quad \texttt{1 + 1}$$

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 substitution with big-step as in Section 19.4 or environments with small-step). Explicit environments just get a bit more unwieldy here.

21.2 One Type of Values

Let us consider an object language with just numbers n, a unary arithmetic operator `-`, and a binary arithmetic operator `+`:

$$\begin{array}{rrcl}
\text{expressions} & e & ::= & v \mid \mathit{uop}\, e_1 \mid e_1\ \mathit{bop}\, e_2 \\
\text{values} & v & ::= & n \\
\text{unary operators} & \mathit{uop} & ::= & \texttt{-} \\
\text{binary operators} & \mathit{bop} & ::= & \texttt{+} \\
\text{numbers} & n & &
\end{array}$$

```
trait Expr // e
trait Uop  // uop
trait Bop  // bop
```

```scala
case class Unary(uop: Uop, e1: Expr) extends Expr            // e ::= uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr // e ::= e1 bop e2

case class N(n: Double) extends Expr // e ::= n
case object Neg extends Uop          // uop ::= -
case object Plus extends Bop         // bop ::= +

def isValue(e: Expr): Boolean = e match {
  case N(_) => true
  case _ => false
}

val e_oneplustwoplusthreeplusfour = Binary(Plus, Binary(Plus, N(1), N(2)), Binary(Plus, N(3)
```

```
defined trait Expr
defined trait Uop
defined trait Bop
defined class Unary
defined class Binary
defined class N
defined object Neg
defined object Plus
defined function isValue
e_oneplustwoplusthreeplusfour: Binary = Binary(
  bop = Plus,
  e1 = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0)),
  e2 = Binary(bop = Plus, e1 = N(n = 3.0), e2 = N(n = 4.0))
)
```

Do Something

First, we need to describe what action does an operation perform. For example, we want to say that the `-` operator negates a number and the `+` operator adds two numbers, respectively. We say this with the following two rules:

$$\frac{n' = -n_1}{\texttt{-}\, n_1 \longrightarrow n'}\ \textsc{DoNeg} \qquad \frac{n' = n_1 + n_2}{n_1 \,\texttt{+}\, n_2 \longrightarrow n'}\ \textsc{DoPlus}$$

These rules say that the expression $\texttt{-}\, n_1$ and the expression $n_1 \,\texttt{+}\, n_2$ reduces in one step to an integer value n' that are the negation of a number n_1 and the addition of the numbers n_1

and n_2, respectively. We use the meta-variables n_1, n_2, and n' to express constraints that particular positions in the expressions numeric values. Note that the in the conclusions are the syntactic operators `-` and `+`, while the $-$ and $+$ in the premises express mathematical negation of a number and addition of two numbers, respectively. As we discussed previously, 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).

Note that the following way of writing DoNeg and DoPlus say the same thing:

$$\textsc{DoNeg}\ \frac{}{\texttt{-}\,n_1 \longrightarrow -n_1} \qquad \textsc{DoPlus}\ \frac{}{n_1 \,\texttt{+}\, n_2 \longrightarrow n_1 + n_2}$$

Observe the distinction between the syntactic operators `-` and `+` and the mathematical operations $-$ and $+$.

Search for Something to Do

Second, we need to describe how we find the next operation to perform. These rules will capture issues like evaluation order described informally above.

For $\texttt{-}\,e_1$, the rule SearchNeg says, "If it is possible to take a step from e_1 to e_1', then $\texttt{-}\,e_1$ steps to $\texttt{-}\,e_1'$." That is, the next expression to reduce is somewhere in the sub-expression e_1.

$$\textsc{SearchNeg}\ \frac{e_1 \longrightarrow e_1'}{\texttt{-}\,e_1 \longrightarrow \texttt{-}\,e_1'}$$

For $e_1 \,\texttt{+}\, e_2$, there are multiple ways we can define the next possible step. We could take a step in on the left (i.e., e_1) or on the right (i.e., e_2). We could reduce e_1 to a value before continuing on to e_2 (i.e., called left-to-right) or vice versa (i.e., called right-to-left), or we can allow e_1 and e_2 to reduce concurrently.

To specify that $e_1 \,\texttt{+}\, e_2$ should be evaluated left-to-right, we use the following two rules:

$$\textsc{SearchPlus1}\ \frac{e_1 \longrightarrow e_1'}{e_1 \,\texttt{+}\, e_2 \longrightarrow e_1' \,\texttt{+}\, e_2} \qquad \textsc{SearchPlus2}\ \frac{e_2 \longrightarrow e_2'}{n_1 \,\texttt{+}\, e_2 \longrightarrow n_1 \,\texttt{+}\, e_2'}$$

The SearchPlus1 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 rule is similar except that it applies only if the left expression is a value (i.e., $n_1 + e_2$). 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 SearchPlus1 could possibly apply, and (2) if e_1 is a number value, then only SearchPlus2 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 sub-expression that is the next operation to perform is called the *redex* (for *reducible expression*).

Observe that there is no rule for a number literal n. It is already a value, so there is no reduction step.

Implementation

We translate these rules directly to an implementation:

```
def step(e: Expr): Expr = {
  require(!isValue(e))
  e match {
    // DoNeg
    case Unary(Neg, N(n1)) => N(-n1)
    // SearchNeg
    case Unary(Neg, e1) => Unary(Neg, step(e1))
    // DoPlus
    case Binary(Plus, N(n1), N(n2)) => N(n1 + n2)
    // SearchPlus2
    case Binary(Plus, N(n1), e2) => Binary(Plus, N(n1), step(e2))
    // SearchPlus1
    case Binary(Plus, e1, e2) => Binary(Plus, step(e1), e2)
  }
}
```

```
defined function step
```

Observe that we do have to pay attention to the specificity of the pattern match and order the `// SearchPlus2` case before the `// SearchPlus1` case. And if there is no matching case, then we would get a `MatchError` exception.

A Test Case and a Step Judgment

We consider a test case to show evaluation order:

```
e_oneplustwoplusthreeplusfour
val e_step_oneplustwoplusthreeplus = step(e_oneplustwoplusthreeplusfour)
```

```
res2_0: Binary = Binary(
  bop = Plus,
  e1 = Binary(bop = Plus, e1 = N(n = 1.0), e2 = N(n = 2.0)),
  e2 = Binary(bop = Plus, e1 = N(n = 3.0), e2 = N(n = 4.0))
)
e_step_oneplustwoplusthreeplus: Expr = Binary(
  bop = Plus,
  e1 = N(n = 3.0),
  e2 = Binary(bop = Plus, e1 = N(n = 3.0), e2 = N(n = 4.0))
)
```

Calling `step(e_oneplustwoplusthreeplusfour)` corresponds to a witness of the judgment

$$(1 + 2) + (3 + 4) \longrightarrow 3 + (3 + 4)$$

Observe that we reduce the left side of the top-level `+`.

Let us instrument `step` to show a derivation:

```
def step(e: Expr): Expr = {
  require(!isValue(e))
  val e_ = e match {
     // DoNeg
    case Unary(Neg, N(n1)) => {
      println("------------------------------------------------ DoNeg")
      N(-n1)
    }
    // SearchNeg
    case Unary(Neg, e1) => {
      println("------------------------------------------------ SearchNeg")
      Unary(Neg, step(e1))
    }
    // DoPlus
    case Binary(Plus, N(n1), N(n2)) => {
```

```scala
      println("------------------------------------------------ DoPlus")
      N(n1 + n2)
    }
    // SearchPlus2
    case Binary(Plus, N(n1), e2) => {
      val e2_ = step(e2)
      println("------------------------------------------------ SearchPlus2")
      Binary(Plus, N(n1), e2_)
    }
    // SearchPlus1
    case Binary(Plus, e1, e2) => {
      val e1_ = step(e1)
      println("------------------------------------------------ SearchPlus1")
      Binary(Plus, e1_, e2)
    }
  }
  println(s"$e ---> $e_")
  e_
}

step(e_oneplustwoplusthreeplusfour)
```

```
------------------------------------------------ DoPlus
Binary(Plus,N(1.0),N(2.0)) ---> N(3.0)
------------------------------------------------ SearchPlus1
Binary(Plus,Binary(Plus,N(1.0),N(2.0)),Binary(Plus,N(3.0),N(4.0))) ---> Binary(Plus,N(3.0),B

defined function step
res3_1: Expr = Binary(
  bop = Plus,
  e1 = N(n = 3.0),
  e2 = Binary(bop = Plus, e1 = N(n = 3.0), e2 = N(n = 4.0))
)
```

Observe that the derivation is simply to find which sub-expression to apply a single Do rule.

Meta-Theory

Considering these 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:

Proposition 21.1 (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 consider such proofs in detail(cf., **?@sec-induction-on-derivations**).

21.3 Dynamic Typing

Let us add boolean values to our JavaScripty variant:

$$\begin{array}{rrcl} \text{values} & v & ::= & b \\ \text{booleans} & b & & \end{array}$$

```
case class B(b: Boolean) extends Expr // e ::= b

def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) => true
  case _ => false
}

val e_true = B(true)
val e_trueplustwo = Binary(Plus, e_true, N(2))
```

```
defined class B
defined function isValue
e_true: B = B(b = true)
e_trueplustwo: Binary = Binary(bop = Plus, e1 = B(b = true), e2 = N(n = 2.0))
```

Since we only add boolean literals that are values, there are no additional rules we need for $e \longrightarrow e'$.

```
def step(e: Expr): Expr = {
  require(!isValue(e), s"$e should not be a value")
  e match {
    // DoPlus
    case Binary(Plus, N(n1), N(n2)) => N(n1 + n2)
```

```
    // SearchPlus2
    case Binary(Plus, N(n1), e2) => Binary(Plus, N(n1), step(e2))
    // SearchPlus1
    case Binary(Plus, e1, e2) => Binary(Plus, step(e1), e2)
  }
}
```

```
defined function step
```

As expected with the expression **true** + 2, we run into undefined behavior with these rules, which manifests haphazardly in our implementation as failing the `require(!isValue(e))`:

```
e_trueplustwo
val v_trueplustwo = step(e_trueplustwo)
```

Our implicit intent is that a type error is where we are *stuck*—that is, there is no next step and the expression e is not a value. But what if we are *stuck* because we have a bug in our rules? We want to explicit about when there is a dynamic type error (i.e., there is a bug in the object program that the programmer gave us) versus a bug in our semantic rules. Previously, we did so informally in implementation with `throw DynamicTypeError(e)` (cf. Section 18.3), but that relies on the exception semantics of the Scala meta-language that was not explicitly described in our semantics specification. We wish to also define precisely the sub-expression `e` that is to blame.

Let us introduce a step-result type

$$\text{step-results} \quad r \quad ::= \quad \mathsf{typeerror}\, e \mid e'$$

to make explicit that `step` can return a type-error result. Specifically, a step-result is either a $\mathsf{typeerror}\, e$ with the expression e to blame or a one-step reduced expression e'.

```
case class DynamicTypeError(e: Expr)          // typeerror e
type Result = Either[DynamicTypeError, Expr] // r ::= typeerror e | e
```

```
defined class DynamicTypeError
defined type Result
```

We have chosen to represent a step-result r in Scala as an `Either[DynamicTypeError, Expr]`.

We now consider the judgment form $e \longrightarrow r$ that says, "Expression e takes a step to a result r." with the intention that `step: Expr => Result` is a total function:

```scala
def step(e: Expr): Result = ???
```

```
defined function step
```

We add rules that explicitly state when we step to a typeerror:

$$\frac{}{-\, n_1 \longrightarrow -n_1}\ \textsc{DoNeg} \qquad \frac{v_1 \neq n_1}{-\, v_1 \longrightarrow \mathsf{typeerror}(-\, v_1)}\ \textsc{TypeErrorNeg} \qquad \frac{e_1 \longrightarrow e_1'}{-\, e_1 \longrightarrow -\, e_1'}\ \textsc{SearchNeg}$$

$$\frac{}{n_1 + n_2 \longrightarrow n_1 + n_2}\ \textsc{DoPlus} \qquad \frac{v_2 \neq n_2}{n_1 + v_2 \longrightarrow \mathsf{typeerror}(n_1 + v_2)}\ \textsc{TypeErrorPlus2} \qquad \frac{e_2 \longrightarrow e_2'}{n_1 + e_2 \longrightarrow n_1 + e_2'}\ \textsc{SearchPlus2}$$

$$\frac{v_1 \neq n_1}{v_1 + e_2 \longrightarrow \mathsf{typeerror}(v_1 + e_2)}\ \textsc{TypeErrorPlus1} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{SearchPlus1}$$

```scala
def step(e: Expr): Either[DynamicTypeError, Expr] = {
  require(!isValue(e))
  e match {
    // DoNeg
    case Unary(Neg, N(n1)) => Right(N(-n1))
    // TypeErrorNeg
    case Unary(Neg, v1) if isValue(v1) => Left(DynamicTypeError(e))
    case Unary(Neg, e1) => step(e1) match {
      // SearchNeg
      case Right(e1) => Right(Unary(Neg, e1))
      // PropagateNeg
      case Left(error) => Left(error)
    }
    case Binary(Plus, N(n1), v2) if isValue(v2) => v2 match {
      // DoPlus
      case N(n2) => Right(N(n1 + n2))
      // TypeErrorPlus2
      case _ => Left(DynamicTypeError(e))
    }
    case Binary(Plus, v1, e2) if isValue(v1) => v1 match {
      case N(n1) => step(e2) match {
```

```
        // SearchPlus2
        case Right(e2) => Right(Binary(Plus, N(n1), e2))
        // PropagatePlus2
        case Left(error) => Left(error)
      }
      // TypeErrorPlus1
      case _ => Left(DynamicTypeError(e))
    }
    case Binary(Plus, e1, e2) => step(e1) match {
      // SearchPlus1
      case Right(e1) => Right(Binary(Plus, e1, e2))
      // PropagatePlus1
      case Left(error) => Left(error)
    }
  }
}
```

```
defined function step
```

```
e_trueplustwo
val v_trueplustwo = step(e_trueplustwo)
```

From the implementation, we discover that there are three more cases, that is,

```
case Left(error) => Left(error)
```

cases to make `step: Expr => Either[DynamicTypeError, Expr]` a total function. In those cases, we encountered a typeerror in a sub-expression from recursively calling `step`, and we simply want to propagate the `DynamicTypeError`:

$$\textsc{PropagateNeg}\ \frac{e_1 \longrightarrow \mathsf{typeerror}\, e}{-\, e_1 \longrightarrow \mathsf{typeerror}\, e} \qquad \textsc{PropagatePlus2}\ \frac{e_2 \longrightarrow \mathsf{typeerror}\, e}{n_1 + e_2 \longrightarrow \mathsf{typeerror}\, e} \qquad \textsc{PropagatePlus1}\ \frac{e_1 \longrightarrow \mathsf{typeerror}\, e}{e_1 + e_2 \longrightarrow \mathsf{typeerror}\, e}$$

Either.map

Recall that the `Either[Err, A]` type is often used like an `Option[A]` type, except that the "bad" case has some data payload (i.e., the `None` case for an `Option[A]` corresponds to the `Left(err)` case for an `Either[Err, A]`). This "propagate error" pattern is so common that the the `Either[Err, A]` type has a higher-order method `map` that takes a callback for what transformation to make with the `Right` case and otherwise propagate the `Left` case.

Thus, we can refactor the `step` implementation from above as follows:

```
def step(e: Expr): Either[DynamicTypeError, Expr] = {
  require(!isValue(e))
  e match {
    // DoNeg
    case Unary(Neg, N(n1)) => Right(N(-n1))
    // TypeErrorNeg
    case Unary(Neg, v1) if isValue(v1) => Left(DynamicTypeError(e))
    // SearchNeg and PropagateNeg
    case Unary(Neg, e1) => step(e1) map { e1 => Unary(Neg, e1) }
    case Binary(Plus, N(n1), v2) if isValue(v2) => v2 match {
      // DoPlus
      case N(n2) => Right(N(n1 + n2))
      // TypeErrorPlus2
      case _ => Left(DynamicTypeError(e))
    }
    case Binary(Plus, v1, e2) if isValue(v1) => v1 match {
      // SearchPlus2 and PropagatePlus2
      case N(n1) => step(e2) map { e2 => Binary(Plus, N(n1), e2)  }
      // TypeErrorPlus1
      case _ => Left(DynamicTypeError(e))
    }
    // SearchPlus1 and PropagatePlus1
    case Binary(Plus, e1, e2) => step(e1) map { e1 => Binary(Plus, e1, e2) }
  }
}
```

```
defined function step
```

Note that the `map` method is common for both `Either[Err, A]` and `Option[A]`:

```
None    map { (i: Int) => i + 1 }
Some(3) map { (i: Int) => i + 1 }
```

```
res12_0: Option[Int] = None
res12_1: Option[Int] = Some(value = 4)
```

21.4 Generic Evaluation Order

Let us add the boolean expressions for conditionals, `!`, `&&`, `||`, and `===`:

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & e_1 \mathbin{\texttt{?}} e_2 \mathbin{\texttt{:}} e_3 \\
\text{unary operators} & \mathit{uop} & ::= & \texttt{!} \\
\text{binary operators} & \mathit{bop} & ::= & \texttt{\&\&} \mid \texttt{||} \mid \texttt{===}
\end{array}
$$

```
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr // e ::= e1 ? e2 : e3

case object Not extends Uop // uop ::= !
case object And extends Bop // bop ::= &&
case object Or extends Bop  // bop ::= ||

case object Eq extends Bop  // bop ::= ===
```

```
defined class If
defined object Not
defined object And
defined object Or
defined object Eq
```

Our first observation is that evaluation order is a concern regardless of the type of operations. Suppose we define that all binary operators are evaluated left-to-right, then we can replace the rules SearchNeg, SearchPlus1, and SearchPlus2 with these more generic versions:

$$
\textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{\mathit{uop}\, e_1 \longrightarrow \mathit{uop}\, e_1'}
\qquad
\textsc{SearchBinary1}\ \frac{e_1 \longrightarrow e_1'}{e_1\ \mathit{bop}\, e_2 \longrightarrow e_1'\ \mathit{bop}\, e_2}
\qquad
\textsc{SearchBinary2}\ \frac{e_2 \longrightarrow e_2'}{v_1\ \mathit{bop}\, e_2 \longrightarrow v_1\ \mathit{bop}\, e_2'}
$$

The same makes sense for the rules that propagate type error, replacing PropagateNeg, PropagatePlus1, and PropagatePlus2 with the following:

$$
\textsc{PropagateUnary}\ \frac{e_1 \longrightarrow \mathsf{typeerror}\, e}{\mathit{uop}\, e_1 \longrightarrow \mathsf{typeerror}\, e}
\qquad
\textsc{PropagateBinary1}\ \frac{e_1 \longrightarrow \mathsf{typeerror}\, e}{e_1\ \mathit{bop}\, e_2 \longrightarrow \mathsf{typeerror}\, e}
\qquad
\textsc{PropagateBinary2}\ \frac{e_2 \longrightarrow \mathsf{typeerror}\, e}{v_1\ \mathit{bop}\, e_2 \longrightarrow \mathsf{typeerror}\, e}
$$

21.5 Non-Determinism

Consider the relationship between the SearchBinary1 and SearchBinary2 rules that define generically for all binary operators *bop* a left-to-right evaluation order and dynamic typing.

$$\frac{v_1 \neq n_1}{v_1 \texttt{ + } e_2 \longrightarrow \textsf{typeerror}(v_1 \texttt{ + } e_2)}\ \textsc{TypeErrorPlus1} \qquad \frac{e_2 \longrightarrow e_2'}{n_1 \texttt{ + } e_2 \longrightarrow n_1 \texttt{ + } e_2'}\ \textsc{SearchPlus2}$$

In the rules above, the TypeErrorPlus1 rule and the SearchPlus2 are disjoint (i.e., one rule applies to determine the next step) for an expression of the form $v_1 \texttt{ + } e_2$. However, if we replace SearchPlus2 with SearchBinary2, that is no longer the case. That is, our step relation $e \longrightarrow r$ is no longer deterministic, or we say that it has *non-determinism*.

What this means from an implementation standpoint is that while the semantics specification with SearchPlus2 states that an implementation must step to a typeerror using TypeErrorPlus1 before taking a step e_2. The pair of rules TypeErrorPlus1 and SearchBinary2 are not disjoint, so an implementation may choose to take some steps in e_2 using SearchBinary2 before stepping to a typeerror using TypeErrorPlus1. The implementation that steps to a typeerror as soon as possible is still valid but other implementations are now permitted.

21.6 Short-Circuiting Evaluation

Let us define the semantics of the core boolean expressions as in JavaScript. The DoNot rule converts value v_1 into a boolean and returns its negation:

$$\frac{v_1 \rightsquigarrow b_1}{\texttt{!}\, v_1 \longrightarrow \neg b_1}\ \textsc{DoNot}$$

What we have seen already is that the `&&` and `||` operators do not behave like mathematical logic operators $\wedge$ and $\vee$ (whereas `!` does behave like $\neg$ after coercion):

$$\frac{v_1 \rightsquigarrow \mathbf{true}}{v_1 \texttt{ \&\& } e_2 \longrightarrow e_2}\ \textsc{DoAndTrue} \qquad \frac{v_1 \rightsquigarrow \mathbf{false}}{v_1 \texttt{ \&\& } e_2 \longrightarrow v_1}\ \textsc{DoAndFalse}$$

$$\frac{\textsc{DoOrTrue} \quad v_1 \rightsquigarrow \mathbf{true}}{v_1 \;\texttt{||}\; e_2 \longrightarrow v_1} \qquad \frac{\textsc{DoOrFalse} \quad v_1 \rightsquigarrow \mathbf{false}}{v_1 \;\texttt{||}\; e_2 \longrightarrow e_2}$$

For the `&&` and `||` operators, given that binary operators evaluate left-to-right, once we have evaluated e_1 to a value, we do not necessarily need to evaluate e_2. That is, we may *short-circuit* evaluating e_2. We say that a *short-circuiting evaluation* of an expression is one where a value is produced before evaluating all sub-expressions to values. We say that the expressions $e_1 \;\texttt{\&\&}\; e_2$ and $e_1 \;\texttt{||}\; e_2$ may short-circuit. In particular, the DoAndFalse rule says that $v_1 \;\texttt{\&\&}\; e_2$ where v_1 converts to **false** evaluates to v_1 without ever evaluating e_2. The analgous rule for `||` is DoOrTrue.

For $e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3$, 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:

$$\frac{\textsc{DoIfTrue} \quad v_1 \rightsquigarrow \mathbf{true}}{v_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_2} \qquad \frac{\textsc{DoIfFalse} \quad v_1 \rightsquigarrow \mathbf{false}}{v_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_3}$$

Observe how similar DoAndTrue and DoOrFalse are to DoIfTrue and DoIfFalse, respectively. We see that `&&` and `||` operators have quite a bit similarity to control-flow operators like $e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3$ and significant differences compared to the mathematical logic operators $\wedge$ and $\vee$, respectively.

Many programming languages implement short-circuiting boolean operators `&&` and `||`.

Searching for a redex in `&&` and `||` are covered by SearchBinary1 and SearchBinary2. For $e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3$, we need a SearchIf rule to reduce the guard expression e_1 to a value:

$$\frac{\textsc{SearchIf} \quad e_1 \longrightarrow e_1'}{e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_1' \;\texttt{?}\; e_2 \;\texttt{:}\; e_3}$$

and propagating $\mathsf{typeerror}$ as needed:

$$\frac{\textsc{PropagateIf} \quad e_1 \longrightarrow \mathsf{typeerror}\, e}{e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow \mathsf{typeerror}\, e}$$

21.7 Polymorphism

The `===` operator does not perform coercions but is still *polymorphic*, that is, it applies regardless of the type of value of its arguments:

$$\frac{\textsc{DoEquality}}{v_1 \;\texttt{===}\; v_2 \longrightarrow v_1 = v_2}$$

We observe that $v_1 = v_2$ is **false** if the types of v_1 and v_2 do not match. In JavaScript, this is called strict equality, as it has another operator `==` that performs coercions before comparing for equality called loose equality.

21.8 Recursion

Let us consider our object language JavaScripty with variables, binding, and optionally-named functions:

$$\begin{array}{rrcl}
\text{values} & v & ::= & x \mid x^{?}\texttt{(}y\texttt{)} \Rightarrow e_1 \\
\text{expressions} & e & ::= & \textbf{const}\; x \;\texttt{=}\; e_1\texttt{;}\; e_2 \mid e_1\texttt{(}e_2\texttt{)} \\
\text{optional variables} & x^{?} & ::= & x \mid \varepsilon \\
\text{variables} & x & &
\end{array}$$

To call a function within itself, we permit functions to have a variable identifier to refer to itself. If the identifier is present, then it can be used for recursion.

```scala
case class Var(x: String) extends Expr                                   // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr         // e ::= const x = e1
case class Fun(xopt: Option[String], y: String, e1: Expr) extends Expr // e ::= x?(y) => e1
case class Call(e1: Expr, e2: Expr) extends Expr                         // e ::= e1(e2)

def isValue(e: Expr): Boolean = e match {
  case N(_) | B(_) | Fun(_, _, _) => true
  case _ => false
}
```

```
defined class Var
defined class ConstDecl
defined class Fun
defined class Call
defined function isValue
```

For exmaple, here is an abstract syntax tree for a recursive function `silly`:

```
val e_sillyRecFun = Fun(Some("silly"), "i",
  If(Binary(Eq, Var("i"), N(0)),
     Var("j"),
     Binary(Plus,
            Var("j"),
            Call(Var("silly"), Binary(Plus, Var("i"), Unary(Neg, N(1)))))))
```

```
e_sillyRecFun: Fun = Fun(
  xopt = Some(value = "silly"),
  y = "i",
  e1 = If(
    e1 = Binary(bop = Eq, e1 = Var(x = "i"), e2 = N(n = 0.0)),
    e2 = Var(x = "j"),
    e3 = Binary(
      bop = Plus,
      e1 = Var(x = "j"),
      e2 = Call(
        e1 = Var(x = "silly"),
        e2 = Binary(
          bop = Plus,
          e1 = Var(x = "i"),
          e2 = Unary(uop = Neg, e1 = N(n = 1.0))
        )
      )
    )
  )
)
```

corresponding to the concrete syntax:

```
function silly(i) { return i === 0 ? j : j + silly(i + -1); }
```

Recall accidental "dynamic scoping" (cf. Chapter 19). We lazily wait until seeing a variable use to determine the variable binding site. The "bug" comes from using the "wrong" environment. A possible fix is to save the "right" environment with the function value—what's known as closure.

A brute force alternative is to always work with *closed* expressions (i.e., expressions that have no *free variable uses*). As soon as we know the variable binding, we eagerly "get rid" of variable uses with substitution. We see that the definition of the set of free variables uses of an expression defines the binding site and scope of a variable:

```scala
def freeVars(e: Expr): Set[String] = e match {
  case N(_) | B(_) => Set.empty
  case Unary(_, e1) => freeVars(e1)
  case Binary(_, e1, e2) => freeVars(e1) union freeVars(e2)
  case If(e1, e2, e3) => freeVars(e1) union freeVars(e2) union freeVars(e3)
  case Var(x) => Set(x)
  case ConstDecl(x, e1, e2) => freeVars(e1) union (freeVars(e2) - x)
  case Fun(xopt, y, e1) => freeVars(e1) -- xopt - y
  case Call(e1, e2) => freeVars(e1) union freeVars(e2)
}

def closed(e: Expr): Boolean = freeVars(e).isEmpty

freeVars(e_sillyRecFun)
closed(e_sillyRecFun)
```

```
defined function freeVars
defined function closed
res16_2: Set[String] = Set("j")
res16_3: Boolean = false
```

Thus, the `step` function expects input expressions that are closed, non-value expressions:

```scala
def step(e: Expr): Either[DynamicTypeError, Expr] = {
  require(closed(e), s"$e should be closed")
  require(!isValue(e), s"$e should not be a value")
  ???
}

step(e_sillyRecFun)
```

Correspondingly, there is no DoVar rule for the judgment form $e \longrightarrow r$ because e must be closed (cf. there is no EvalVar rule in Section 19.4).

The DoConstDecl rule for the variable binding expression **const** x = e_1 ; e_2 eagerly applies substitution to eliminate free-variable uses:

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

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

We then need a SearchConstDecl rule to step e_1 to a value in a $\textbf{const}\ x = e_1;\ e_2$ expression:

$$\textsc{SearchConstDecl}\ \frac{e_1 \longrightarrow e_1'}{\textbf{const}\ x = e_1;\ e_2 \longrightarrow \textbf{const}\ x = e_1';\ e_2} \qquad \textsc{PropagateConstDecl}\ \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{\textbf{const}\ x = e_1;\ e_2 \longrightarrow \mathsf{typeerror}\ e}$$

We have two cases for reducing a function call $e_1(e_2)$, depending on whether the function is named or not:

$$\textsc{DoCall}\ \frac{}{((x) \Rightarrow e_1)(v_2) \longrightarrow [v_2/x]e_1} \qquad \textsc{DoCallRec}\ \frac{v_1 = (x_1(x_2) \Rightarrow e_1)}{v_1(v_2) \longrightarrow [v_1/x_1][v_2/x_2]e_1}$$

If it is unnamed, the DoCall rule applies binding the actual argument v_2 to the formal parameter x by substituting v_2 for free variable uses x in e_1. If it named, then are two formal parameters x_1 and x_2 where x_1 is bound to the function value itself v_1 and x_2 is bound to the actual argument v_2. In this case, the DoCallRec rule applies by stepping to $[v_1/x_1][v_2/x_2]e_1$. The DoCallRec shows the essence of recursion—a self-reference to the function value itself!

If v_1 in the function call expression $v_1(e_2)$ is not a function value, then we have dynamic type error:

$$\textsc{TypeErrorCall}\ \frac{v_1 \neq x^?(y) \Rightarrow e_1}{v_1(e_2) \longrightarrow \mathsf{typeerror}(v_1(e_2))}$$

Observe that we step to a typeerror only when $v_1(v_2)$. That is, we follow JavaScript here in delaying the check for a dynamic type error until both the function position and the argument position are values.

We define evaluating function call $e_1(e_2)$ as left-to-right and continuing even if e_1 reduces to a non-function value:

$$\textsc{SearchCall1}\ \frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)} \qquad \textsc{SearchCall2}\ \frac{e_2 \longrightarrow e_2'}{v_1(e_2) \longrightarrow v_1(e_2')}$$

$$\frac{e_1 \longrightarrow \textsf{typeerror}\, e}{e_1\,(e_2) \longrightarrow \textsf{typeerror}\, e}\;\textsc{PropagateCall1} \qquad \frac{e_2 \longrightarrow \textsf{typeerror}\, e}{v_1\,(e_2) \longrightarrow \textsf{typeerror}\, e}\;\textsc{PropagateCall2}$$

21.9 Substitution

The term *capture-avoiding subsitution* means that for $[e_1/x]e_2$, we get the expression that is like e_2, but we have replaced all instances of variable x with e_1 while carefully respecting static scoping (cf., Chapter 14). There are two thorny issues that arise.

Shadowing The substitution

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

should yield (`const a = 1; a + b`). That is, only *free* instances of x in e_2 should be replaced.

Free Variable Capture The substitution

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

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

$$(\texttt{const a = 1; a + (a + 2)})$$

because the free variable `a` in e_1 gets "captured" by the **const** binding of `a`.

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

```
const c = 1; c + b
```

in which case simple 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:

$$(\texttt{const a = 1; a}) \quad \equiv_\alpha \quad (\texttt{const b = 1; b})$$

For historical reasons, this equivalence is known α-equivalence, and the process of renaming bound variables is called α-renaming (cf. Section 14.7).

In rules DoConstDecl, DoCall, and DoCallRec, our situation is more restricted than the general case discussed above. In particular, the substitution is of the form $[v/x]e$ where the replacement for v has to be a value with no free variables, so only the shadowing issue arises.

$$
\begin{array}{rcll}
[e'/x]y & \stackrel{\text{def}}{=} & e' & \text{if } x = y \\
[e'/x]y & \stackrel{\text{def}}{=} & y & \text{if } x \neq y \\
[e'/x](\textbf{const}\ y\ \texttt{=}\ e_1\texttt{;}\ e_2) & \stackrel{\text{def}}{=} & \textbf{const}\ y\ \texttt{=}\ ([e'/x]e_1)\texttt{;}\ e_2 & \text{if } x = y \\
[e'/x](\textbf{const}\ y\ \texttt{=}\ e_1\texttt{;}\ e_2) & \stackrel{\text{def}}{=} & \textbf{const}\ y\ \texttt{=}\ ([e'/x]e_1)\texttt{;}\ ([e'/x]e_2) & \text{if } x \neq y \\
[e'/x](\texttt{(}y\texttt{)}\ \texttt{=>}\ e_1) & \stackrel{\text{def}}{=} & \texttt{(}y\texttt{)}\ \texttt{=>}\ e_1 & \text{if } x = y \\
[e'/x](\texttt{(}y\texttt{)}\ \texttt{=>}\ e_1) & \stackrel{\text{def}}{=} & \texttt{(}y\texttt{)}\ \texttt{=>}\ ([e'/x]e_1) & \text{if } x \neq y \\
[e'/x](y_1\texttt{(}y_2\texttt{)}\ \texttt{=>}\ e_1) & \stackrel{\text{def}}{=} & y_1\texttt{(}y_2\texttt{)}\ \texttt{=>}\ e_1 & \text{if } x = y_1 \text{ or } x = y_2 \\
[e'/x](y_1\texttt{(}y_2\texttt{)}\ \texttt{=>}\ e_1) & \stackrel{\text{def}}{=} & y_1\texttt{(}y_2\texttt{)}\ \texttt{=>}\ ([e'/x]e_1) & \text{if } x \neq y_1 \text{ and } x \neq y_2 \\
[e'/x]n & \stackrel{\text{def}}{=} & n & \\
[e'/x]b & \stackrel{\text{def}}{=} & b & \\
[e'/x](uop\, e_1) & \stackrel{\text{def}}{=} & uop([e'/x]e_1) & \\
[e'/x](e_1\ bop\ e_2) & \stackrel{\text{def}}{=} & ([e'/x]e_1)\ bop\ ([e'/x]e_2) & \\
[e'/x](e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3) & \stackrel{\text{def}}{=} & [e'/x]e_1\ \texttt{?}\ [e'/x]e_2\ \texttt{:}\ [e'/x]e_3 & \\
[e'/x](e_1\texttt{(}e_2\texttt{)}) & \stackrel{\text{def}}{=} & ([e'/x]e_1)\texttt{(}[e'/x]e_2\texttt{)} &
\end{array}
$$

Figure 21.1: Defining substitution assuming e and e' use disjoint sets of bound variables.

In Figure 21.1, we define substitution $[e'/x]e$ 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 in e appropriately as described above. Or if we require that e' has to be a closed expression, then this pre-condition is trivally satisfied. The most interesting cases are for variable uses and 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 **const**-binding **const** y `=` e_1`;` e_2, we recall that the scope of x_1 is e_2, so we substitute in e_2 depending on whether or not $x = y$. Observe that the same reasoning applies to function literals $x^?$`(`y`)` `=>` e_1. The remaining expression forms simply "pass through" the substitution.

```scala
def substitute(with_e: Expr, x: String, in_e: Expr) = {
  require((freeVars(with_e) intersect freeVars(in_e)).isEmpty)
  def subst(in_e: Expr): Expr = in_e match {
    case Var(y) => if (x == y) with_e else in_e
    case ConstDecl(y, e1, e2) =>
      if (x == y) ConstDecl(y, subst(e1), e2) else ConstDecl(y, subst(e1), subst(e2))
    case Fun(yopt, y, e1) =>
      if (Some(x) == yopt || x == y) in_e else Fun(yopt, y, subst(e1))
    case N(_) | B(_) => in_e
```

```
    case Unary(uop, e1) => Unary(uop, subst(e1))
    case Binary(bop, e1, e2) => Binary(bop, subst(e1), subst(e2))
    case If(e1, e2, e3) => If(subst(e1), subst(e2), subst(e3))
    case Call(e1, e2) => Call(subst(e1), subst(e2))
  }
  subst(in_e)
}
```

```
defined function substitute
```

```
def toBoolean(e: Expr): Boolean = {
  require(isValue(e))
  e match {
    case B(b) => b
    case N(n) if (n compare 0.0) == 0 || (n compare -0.0) == 0 || n.isNaN => false
    case _ => true
  }
}

def step(e: Expr) = {
  require(closed(e), s"$e should be closed")
  def step(e: Expr): Either[DynamicTypeError, Expr] = {
    require(!isValue(e), s"$e should not be a value")
    e match {
      // DoNeg
      case Unary(Neg, N(n1)) => Right(N(-n1))
      // DoPlus
      case Binary(Plus, N(n1), N(n2)) => Right(N(n1 + n2))

      // DoNot
      case Unary(Not, v1) if isValue(v1) => Right(B(toBoolean(v1)))
      // DoAndTrue and DoAndFalse
      case Binary(And, v1, e2) if isValue(v1) => Right(if (toBoolean(v1)) e2 else v1)
      // DoOrTrue and DoOrFalse
      case Binary(Or, v1, e2) if isValue(v1) => Right(if (toBoolean(v1)) v1 else e2)
      // DoIf
      case If(v1, e2, e3) if isValue(v1) => Right(if (toBoolean(v1)) e2 else e3)
      // DoEquality
      case Binary(Eq, v1, v2) if isValue(v1) && isValue(v2) => Right(B(v1 == v2))

      // DoConstDecl
```

```scala
      case ConstDecl(x, v1, e2) if isValue(v1) => Right(substitute(v1, x, e2))
      // DoCall and DoCallRec
      case Call(v1 @ Fun(xopt, y, e1), v2) if isValue(v2) => {
        val e1_ = substitute(v2, y, e1)
        Right(xopt match {
          case None => e1_
          case Some(x) => substitute(v1, x, e1_)
        })
      }

      // SearchUnary and PropagateUnary
      case Unary(uop, e1) => step(e1) map { e1 => Unary(uop, e1) }
      // SearchBinary2 and PropagateBinary2
      case Binary(bop, v1, e2) if isValue(v1) => step(e2) map { e2 => Binary(bop, v1, e2) }
      // SearchBinary1 and PropagateBinary1
      case Binary(bop, e1, e2) => step(e1) map { e1 => Binary(bop, e1, e2) }

      // SearchIf and PropagateIf
      case If(e1, e2, e3) => step(e1) map { e1 => If(e1, e2, e3) }

      // SearchConstDecl and PropagateConstDecl
      case ConstDecl(x, e1, e2) => step(e1) map { e1 => ConstDecl(x, e1, e2) }

      // SearchCall2 and PropagateCall2
      case Call(v1, e2) if isValue(v1) => step(e2) map { e2 => Call(v1, e2) }

      // TypeErrorNeg
      case Unary(Neg, v1) if isValue(v1) => Left(DynamicTypeError(e))
      // TypeErrorPlus1
      case Binary(Plus, v1, _) if isValue(v1) => Left(DynamicTypeError(e))
      // TypeErrorPlus2
      case Binary(Plus, _, v2) if isValue(v2) => Left(DynamicTypeError(e))
      // TypeErrorCall
      case Call(v1, _) if isValue(v1) => Left(DynamicTypeError(e))

      // Anything else is an implementation bug
    }
  }
  step(e)
}

val e_closedSillyRecFun = ConstDecl("j", N(1), Call(e_sillyRecFun, N(3)))
```

```scala
val Right(e_oneStepClosedSillyRecFun) = step(e_closedSillyRecFun)
val Right(e_twoStepsClosedSillyRecFun) = step(e_oneStepClosedSillyRecFun)
```

```
defined function toBoolean
defined function step
e_closedSillyRecFun: ConstDecl = ConstDecl(
  x = "j",
  e1 = N(n = 1.0),
  e2 = Call(
    e1 = Fun(
      xopt = Some(value = "silly"),
      y = "i",
      e1 = If(
        e1 = Binary(bop = Eq, e1 = Var(x = "i"), e2 = N(n = 0.0)),
        e2 = Var(x = "j"),
        e3 = Binary(
          bop = Plus,
          e1 = Var(x = "j"),
          e2 = Call(
            e1 = Var(x = "silly"),
            e2 = Binary(
              bop = Plus,
              e1 = Var(x = "i"),
              e2 = Unary(uop = Neg, e1 = N(n = 1.0))
            )
          )
        )
      )
    ),
    e2 = N(n = 3.0)
  )
)
e_oneStepClosedSillyRecFun: Expr = Call(
  e1 = Fun(
    xopt = Some(value = "silly"),
    y = "i",
    e1 = If(
      e1 = Binary(bop = Eq, e1 = Var(x = "i"), e2 = N(n = 0.0)),
      e2 = N(n = 1.0),
      e3 = Binary(
        bop = Plus,
        e1 = N(n = 1.0),
```

```
        e2 = Call(
          e1 = Var(x = "silly"),
          e2 = Binary(
            bop = Plus,
            e1 = Var(x = "i"),
            e2 = Unary(uop = Neg, e1 = N(n = 1.0))
          )
        )
      )
    )
  ),
  e2 = N(n = 3.0)
)
e_twoStepsClosedSillyRecFun: Expr = If(
  e1 = Binary(bop = Eq, e1 = N(n = 3.0), e2 = N(n = 0.0)),
  e2 = N(n = 1.0),
  e3 = Binary(
    bop = Plus,
    e1 = N(n = 1.0),
    e2 = Call(
      e1 = Fun(
        xopt = Some(value = "silly"),
        y = "i",
        e1 = If(
          e1 = Binary(bop = Eq, e1 = Var(x = "i"), e2 = N(n = 0.0)),
          e2 = N(n = 1.0),
          e3 = Binary(
            bop = Plus,
            e1 = N(n = 1.0),
            e2 = Call(
              e1 = Var(x = "silly"),
              e2 = Binary(
                bop = Plus,
                e1 = Var(x = "i"),
                e2 = Unary(uop = Neg, e1 = N(n = 1.0))
              )
            )
          )
        )
      ),
      e2 = Binary(
        bop = Plus,
        e1 = N(n = 3.0),
```

```
        e2 = Unary(uop = Neg, e1 = N(n = 1.0))
      )
    )
  )
)
```

21.10 Multi-Step Reduction

We have now defined how to take one-step of evaluation (without typeerror), namely a judgment of the form $e \longrightarrow e'$. The multi-step reduction judgment form

$$e \longrightarrow^* e'$$

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

$$\boxed{e \longrightarrow^* e'} \qquad \frac{}{e \longrightarrow^* e}\ \textsc{MultiStepZero} \qquad \frac{e \longrightarrow e' \qquad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ \textsc{MultiStepAtLeastOne}$$

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

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:

Proposition 21.2 (Big-Step and Small-Step Equivalence). $\cdot \vdash e \Downarrow v$ *if and only if* $e \longrightarrow^* v$.

The multi-step reduction judgment form $e \longrightarrow^* e'$ enables us to state when an expression e' is reachable under some number of steps from e (i.e., $e \longrightarrow \cdots$).

At the same time, we have implicitly assumed that there is a top-level or outer loop that repeatedly applies a step until reaching a value or typeerror. Let us a define a judgment for $e \hookrightarrow r$ that says, “Evaluation e reduces to a result r that is either a value or a typeerror using some number of steps.”

$$\boxed{e \hookrightarrow r} \qquad \frac{e\ \mathsf{value}}{e \hookrightarrow e}\ \textsc{ReducesToValue} \qquad \frac{e \longrightarrow \mathsf{typeerror}\, e'}{e \hookrightarrow \mathsf{typeerror}\, e'}\ \textsc{ReducesToTypeError} \qquad \frac{e \longrightarrow e' \qquad e' \hookrightarrow r}{e \hookrightarrow r}\ \textsc{ReducesToStep}$$

And let us implement $e \hookrightarrow r$ as follows with `iterateStep`:

```
def iterateStep(e: Expr): Either[DynamicTypeError, Expr] =
  // ReducesToValue
  if (isValue(e)) Right(e)
  else step(e) match {
    // ReducesToTypeError
    case Left(error) => Left(error)
    // ReducesToStep
    case Right(e) => iterateStep(e)
  }
```

```
defined function iterateStep
```

Note again the passthrough `case Left(error) => Left(error)`. The `Either[Err, A]` type has a method `flatMap` similar to the `map` method, except it permits its callback to also "fail." We can thus refactor `iterateStep` as follows using `flatMap`:

```
def iterateStep(e: Expr): Either[DynamicTypeError, Expr] =
  // ReducesToValue
  if (isValue(e)) Right(e)
  // ReducesToTypeError and ReducesToStep
  else step(e) flatMap iterateStep
```

```
defined function iterateStep
```

That is, if either `step` or `iterateStep` "fails" by returning a `Left` value, that `Left` will be returned. Otherwise, the resulting `Expr` from a "successful" `step(e)` was passed to `iterateStep`.

Let's run an integration test for `step` from `iterateStep`:

```
iterateStep(e_closedSillyRecFun)
```

```
res22: Either[DynamicTypeError, Expr] = Right(value = N(n = 4.0))
```

One benefit of the small-step semantics is that we can easily log the intermediate steps of reduction:

```scala
def iterateStep(e: Expr) = {
  println(e)
  def loop(e: Expr): Either[DynamicTypeError, Expr] =
    // ReducesToValue
    if (isValue(e)) Right(e)
    // ReducesToTypeError and ReducesToStep
    else step(e) flatMap { e => println(s"--> $e"); loop(e) }
  loop(e)
}

iterateStep(e_closedSillyRecFun)
```

```
ConstDecl(j,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0)),Var(j),Binary(Plus,Var
--> Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0)),N(1.0),Binary(Plus,N(1.0),Call(Var(si
--> If(Binary(Eq,N(3.0),N(0.0)),N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(E
--> If(B(false),N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0))
--> Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0)),N(1.0),Binary(Plus
--> Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0)),N(1.0),Binary(Plus
--> Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0)),N(1.0),Binary(Plus
--> Binary(Plus,N(1.0),If(Binary(Eq,N(2.0),N(0.0)),N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(s
--> Binary(Plus,N(1.0),If(B(false),N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binar
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0))
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0))
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binary(Eq,Var(i),N(0.0))
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),If(Binary(Eq,N(1.0),N(0.0)),N(1.0),Binary(Plus,N(1
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),If(B(false),N(1.0),Binary(Plus,N(1.0),Call(Fun(Some
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binar
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binar
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),Call(Fun(Some(silly),i,If(Binar
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),If(Binary(Eq,N(0.0),N(0.0)),N(1
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),If(B(true),N(1.0),Binary(Plus,N
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),Binary(Plus,N(1.0),N(1.0))))
--> Binary(Plus,N(1.0),Binary(Plus,N(1.0),N(2.0)))
--> Binary(Plus,N(1.0),N(3.0))
--> N(4.0)

defined function iterateStep
res23_1: Either[DynamicTypeError, Expr] = Right(value = N(n = 4.0))
```

JavaScripty: Variables, Numbers, Booleans, Functions, and Strings

22 Lab: Small-Step Operational Semantics

Learning Goals

The primary learning goals of this assignment are to build intuition for the following:

- the distinction between a big-step and a small-step operational semantics;
- evaluation order; and
- substitution and program transformation.

Functional Programming Skills Iteration. Introduction to higher-order functions.
Programming Language Ideas Semantics: evaluation order. Small-step operational semantics. Substitution and program transformation.

Instructions

A version of project files for this lab resides in the public pppl-lab3 repository. Please follow separate instructions to get a private clone of this repository for your work.

You will be replacing `???` or `case _ => ???` in the `Lab3.scala` file with solutions to the coding exercises described below.

Your lab will not be graded if it does not compile. You may check compilation with your IDE, `sbt compile`, or with the "sbt compile" GitHub Action provided for you. Comment out any code that does not compile or causes a failing assert. Put in `???` as needed to get something that compiles without error.

You may add additional tests to the `Lab3Spec.scala` file. In the `Lab3Spec.scala`, there is empty test class `Lab3StudentSpec` that you can use to separate your tests from the given tests in the `Lab3Spec` class. You are also likely to edit `Lab3.worksheet.sc` for any scratch work. You can also use `Lab3.worksheet.js` to write and experiment in a JavaScript file that you can then parse into a JavaScripty AST (see `Lab3.worksheet.sc`).

If you like, you may use this notebook for experimentation. However, **please make sure your code is in `Lab3.scala`; code in this notebook will not graded.**

Note that there is a section with concept exercises (Section 22.6). Make sure to complete the concept exercises in that section and **turn in this file as part of your submission for the**

concept exercises. However, all code and testing exercises from other sections are submitted in `Lab3.scala` or `Lab3Spec.scala`.

Recall that you need to switch kernels between running JavaScript and Scala cells.

22.1 Small-Step Interpreter: JavaScripty Functions

We consider the same JavaScripty variant as in the previous exercise on big-step operational semantics (Section 20.1) where the interesting language feature are first-class functions:

```
trait Expr
case class Fun(xopt: Option[String], y: String, e1: Expr) extends Expr // e ::= x?(y) => e1
case class Call(e1: Expr, e2: Expr) extends Expr                        // e ::= e1(e2)
```

```
defined trait Expr
defined class Fun
defined class Call
```

We consider a `Fun` constructor for representing JavaScripty function literals. This version of `Fun` allows for named functions. When a function expression $x(y) \Rightarrow e'$ has a name, then it is can be recursive. As noted previously about recursive functions (Section 19.5), variable x is an additional formal parameter, and the function body e' may have free variable uses of x. The variable x gets bound to itself (i.e., the function value for $x(y) \Rightarrow e'$) on a function call.

In the abstract syntax representation, the `xopt: Option[String]` parameter in our `Fun` constructor is `None` if there is no identifier present (cannot be used recursively), or `Some(x: String)` if there is an identifier, `x`, present (can be used recursively).

In this lab, we will do two things. First, we will move to implementing a small-step interpreter with a function `step` that takes an `e: Expr` and returns a one-step reduction of `e`. A small-step interpreter makes explicit the evaluation order of expressions. Second, we will remove environments and instead use a language semantics based on substitution. This change will result in static, lexical scoping without needing closures, thus demonstrating another way to fix dynamical scoping.

These two changes are orthogonal, that is, one could implement a big-step interpreter using substitution (as in Section 19.4) or a small-step interpreter using environments. Substitution is a fairly simple way to get lexical scoping, but in practice, it is rarely used because it is not the most efficient implementation.

22.2 Static Scoping

Exercise 22.1 (Substitute)**.** Since our implementation requires substitution, we begin by implementing `substitute`, which substitutes value `v` for all *free* occurrences of variable `x` in expression `e`:

```
def substitute(e: Expr, v: Expr, x: String): Expr = ???
```

```
defined function substitute
```

We advise defining `substitute` by induction on `e`. The cases to be careful about are `ConstDecl` and `Fun` because these are the variable binding constructs (as discussed in the reading on substitution in Section 21.9). In particular, calling `substitute` on expression

```
a; { const a = 4; a }
```

with value `3` for variable `a` should return

```
3; { const a = 4; a }
```

not

```
3; { const a = 4; 3 }
```

This function is a helper for the `step` function, but you might want to implement all of the cases of `step` that do not require `substitute` first.

22.3 Iteration

Our `step` performs a single reduction step. We may want to test it by repeatedly calling it with an expression until reducing to value. Thus, from a software engineering standpoint, you might want to evolve the `iterate` function described below together with your implementation of `step`.

This idea of repeatedly performing an action until some condition is satisfied is a loop or iteration. We have seen that we can iterate with a tail-recursive helper function. For example, consider the `sumTo` function that sums the integers from `0` to `n`:

```
def sumTo(n: Int): Int = {
  def loop(acc: Int, i: Int): Int = {
    require(n >= 0)
    if (i > n) acc
    else loop(acc + i, i + 1)
  }
  loop(0, 0)
}
sumTo(100)
```

```
defined function sumTo
res2_1: Int = 5050
```

This pattern of repeating something until a condition is satisfied is exceedingly common (e.g., computing the square root using Newton-Raphson approximation until the error is small enough from a previous assignment).

Because this pattern is so common, we want to get practice refactoring this pattern into a library function. This library function will be a higher-order function because it takes the "something" (i.e., what to do in each loop iteration) as a function parameter.

Exercise 22.2 (Iterate with Error Side-Effects)**.** Implement the generic, higher-order library function `iterateBasic`. The `iterateBasic` function repeatedly calls (i.e., iterates) the callback `stepi` until it returns `None` starting from `acc0: A`. Note that `iterateBasic` is generic in the accumulation type `A`. The `stepi` callback takes the current accumulator of type `A` and the iteration number as an `Int` and indicates continuing by returning `Some(acc)` for some next accumulator value `acc`.

```
def iterateBasic[A](acc0: A)(stepi: (A, Int) => Option[A]): A = {
  def loop(acc: A, i: Int): A = ???
  loop(acc0, 0)
}
```

```
defined function iterateBasic
```

We can test `iterateBasic` by using it with a client like `sumTo`:

```
def sumTo(n: Int) = {
  iterateBasic(0) { case (acc, i) =>
    require(n >= 0)
    if (i > n) None
```

```
    else Some(acc + i)
  }
}
sumTo(100)
```

We see how `sumTo` can use `iterateBasic`.

Exercise 22.3 (Iterate with Error Values)**.** One unfortunate aspect of the above is that `sumTo` "exits `iterateBasic` with an error" by throwing an exception (i.e., with the `require(n >= 0)`). Let us refactor `iterateBasic` to allow for explicit error values using `Either[Err, A]`:

```
def iterate[Err, A](acc0: A)(stepi: (A, Int) => Option[Either[Err, A]]): Either[Err, A] = {
  def loop(acc: A, i: Int): Either[Err, A] = ???
  loop(acc0, 0)
}

def sumTo(n: Int): Either[IllegalArgumentException, Int] = {
  iterate(0) { case (acc, i) =>
    if (n < 0) Some(Left(new IllegalArgumentException("requirement failed")))
    else if (i > n) None
    else Some(Right(acc + i))
  }
}
sumTo(100)
sumTo(-1)
```

The `iterate` is now parametrized by an error type `Err` and returns an `Either[Err, A]`. The `stepi` callback should return `None` if it wants to stop normally, `Some(Left(err))` if it wants to stop with an error, and `Some(Right(acc)` if it wants to continue with an accumulator value `acc`.

We can now see how we can use `iterate` as a library function to iterate your `step` implementation. In particular, this is how `iterate` will be used to iterate `step` while adding some debugging output:

```
def iterateStep(e: Expr) = {
  require(closed(e), s"iterateStep: ${e} not closed")
  if (debug) {
    println("------------------------------------------------------------")
    println("Evaluating with step ...")
  }
  val v = iterate(e) { (e: Expr, n: Int) =>
```

```
    if (debug) { println(s"Step $n: $e") }
    if (isValue(e)) None else Some(step(e))
  }
  if (debug) { println("Value: " + v) }
  v
}
```

Of particular interest is the anonymous function passed to `iterate` that calls your implementation of `step`.

22.4 Small-Step Interpreter

In this section, we implement the one-step evaluation judgment form $e \longrightarrow r$ that says, "Expression e can take one step of evaluation to a step-result r."

$$\text{step-results} \quad r \quad ::= \quad \mathsf{typeerror}\, e \mid e'$$

A step-result r is either a $\mathsf{typeerror}\, e$ indicating a dynamic type error in attempting to reduce e or a successful one-step reduction to an expression e'.

We represent a step-result r in Scala using a type `Either[DynamicTypeError, Expr]`:

```
case class DynamicTypeError(e: Expr) {
  override def toString = s"TypeError: in expression $e"
}
type Result = Either[DynamicTypeError, Expr] // r ::= typeerror e | e
```

```
defined class DynamicTypeError
defined type Result
```

Note that unlike before, `DynamicTypeError` is not an `Exception`, so it cannot be `throw`n.

The small-step semantics that we should implement are given in the section below (Section 22.5). The language we implement is JavaScripty with numbers, booleans, strings, **undefined**, printing, and first-class functions. It is a simpler language than the previous lab because we remove type coercions (except to booleans) and replace most coercion cases with dynamic type error $\mathsf{typeerror}\, e$.

Exercise 22.4 (Step without Dynamic Type Checking)**.** We advise first implementing the cases restricted to judgments of the form $e \longrightarrow e'$, that is, implement the Do and Search rules while ignoring the TypeError and Propagate rules. Start with implementing a `stepBasic` function with type:

```
def stepBasic(e: Expr): Expr = ???
```

```
defined function stepBasic
```

That is, just crash with a `MatchError` exception if your `step` encounters any ill-typed expression `e`.

The suggested practice here is to read some rules, write a few tests for those rules, and implement the cases for those tests.

Exercise 22.5 (Step with Dynamic Type Checking)**.** Then, copy your code from `stepBasic` to `stepCheck`:

```
def stepCheck(e: Expr): Either[DynamicTypeError, Expr] = ???
```

```
defined function stepCheck
```

to then add dynamic type checking. You will likely need to refactor your code to satisfy the new types before implementing the TypeError and Propagate rules.

Exercise 22.6 (To Boolean)**.** You will need to implement a `toBoolean` function to convert JavaScripty values to booleans, following the ToBoolean rules in Section 22.5.

```
def toBoolean(e: Expr): Boolean = ???
```

```
defined function toBoolean
```

However, you will not need any other type coercion functions here.

Notes

- Note that the tests call the `step` function that is originally defined as:

```
//def step(e: Expr): Either[DynamicTypeError, Expr] = Right(stepBasic(e))
def step(e: Expr): Either[DynamicTypeError, Expr] = stepCheck(e)
```

You can first test `stepBasic` by uncommenting the first line and commenting out the second line.

- Note that the provided tests are minimal. You will want to add your own tests to cover most language features.

22.5 Small-Step Operational Semantics

In this section, we give the small-step operational semantics for JavaScripty with numbers, booleans, strings, **undefined**, printing, and first-class functions. We have type coercions to booleans but otherwise use dynamic type error for other cases.

We write $[v/x]e$ for substituting value v for all free occurrences of the variable x in expression e (i.e., a call to `substitute`).

It is informative to compare the small-step semantics used in this lab and the big-step semantics from last homework.

22.5.1 Do Rules

22.5.2 Search Rules

22.5.3 Coercing to Boolean

22.5.4 Dynamic Typing Rules

22.6 Concept Exercises

Make sure to complete the concept exercises in this section and turn in this file as part of your submission. However, all code and testing exercises from other sections are submitted in `Lab3.scala` or `Lab3Spec.scala`.

Exercise 22.7 (Evaluation Order)**.** Consider the small-step operational semantics shown in Section 22.5. What is the evaluation order for $e_1 + e_2$? Explain.

Edit this cell:

???

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

$$\textsc{DoNeg}\ \frac{n' = -n_1}{\texttt{-}\, n_1 \longrightarrow n'} \qquad \textsc{DoArith}\ \frac{n' = n_1\ \mathit{bop}\ n_2 \qquad \mathit{bop} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{n_1\ \mathit{bop}\ n_2 \longrightarrow n'} \qquad \textsc{DoPlusString}\ \frac{\mathit{str}' = \mathit{str}_1 \mathit{str}_2}{\mathit{str}_1\ \texttt{+}\ \mathit{str}_2 \longrightarrow \mathit{str}'}$$

$$\textsc{DoInequalityNumber}\ \frac{b' = n_1\ \mathit{bop}\ n_2 \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{n_1\ \mathit{bop}\ n_2 \longrightarrow b'} \qquad \textsc{DoInequalityString}\ \frac{b' = \mathit{str}_1\ \mathit{bop}\ \mathit{str}_2 \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\mathit{str}_1\ \mathit{bop}\ \mathit{str}_2 \longrightarrow b'}$$

$$\textsc{DoEquality}\ \frac{b' = (v_1\ \mathit{bop}\ v_2) \qquad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{v_1\ \mathit{bop}\ v_2 \longrightarrow b'} \qquad \textsc{DoNot}\ \frac{v_1 \rightsquigarrow b_1}{\texttt{!}\ v_1 \longrightarrow \neg b_1} \qquad \textsc{DoAndTrue}\ \frac{v_1 \rightsquigarrow \mathbf{true}}{v_1\ \texttt{\&\&}\ e_2 \longrightarrow e_2} \qquad \textsc{DoAndFalse}\ \frac{v_1 \rightsquigarrow \mathbf{false}}{v_1\ \texttt{\&\&}\ e_2 \longrightarrow v_1}$$

$$\textsc{DoOrTrue}\ \frac{v_1 \rightsquigarrow \mathbf{true}}{v_1\ \texttt{||}\ e_2 \longrightarrow v_1} \qquad \textsc{DoOrFalse}\ \frac{v_1 \rightsquigarrow \mathbf{false}}{v_1\ \texttt{||}\ e_2 \longrightarrow e_2} \qquad \textsc{DoIfTrue}\ \frac{v_1 \rightsquigarrow \mathbf{true}}{v_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \longrightarrow e_2} \qquad \textsc{DoIfFalse}\ \frac{v_1 \rightsquigarrow \mathbf{false}}{v_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \longrightarrow e_3}$$

$$\textsc{DoSeq}\ \frac{}{v_1\ \texttt{,}\ e_2 \longrightarrow e_2} \qquad \textsc{DoPrint}\ \frac{v_1 \text{ printed}}{\texttt{console.log(}v_1\texttt{)} \longrightarrow \mathbf{undefined}} \qquad \textsc{DoConst}\ \frac{}{\mathbf{const}\ x\ \texttt{=}\ v_1\texttt{;}\ e_2 \longrightarrow [v_1/x]e_2}$$

$$\textsc{DoCall}\ \frac{}{\texttt{((}x\texttt{)}\ \texttt{=>}\ e_1\texttt{)(}v_2\texttt{)} \longrightarrow [v_2/x]e_1} \qquad \textsc{DoCallRec}\ \frac{v_1 = (x_1\texttt{(}x_2\texttt{)}\ \texttt{=>}\ e_1)}{v_1\texttt{(}v_2\texttt{)} \longrightarrow [v_1/x_1][v_2/x_2]e_1}$$

Figure 22.1: The Do rules for JavaScripty with numbers, booleans, strings, **undefined**, printing, and first-class functions.

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

$$\textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{\mathit{uop}\, e_1 \longrightarrow \mathit{uop}\, e_1'} \qquad \textsc{SearchBinary1}\ \frac{e_1 \longrightarrow e_1'}{e_1\ \mathit{bop}\ e_2 \longrightarrow e_1'\ \mathit{bop}\ e_2} \qquad \textsc{SearchBinary2}\ \frac{e_2 \longrightarrow e_2'}{v_1\ \mathit{bop}\ e_2 \longrightarrow v_1\ \mathit{bop}\ e_2'}$$

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

$$\textsc{SearchConst}\ \frac{e_1 \longrightarrow e_1'}{\mathbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \longrightarrow \mathbf{const}\ x\ \texttt{=}\ e_1'\texttt{;}\ e_2} \qquad \textsc{SearchCall1}\ \frac{e_1 \longrightarrow e_1'}{e_1\texttt{(}e_2\texttt{)} \longrightarrow e_1'\texttt{(}e_2\texttt{)}} \qquad \textsc{SearchCall2}\ \frac{e_2 \longrightarrow e_2'}{v_1\texttt{(}e_2\texttt{)} \longrightarrow v_1\texttt{(}e_2'\texttt{)}}$$

Figure 22.2: The Search rules for JavaScripty with numbers, booleans, strings, **undefined**, printing and first-class functions.

$$\boxed{v \rightsquigarrow b}\qquad \frac{n \in \{0.0, -0.0, \mathsf{NaN}\}}{n \rightsquigarrow \mathbf{false}}\ \textsc{ToBooleanNumFalse} \qquad \frac{n \notin \{0.0, -0.0, \mathsf{NaN}\}}{n \rightsquigarrow \mathbf{true}}\ \textsc{ToBooleanNumTrue} \qquad \frac{}{b \rightsquigarrow b}\ \textsc{ToBooleanBoolean}$$

$$\frac{}{\texttt{""} \rightsquigarrow \mathbf{false}}\ \textsc{ToBooleanStrFalse} \qquad \frac{\mathit{str} \neq \texttt{""}}{\mathit{str} \rightsquigarrow \mathbf{true}}\ \textsc{ToBooleanStrTrue} \qquad \frac{}{\mathbf{undefined} \rightsquigarrow \mathbf{false}}\ \textsc{ToBooleanUndefined}$$

$$\frac{}{x^{?}(y) \Rightarrow e \rightsquigarrow \mathbf{true}}\ \textsc{ToBooleanFun}$$

Figure 22.3: The ToBoolean rules for JavaScripty with numbers, booleans, strings, **undefined**, and first-class functions.

Exercise 22.8 (Changing Evaluation Order)**.** How do we change the rules to obtain the opposite evaluation order?

Edit this cell:

???

Exercise 22.9 (Using Short-Circuit Evaluation)**.** Give an example that illustrates the usefulness of short-circuit evaluation. Explain your example.

Edit this cell:

???

Exercise 22.10 (Removing Short-Circuit Evaluation)**.** Consider the small-step operational semantics shown in Section 22.5. Does e_1 `&&` e_2 short circuit? Explain. If e_1 `&&` e_2 short circuits, give rules that eliminates short circuiting. If it does not short circuit, give the short-circuiting rules.

Edit this cell:

???

$$\boxed{e \longrightarrow r}$$

$$\textsc{TypeErrorNeg}\quad \frac{v_1 \neq n_1}{\texttt{-}\, v_1 \longrightarrow \mathsf{typeerror}(\texttt{-}\, v_1)}$$

$$\textsc{TypeErrorPlus1}\quad \frac{v_1 \neq n_1 \qquad v_1 \neq \mathit{str}_1}{v_1 \,\texttt{+}\, v_2 \longrightarrow \mathsf{typeerror}(v_1 \,\texttt{+}\, v_2)}$$

$$\textsc{TypeErrorPlusString2}\quad \frac{v_2 \neq \mathit{str}_2}{\mathit{str}_1 \,\texttt{+}\, v_2 \longrightarrow \mathsf{typeerror}(\mathit{str}_1 \,\texttt{+}\, v_2)}$$

$$\textsc{TypeErrorArith1}\quad \frac{v_1 \neq n_1 \qquad \mathit{bop} \in \{\texttt{-}, \texttt{*}, \texttt{/}\}}{v_1\ \mathit{bop}\ v_2 \longrightarrow \mathsf{typeerror}(v_1\ \mathit{bop}\ v_2)}$$

$$\textsc{TypeErrorArith2}\quad \frac{v_2 \neq n_2 \qquad \mathit{bop} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{n_1\ \mathit{bop}\ v_2 \longrightarrow \mathsf{typeerror}(n_1\ \mathit{bop}\ v_2)}$$

$$\textsc{TypeErrorInequality1}\quad \frac{v_1 \neq n_1 \qquad v_1 \neq \mathit{str}_1 \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{v_1\ \mathit{bop}\ v_2 \longrightarrow \mathsf{typeerror}(v_1\ \mathit{bop}\ v_2)}$$

$$\textsc{TypeErrorInequalityNumber2}\quad \frac{v_2 \neq n_2 \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{n_1\ \mathit{bop}\ v_2 \longrightarrow \mathsf{typeerror}(n_1\ \mathit{bop}\ v_2)}$$

$$\textsc{TypeErrorInequalityString2}\quad \frac{v_2 \neq \mathit{str}_2 \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\mathit{str}_1\ \mathit{bop}\ v_2 \longrightarrow \mathsf{typeerror}(\mathit{str}_1\ \mathit{bop}\ v_2)}$$

$$\textsc{TypeErrorCall}\quad \frac{v_1 \neq x^?\texttt{(}y\texttt{)} \Rightarrow e_1}{v_1\texttt{(}v_2\texttt{)} \longrightarrow \mathsf{typeerror}(v_1\texttt{(}v_2\texttt{)})}$$

$$\textsc{PropagateUnary}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{\mathit{uop}\ e_1 \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateBinary1}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{e_1\ \mathit{bop}\ e_2 \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateBinary2}\quad \frac{e_2 \longrightarrow \mathsf{typeerror}\ e}{v_1\ \mathit{bop}\ e_2 \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateIf}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagatePrint}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{\texttt{console.log(}e_1\texttt{)} \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateConst}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{\mathbf{const}\ x \,\texttt{=}\, e_1\texttt{;}\ e_2 \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateCall1}\quad \frac{e_1 \longrightarrow \mathsf{typeerror}\ e}{e_1\texttt{(}e_2\texttt{)} \longrightarrow \mathsf{typeerror}\ e}$$

$$\textsc{PropagateCall2}\quad \frac{e_2 \longrightarrow \mathsf{typeerror}\ e}{v_1\texttt{(}e_2\texttt{)} \longrightarrow \mathsf{typeerror}\ e}$$

Figure 22.4: The TypeError and Propagate rules for JavaScripty with numbers, booleans, strings, **undefined**, printing, and first-class functions.

22.7 Testing

This section has some space to write some tests in our subset of JavaScript. You might want to work on these tests while you are implementing `step`. As before, you will add your tests to `Lab3StudentSpec`. Your interpreter will run the tests against the expected result you provide. We will write three tests, all of these tests must properly parse.

Exercise 22.11 (Test 1: Higher-Order Function)**.** Write a test case that has a function that takes a function value as an argument (i.e., is a higher order function):

```
???
```

Exercise 22.12 (Test 2: Recursion)**.** Write a test case that uses recursion

```
???
```

Exercise 22.13 (Test 3: Any Test in this variant of JavaScripty)**.** Write another test

```
???
```

Notes

- Remember to add these to `Lab3StudentSpec` in `Lab3Spec.scala`.
- Add the JavaScripty code as a string in `jsyStr` and the expected result in `answer`.

22.8 Accelerated Component

For the accelerated component of this lab, we will give rules and implement the behavior that enables us to match JavaScript semantics. In particular, we will give rules and implement type coercions for numbers and strings, and we will update our small-step operational semantics to use them.

22.8.1 Additional Type Coercions

Exercise 22.14. Give the inference rules defining the judgment form for coercing a value to a number $v \rightsquigarrow n$

Edit this cell:

???

Exercise 22.15. Give the inference rules defining the judgment form for coercing a value to a string $v \rightsquigarrow str$

Edit this cell:

???

Exercise 22.16. Implement the `toNumber` and `toStr`coercions.

```
def toNumber(v: Expr): Double = ???
def toStr(v: Expr): String = ???
```

```
defined function toNumber
defined function toStr
```

Notes

- If your recall Lab 2, we implemented these functions. They will be the same here, except we must add the rules for functions.

22.8.2 Updating the Small-Step Operational Semantics

Now that we are allowing type coersions, our operational semantics will change. For example, consider the following Do rule:

$$\textsc{DoNeg}\ \frac{v \rightsquigarrow n}{\texttt{-}\,v \longrightarrow -n}$$

This is a new rule for DoNeg, which is read as if v coerces to n, then $\texttt{-}\,v$ steps to $-n$. Now that we are allowing non-numbers to be coerced and then negated, we no longer have our TypeErrorNeg rule.

Note that we only need to update Do rules with coercions and remove TypeError rules. We do not need to update the Search or Propagage rules.

Exercise 22.17. Explain why we only need to update the Do rules with coercions and remove TypeError rules, and we do not need to update the Search or Propagage rules.

Edit this cell:

???

One rule of particular interest is DoArith, which we need to split to account for $e_1 + e_2$ being overloaded for numbers and strings. Given this, we need to rewrite DoArith so it does not include + (and adds coersions), add the rule DoPlusNumber, and alter DoPlusString to become two rules.

Exercise 22.18. Give these new Do rules

Edit this cell:

???

Similar to this, our DoInequalityNumber rule must be split into two and altered:

Exercise 22.19. Give the two new $\text{DoInequalityNumber}_1$ and $\text{DoInequalityNumber}_2$ rules:

Edit this cell:

???

Once we have all of these rules defined, we notice that most of our typeerror rules no longer result in type errors. Therefore, most of them should be deleted. In fact, the only non-propagate rule left for type errors is TypeErrorCall, since we are still not allowed to call something that is not a function (in JavaScript).

22.8.3 Update Step

Exercise 22.20. Now that we have our type conversion functions implemented and our new rules defined, we are ready to update `step`. Implement `stepCoerce` by first copying your code from `stepCheck` and then update based on your new rules.

```
def stepCoerce(e: Expr): Either[DynamicTypeError,Expr] = ???
```

```
defined function stepCoerce
```

As before, let the rules guide your implementation.

Notes

- None of your other functions should need to be altered.

Submission

If you are a University of Colorado Boulder student, we use Gradescope for assignment submission. In summary,

- ☐ Create a private GitHub repository by clicking on a GitHub Classroom link from the corresponding Canvas assignment entry.
- ☐ Clone your private GitHub repository to your development environment (using the `<> Code` button on GitHub to get the repository URL).
- ☐ Work on this lab from your cloned repository. Use Git to save versions on GitHub (e.g., `git add`, `git commit`, `git push` on the command line or via VSCode).
- ☐ Submit to the corresponding Gradescope assignment entry for grading by choosing GitHub as the submission method.

You need to have a GitHub identity and must have your full name in your GitHub profile in case we need to associate you with your submissions.

23 Review: Semantics

Instructions

This assignment is a review exercise in preparation for a subsequent assessment activity.

This is a peer-quizzing activity with two students. Each section has an even number of exercises. Student A quizzes Student B on the odd numbered exercises, and Student B quizzes Student A on the even numbered exercises.

To the best of your ability, give feedback using the learning-levels rubric below on where your peer is in reaching or exceeding Proficient (P) on each question live. Guidance of what a Proficient (P) answer looks like are given.

There may or may not be a member of the course staff assigned to your slot. It is expected that regardless of whether a member of the course staff is present, this is a peer-quizzing activity. If a member of the course staff is present, you may ask for their help and guidance on answering the questions and/or their assessment of where you are at in your learning level.

It is not expected that you can complete all exercises in the allotted time. You and your partner may pick and choose which sections you want to focus on and use the remaining questions as a study guide. You and your partner may, of course, continue working together after the scheduled session.

At the same time, most questions can be answered in a few minutes with a Proficient (P) level of understanding. Aim for 3–4 sections in 30 minutes.

Your submission for this session is an overall assessment of where your partner is in their reaching-or-exceeding-proficiency level. Be constructive and honest. **Neither your nor your partners grade will depend on your learning-level assessment.** Instead, your score for this assignment will be based on the thoughtfulness of your feedback to your partner.

Submit on Gradescope as a pair. That is, use Gradescope's group assignment feature to submit as a group. The submission form has a spot for each of you to provide your assessment and feedback for each other.

Please proactively fill slots with an existing sign-up to have a partner. In case your peer does not show up to the slot, try to join another slot happening at the same time from the course calendar. If that fails and a course staff member is present, you may do the exercise with the staff member and get credit. If there is no staff member present, you may try to find a slot at a later time if you like or else write to the Course Manager on Piazza timestamped during the slot.

Learning-Levels Rubric

4 - Exceeding (E) Student demonstrates synthesis of the underlying concepts. Student can go beyond merely describing the solution to explaining the underlying reasoning and discussing generalizations.

3 - Proficient (P) Student is able to explain the overall solution and can answer specific questions. While the student is capable of explaining their solution, they may not be able to confidently extend their explanation beyond the immediate context.

2 - Approaching (A) Student may able to describe the solution but has difficulty answering specific questions about it. Student has difficulty explaining the reasoning behind their solution.

1 - Novice (N) Student has trouble describing their solution or responding to guidance. Student is unable to offer much explanation of their solution.

23.1 Dynamic versus Static Scoping

Exercise 23.1. Consider the following JavaScripty code. What is the resulting value you would expect from an interpreter that implements dynamic scoping? What about one that implements static scoping? Explain.

```
const x = 4;
const f = (y) => x * 2;
((x) => f(5))(8)
```

A Proficient (P) answer recognizes that the resulting value is different under dynamic versus static scoping. It discusses that this difference is due to the fact that an interpreter that implements dynamic scoping would evaluate the function `(y) => x * 2` (which is bound to `f`) with the value environment at the time it is called, instead of the environment at the time it is defined. Static scoping, which is what we usually expect, evaluates functions with the value environment in which they were defined in.

An Exceeding (E) answer also explains how dynamic scoping arises accidentally in an interpreter that uses value environments, perhaps by giving an operational semantics rule for function call that exhibits dynamic scoping.

Exercise 23.2. Consider the following inference rules which define a big-step or small-step operational semantics for non-recursive function literals and function call expressions. For each one, write if it implements dynamic scoping or static scoping. Choose one rule to explain fully. In this explanation, discuss why the rule does or does not implement dynamic scoping. Also write out what each aspect of the rule is stating.

1.

$$\boxed{e \longrightarrow e'} \qquad \frac{}{\texttt{((}x\texttt{)} \Rightarrow e_1\texttt{)(}v_2\texttt{)} \longrightarrow [v_2/x]e_1}\ \textsc{DoCall} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\texttt{(}e_2\texttt{)} \longrightarrow e_1'\texttt{(}e_2\texttt{)}}\ \textsc{SearchCall1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\texttt{(}e_2\texttt{)} \longrightarrow v_1\texttt{(}e_2'\texttt{)}}\ \textsc{SearchCall2}$$

2.

$$\boxed{E \vdash e \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow \texttt{(}x\texttt{)} \Rightarrow e' \qquad E \vdash e_2 \Downarrow v_2 \qquad E[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1\texttt{(}e_2\texttt{)} \Downarrow v'}\ \textsc{EvalCall}$$

3.

$$\boxed{E \vdash e \Downarrow v} \qquad \frac{}{E \vdash \texttt{(}x\texttt{)} \Rightarrow e \Downarrow \texttt{(}x\texttt{)} \Rightarrow e[E]}\ \textsc{EvalFun}$$

$$\frac{E \vdash e_1 \Downarrow \texttt{(}x\texttt{)} \Rightarrow e'[E'] \qquad E \vdash e_2 \Downarrow v_2 \qquad E'[x \mapsto v_2] \vdash e' \Downarrow v'}{E \vdash e_1\texttt{(}e_2\texttt{)} \Downarrow v'}\ \textsc{EvalCall}$$

4.

$$\boxed{e \Downarrow v} \qquad \frac{e_1 \Downarrow \texttt{(}x\texttt{)} \Rightarrow e' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e' \Downarrow v'}{e_1\texttt{(}e_2\texttt{)} \Downarrow v'}\ \textsc{EvalCall}$$

A Proficient (P) answer correctly states which rules implement dynamic scoping (2) and which do not (1, 3, 4). For the explanation, it explains what general strategy is used (e.g., big-step with value environments and closures) and why or why not does this strategy result in dynamic scoping. For example, using closures results in static scoping because the value environment in which the function was defined is saved in the closure. When functions are called, the value environment from the closure is extended with the parameter, and then the body is evaluated. A Proficient (P) answer also correctly states each aspect of the premise and conclusion of the rule.

23.2 Small-Step Semantics with Coercions

We consider a subset of JavaScripty that includes variables (x), variable binding (**const**), arithmetic plus (`+`), arithmetic negation (`-`), boolean conjunction (`&&`), and boolean negation (`!`). The values of our language are numbers (n) and booleans (b). Below is the abstract syntax.

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & x \mid \textbf{const}\ x \texttt{ = } e_1 \texttt{;}\ e_2 \mid e_1\ \mathit{bop}\ e_2 \mid \mathit{uop}\, e_1 \mid n \mid b \\
\text{binary operators} & \mathit{bop} & ::= & \texttt{+} \mid \texttt{\&\&} \\
\text{unary operators} & \mathit{uop} & ::= & \texttt{-} \mid \texttt{!} \\
\text{values} & v & ::= & n \mid b \\
\text{variables} & x & &
\end{array}
$$

Figure 23.1: Syntax for JavaScripty with variables and some minimal arithmetic and boolean expressions.

Our rules for coercing a value to a number are as follows.

$$
\boxed{v \rightsquigarrow n}
\qquad
\frac{}{n \rightsquigarrow n}\,\textsc{ToNumberNum}
\qquad
\frac{}{\textbf{true} \rightsquigarrow 1}\,\textsc{ToNumberTrue}
\qquad
\frac{}{\textbf{false} \rightsquigarrow 0}\,\textsc{ToNumberFalse}
$$

Exercise 23.3. Define the judgment form $v \rightsquigarrow b$. That is, write the inference rules for coercing a value to a boolean. Recall `0` coerces to **false**, and anything else coerces to **true**.

> A Proficient (P) answer correctly gives two rules. All rules are not inductive (i.e., do not have a $v \rightsquigarrow b$ judgment in the premise). For clarity, the judgment form should be given in a box above the rules.

Exercise 23.4. Write the Search and Do rules for stepping a **const** x `=` e_1 `;` e_2 expression and a variable-use expression x. Our small-step judgment form is $e \longrightarrow e'$ that says, "Closed expression e reduces to closed-expression e' in one step."

> A Proficient (P) answer writes two correct rules. The rules are DoConst and SearchConst. DoConst should require that e_1 is a value, then step to e_2 with a substitution for the binding of variable x.
>
> An Exceeding (E) answer writes all of the rules correctly, recognizes why we do not need a Do rule for variable uses, and understands why substitution enables us to maintain the invariant that e and e' in $e \longrightarrow e'$ are closed expressions. There is no Do rule for variable uses because expression e must be a closed expression.

$$\boxed{e \longrightarrow e'} \qquad \frac{\textsc{DoNeg}}{} \frac{v_1 \rightsquigarrow n_1}{\texttt{-}\, v_1 \longrightarrow -n_1} \qquad \textsc{DoNot}\ \frac{v_1 \rightsquigarrow b_1}{\texttt{!}\, v_1 \longrightarrow \neg b_1} \qquad \textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{\mathit{uop}\, e_1 \longrightarrow \mathit{uop}\, e_1'}$$

Figure 23.2: A small-step operational semantics for the unary expressions.

Below is a small-step operational semantics for stepping unary expressions:

Exercise 23.5. Complete the inductive definition of $e \longrightarrow e'$ by writing the Search and Do rules for stepping binary expressions. You need to write two Search rules and two Do rules.

> An Proficient (P) answer writes four or five rules. There should be two Search rules that should include a premise that one side of the binary expression takes a step. The Do rules should probably use coercions following the example for unary expressions (e.g., $v_1 \rightsquigarrow n_1$).
>
> An Exceeding (E) answer will consider different semantic choices. For the Search rules, the answers can consider left-to-right evaluation, right-to-left evaluation, or non-deterministic evaluation. Such an answer will explain that a deterministic left-to-right or right-to-left would require the other side of the binary expression is already a value. For the DoAnd rule(s), an Exceeding (E) answer will consider whether to implement short-circuiting or not.

Note that one possible correct solution for the rules asked about in the above exercises are given in the preceding chapters (cf. Chapter 21 or Section 22.5)

Exercise 23.6. Consider the following expression e_0:

$$\textbf{const}\ \texttt{h} = \textbf{true}\texttt{;}\ \texttt{(h + 3) \&\&}\ \textbf{false}$$

Then, is the judgment

$$\textbf{const}\ \texttt{h} = \textbf{true}\texttt{;}\ \texttt{(h + 3) \&\&}\ \textbf{false} \longrightarrow e_1$$

for some expression e_1 derivable using your rules? If so, give the derivation with the appropriate e_1.

Then, is the judgment $e_1 \longrightarrow e_2$ derivable using your rules? If so, give the derivation with the appropriate e_2.

Repeat until giving derivations for $e_i \longrightarrow e_{i+1}$ until the step-judgment is not derivable. Explain why this last step-judgment is not derivable.

Explain how these derivations are connected to your interpreter implementation of these rules from the lab assignment.

A Proficient (P) shows derivations for the judgments

$$\textbf{const}\ \texttt{h} = \textbf{true}\texttt{;}\ \texttt{(h + 3) \&\&}\ \textbf{false} \longrightarrow \texttt{(}\textbf{true}\ \texttt{+ 3) \&\&}\ \textbf{false}$$

$$\texttt{(}\textbf{true}\ \texttt{+ 3) \&\&}\ \textbf{false} \longrightarrow \texttt{4 \&\&}\ \textbf{false}$$

$$\texttt{4 \&\&}\ \textbf{false} \longrightarrow \textbf{false}$$

It should state that there is no deriviation for the judgment **false** $\longrightarrow e_4$ for any expression e_4 because the step-judgment form defines a reduction step and **false** is a value. Note that a Proficient (P) answer may give different judgments here corresponding to a different evaluation-order semantics. The particular judgments should correspond to the given rules. Each of the derivations has some number of SEARCH rule applications and ends with a DO rule application as an axiom (reading from the bottom up).

Regarding the interpreter implementation, a Proficient (P) answer should also recognize that a judgment $e_i \longrightarrow e_{i+1}$ corresponds to a call of `step`. An Exceeding (E) answer recognizes that each application of a SEARCH rule corresponds to a recursive call of `step` and observes that there is only one recursive call in each SEARCH rule to find the redex in which to apply a DO rule.

23.3 Short-Circuit Evaluation and Evaluation Order

Exercise 23.7. Does the rule you wrote in Exercise 23.5 for `&&` short circuit? Explain why or why not. If it does, rewrite the rule so that it does not short circuit. If it does not, rewrite it so it does short circuit.

A Proficient (P) answer understands why `&&` does or does not short-circuit. This is determined by the DOAND rule(s). If DOAND rule(s) do not require each sub-expression to be a value before eliminating the `&&` operator, then it does short circuit. If both sub-expressions are required to be a value before eliminating the `&&` operator, then it does not short-circuit. Note that the direction that the DOAND rules depends on the way the SEARCHBINARY rules implement evaluation order.

Exercise 23.8. What is the evaluation order of the search rules you wrote in Exercise 23.5? Explain. Write new rules that changes the evaluation order (e.g., from left-to-right to right-to-left).

A Proficient (P) answer understands how SEARCH rules determine evaluation order. If SEARCHBINARY2 requires e_1 to be a value (v_1) in order to step e_2, then the evaluation order is left-to-right. This is assuming SEARCHBINARY1 is written correctly by not requiring either to be a value. These rules force e_1 to be evaluated before e_2 in a binary expression.

23.4 Big-Step Semantics with Substitution and Dynamic Type Errors

In the preceding chapters, we considered a big-step semantics with environments and a small-step semantics with substitution. We noted that these are orthogonal considerations, so in this section, consider one of the alternatives, namely big-step semantics with substitution.

We consider a subset of JavaScripty that includes potentially recursive and non-recursive function literals, function calls, and the binary expressions from the language described in Figure 23.1. The abstract syntax is as follows:

$$
\begin{array}{rrcl}
\text{values} & v & ::= & n \mid b \mid x^{?}\,\texttt{(}y\texttt{)}\ \texttt{=>}\ e_1 \\
\text{expressions} & e & ::= & x \mid n \mid b \mid x^{?}\,\texttt{(}y\texttt{)}\ \texttt{=>}\ e_1 \mid e_1\ bop\ e_2 \mid e_1\texttt{(}e_2\texttt{)} \\
\text{binary operators} & bop & ::= & \texttt{\&\&} \mid \texttt{+} \\
\text{optional variables} & x^{?} & ::= & x \mid \varepsilon \\
\text{variables} & x & &
\end{array}
$$

Below are the inference rules for evaluating values, binary expressions, and non-recursive function calls, that is, all except for potentially recursive function calls.

$$
\boxed{e \Downarrow v}
\qquad
\textsc{EvalVal}\ \frac{}{v \Downarrow v}
\qquad
\textsc{EvalPlus}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1\ \texttt{+}\ e_2 \Downarrow n_1 + n_2}
\qquad
\textsc{EvalAndTrue}\ \frac{e_1 \Downarrow \textbf{true} \qquad e_2 \Downarrow b_2}{e_1\ \texttt{\&\&}\ e_2 \Downarrow b_2}
$$

$$
\textsc{EvalAndFalse}\ \frac{e_1 \Downarrow \textbf{false}}{e_1\ \texttt{\&\&}\ e_2 \Downarrow \textbf{false}}
\qquad
\textsc{EvalCall}\ \frac{e_1 \Downarrow \texttt{(}y\texttt{)}\ \texttt{=>}\ e' \qquad e_2 \Downarrow v_2 \qquad [v_2/y]e' \Downarrow v'}{e_1\texttt{(}e_2\texttt{)} \Downarrow v'}
$$

Figure 23.3: A big-step operational semantics for JavaScripty with function literals, function calls, and some binary expressions.

Exercise 23.9. Consider the rules in Figure 23.3, why do we not need a rule for evaluating variables?

A Proficient (P) answer understands that we do not need a rule for evaluating variables because we are using substitution. Variable uses will always be replaced by their values using substitute.

An Exceeding (E) answer also recognizes that we are requiring all of our expressions to be closed in our evaluation judgment form $e \Downarrow v$. Considering an implementation, open expressions would lead to a match error for not matching on a top-level `Var`.

Exercise 23.10. Is the judgment

$$\texttt{((x) => ((y) => y + x))(2)} \Downarrow v$$

derivable using the rules given above? If so, give the derivation. If not, explain why not.

Assume that the concrete syntax allows JavaScripty programmers to parenthesize expressions, so the expression is syntactially valid.

An Proficient (P) answer states the value v is the function value `(y) => y + 2` and gives a correct derivation with four rule applications. Reading bottom up, the root rule application is an EvalCall with three sub-derivations each being applications of EvalVal.

An Exceeding (A) answer might also first observe that the expression is well-typed, so the judgment should hold before proceeding to giving a correct derivation.

Exercise 23.11. Write the inference rule for recursive function calls.

A Proficient (P) answer gives the correct rule for recursive function calls. This rule must:

- Show that e_1 evaluates to the identified recursive function value, say v_1.
- Correctly substitute the function identifier with v_1 in the expression e_1 with the formal parameter substituted with the actual argument.
- Show that the evaluation of the above expression is what the call evaluates to.

Exercise 23.12. The Scala AST representation for our JavaScripty language has the `Var` constructor for variables, `N` for numbers, `B` for booleans, and `Fun` for functions. We will represent $x^?$ (function identifiers) with the `Option[String]` type. Implement the remainder of the following substitute function for our small-step interpreter.

```scala
trait Expr
case class Var(x: String) extends Expr
case class N(n: Double) extends Expr
case class B(b: Boolean) extends Expr
case class Fun(xopt: Option[String], y: String, e1: Expr) extends Expr
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
case class Call(e1: Expr, e2: Expr) extends Expr

trait Bop
case object And extends Bop
case object Plus extends Bop

def isValue(e: Expr) = e match {
    case N(_) | B(_) | Fun(_, _, _) => true
    case _ => false
}

def substitute(v: Expr, x: String, e: Expr): Expr = {
    def subst(e: Expr): Expr = e match {
        case Var(y) => ???
        case _ => ???
    }
    ???
}
```

```
defined trait Expr
defined class Var
defined class N
defined class B
defined class Fun
defined class Binary
defined class Call
defined trait Bop
defined object And
defined object Plus
defined function isValue
defined function substitute
```

A Proficient (P) answer implements cases for all `Expr` cases. For `Var`, the answer needs to distinguish between shadowing and not shadowing. For `Fun`, the answer needs to consider the binding of the formal parameter and the possible binding of the optional parameter for the function name. The `N` and `B` can be done in the same pattern match, though having them separate is fine. The other cases are not binding constructs, so the answer just needs to reconstruct with recursively calling `subst` on each sub-expression. Finally, the answer needs to call `subst`.

An Exceeding (E) answer implements all substitution cases correctly. It might observe that the `subst` helper function is not strictly necessary but convenient since the `v` and `x` are fixed throughout the recursion. It should also note that we have an unstated requirement that `v` is closed, so we do not need to worry about the possiblity of free-variable capture.

In Figure 23.3, we define evaluation only for well-typed terms. Let us extend the evaluation judgment form with dynamic type checking.

A dynamic type error $\textsf{typeerror}\,e$ results if the $\texttt{\&\&}$ operator is applied to numbers, if $\texttt{+}$ is applied to booleans, or if a non-function value is called. We extend our evaluation judgment form $e \Downarrow r$ to return evaluation-results that may be a $\textsf{typeerror}$ or a value:

$$\text{evaluation-results} \quad r \quad ::= \quad \textsf{typeerror}\,e \mid v$$

Some of our dynamic type error rules are given below:

$$\boxed{e \Downarrow r} \qquad \frac{e_1 \Downarrow v_1 \qquad v_1 \neq n_1}{e_1 \texttt{ + } e_2 \Downarrow \textsf{typeerror}(e_1 \texttt{ + } e_2)}\ \textsc{TypeErrorPlus1} \qquad \frac{e_2 \Downarrow v_2 \qquad v_2 \neq b_2}{e_1 \texttt{ \&\& } e_2 \Downarrow \textsf{typeerror}(e_1 \texttt{ \&\& } e_2)}\ \textsc{TypeErrorAnd2}$$

$$\frac{e_1 \Downarrow v_1 \qquad v_1 \neq x^?(y) \texttt{ => } e_1}{e_1\texttt{(}e_2\texttt{)} \Downarrow \textsf{typeerror}(e_1\texttt{(}e_2\texttt{)})}\ \textsc{TypeErrorCall} \qquad \frac{e_2 \Downarrow \textsf{typeerror}\,e}{e_1\ \mathit{bop}\ e_2 \Downarrow \textsf{typeerror}\,e}\ \textsc{PropagateBinary2}$$

Exercise 23.13. Give the missing rules TypeError and Propagate rules.

A Proficient (P) answer gives correct rules analogous to the above rules for TypeErrorPlus2, TypeErrorAnd1, PropagateBinary1, PropagateCall1, and PropagateCall2.

An Exceeding (E) answer also understands why we need the Propagate rules and can explain the issue with not having them. It also explains what would happen in an implementation if they were not given. An Exceeding (E) answer may be able to give one or two rules explicitly and could explain the others unambiguously without necessarily giving them explicitly.

Exercise 23.14. Implement enough cases of the dynamic type-checking evaluator you defined in Exercise 23.13 in Scala using the `Either[DynamicTypeError, Expr]` type for an evaluation-result r to be able to evaluate expressions in the sub-language:

$$\begin{array}{rrcl} \text{expressions} & e & ::= & n \mid b \mid e_1 \; bop \; e_2 \\ \text{binary operators} & bop & ::= & \texttt{+} \end{array}$$

```
case class DynamicTypeError(e: Expr)
def eval(e: Expr): Either[DynamicTypeError, Expr] = ???
```

```
defined class DynamicTypeError
defined function eval
```

Recall that the constructors for `Either[Err,A]` are `Left(err: Err)` and `Right(a: A)` and the `map` and `flatMap` methods transform `Right` values.

> A Proficient (P) answer recognizes that the EvalVal, EvalPlus, TypeErrorPlus1, and TypeErrorPlus2 rules, as well as PropagateBinary1 and PropagateBinary2 rules instantiated for `+` need to implemented for this sub-language:

```
def eval(e: Expr): Either[DynamicTypeError, Expr] = e match {
    // EvalVal
    case v if isValue(v) => Right(v)
    case Binary(Plus, e1, e2) => eval(e1) match {
        case Right( N(n1) ) => eval(e2) match {
            // EvalPlus
            case Right( N(n2) ) => Right( N(n1 + n2) )
            // TypeErrorPlus2
            case Right(_) => Left(DynamicTypeError(e))
            // PropagateBinary2 (for Plus)
            case err @ Left(_) => err
        }
        // TypeErrorPlus1
        case Right(_) => Left(DynamicTypeError(e))
        // PropagateBinary1 (for Plus)
        case err @ Left(_) => err
    }
    case _ => ???
}
```

```
defined function eval
```

An Exceeding (E) answer may use `flatMap` to more conveniently implement the Propagate rules, though it should recognize that those rules are indeed being implemented within the calls to `flatMap`.

Part V

Static Checking

24 Higher-Order Functions

Returning to programming principles, recall that in many languages like Scala, functions are first-class. What this means is that *functions are values* — they may be passed as arguments or returned as returned values from other functions. Functions that take function arguments are called *higher-order functions*.

24.1 Currying

Recall that we can write down function literals and bind them to variables:

```scala
(n: Int) => n + 1
((n: Int) => n + 1)(41)

val incr: Int => Int = { n => n + 1 }
incr(41)
```

```
res0_0: Int => Int = ammonite.$sess.cmd0$Helper$$Lambda$1811/0x0000000800a39840@edb2f86
res0_1: Int = 42
incr: Int => Int = ammonite.$sess.cmd0$Helper$$Lambda$1813/0x0000000800a3b040@3c5a87fa
res0_3: Int = 42
```

We have seen that with first-class functions (and lexical scoping), we do not need tuples or other data structures to have multi-parameter functions. In particular, a function that returns another function behaves like a multi-parameter function. This is called *currying*.

```scala
val plus: Int => Int => Int = { x => y => x + y }
plus(3)(4)
```

```
plus: Int => Int => Int = ammonite.$sess.cmd1$Helper$$Lambda$1925/0x0000000800a96040@7a746d0c
res1_1: Int = 7
```

Since currying is a common thing to do, Scala has some syntactic sugar for it:

```scala
def plus(x: Int)(y: Int): Int = x + y
plus(3)(4)
```

```
defined function plus
res2_1: Int = 7
```

One reason to use currying is to enable *partial application*. For example, we can define `incr` using `plus`:

```scala
val incr: Int => Int = plus(1)
incr(41)
```

```
incr: Int => Int = ammonite.$sess.cmd3$Helper$$Lambda$1942/0x0000000800a9b840@570dea5c
res3_1: Int = 42
```

Sometimes partial application is simply for defining new functions in terms of others in a compact manner. Other times, partial application enables some non-trivial partial computation.

This is a silly example to illustrate the latter, defining a function `addToFactorial` that computes the `n` ! and then returns a function to add some number to that:

```scala
def addToFactorial(n: Int): Int => Int = {
  def factorial(n: Int): Int = n match {
    case 0 => 1
    case _ => n * factorial(n - 1)
  }
  val nth = factorial(n)
  m => nth + m
}
```

```
defined function addToFactorial
```

We can compute 10! once and then reuse it with the function `tenFactorialPlus`:

```scala
val tenFactorialPlus = addToFactorial(10)
tenFactorialPlus(47)
tenFactorialPlus(59)
```

```
tenFactorialPlus: Int => Int = ammonite.$sess.cmd4$Helper$$Lambda$1970/0x0000000800ab2040@78
res5_1: Int = 3628847
res5_2: Int = 3628859
```

24.2 Collections and Callbacks

We have seen standard data types like lists, options, maps, and sets that are often called *collections*, as they are generic in the values they collect together (see Section 6.1).

What is common to collection libraries is that the client of the library must have some way to work with the elements managed by the collection. Because the client decides the element type, the library implements higher-order functions that take a *callback* function argument to tell the library "what to do with the elements." For example, we have already seen one higher-order function `foreach` in the Scala standard library that enables the client to perform a side-effect for each element of a list:

```
List(1, 2, 3).foreach(println)
```

```
1
2
3
```

Note that Scala standard library chooses to define `foreach` as a method on objects of type `List[A]`.

In the following, we describe some standard higher-order functions on collections. Our intent is to discuss the fundamental higher-order programming patterns. While the examples are drawn from the Scala standard library, the patterns reoccur in many other contexts and languages. We also do not intend to describe the application programming interface (API) exhaustively, see the API documentation for that or other sources for library-specific tutorials.

24.2.1 Map

Recall that we use lists directly by pattern matching and recursion. For example, we can define functions to increment or double each integer in a given `List[Int]` or to get the length of each string in a given `List[String]`:

```
def increment(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => (h + 1) :: increment(t)
}
increment(List(1, 2, 3))
```

```
defined function increment
res7_1: List[Int] = List(2, 3, 4)
```

```
def double(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => (h * 2) :: double(t)
}
double(List(1, 2, 3))
```

```
defined function double
res8_1: List[Int] = List(2, 4, 6)
```

```
def eachLength(l: List[String]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h.length :: eachLength(t)
}
eachLength(List("Neo", "Trinity", "Morpheus"))
```

```
defined function eachLength
res9_1: List[Int] = List(3, 7, 8)
```

We see that transformation pattern is very common: we want to *map* each element from the input list to the corresponding element in the output list. We can implement this pattern generically given a callback function argument `f: A => B` that tells us how to map an `A` to a `B`:

```
def map[A, B](l: List[A])(f: A => B): List[B] = l match {
  case Nil => Nil
  case h :: t => f(h) :: map(t)(f)
}
```

```
defined function map
```

And we can then define the `increment`, `double`, and `eachLength` as a clients of the `map` function:

```
def increment(l: List[Int]): List[Int] = map(l) { h => h + 1 }
increment(List(1, 2, 3))
```

```
defined function increment
res11_1: List[Int] = List(2, 3, 4)
```

```
def double(l: List[Int]): List[Int] = map(l) { h => h * 2 }
double(List(1, 2, 3))
```

```
defined function double
res12_1: List[Int] = List(2, 4, 6)
```

```
def eachLength(l: List[String]): List[Int] = map(l) { h => h.length }
eachLength(List("Neo", "Trinity", "Morpheus"))
```

```
defined function eachLength
res13_1: List[Int] = List(3, 7, 8)
```

We have abstracted all of the common boilerplate code into the definition of `map` and have just what makes `increment`, `double`, and `eachLength` differ as the callback argument.

As noted above, the Scala standard library chooses to define these higher-order functions as methods on the `List[A]` data type, so we use the built-in version of `map` as follows:

```
List(1, 2, 3).map(i => i * 3)
```

```
res14: List[Int] = List(3, 6, 9)
```

Note that it is idiomatic in Scala to use the binary operator form for `map`:

```
List(1, 2, 3) map { i => i * 3 }
List(1, 2, 3) map { _ * 3 }
```

```
res15_0: List[Int] = List(3, 6, 9)
res15_1: List[Int] = List(3, 6, 9)
```

The binary operator form yields chains reminiscent of Unix pipes:

```
List(1, 2, 3) map { i => i * 3 } map { i => i + 1 }
List(1, 2, 3) map { _ * 3 } map { _ + 1 }
```

```
res16_0: List[Int] = List(4, 7, 10)
res16_1: List[Int] = List(4, 7, 10)
```

The chain of method call form is what modern Java (and other object-oriented languages) libraries call *fluent interfaces*:

```scala
List(1, 2, 3)
  .map(i => i * 3)
  .map(i => i + 1)

List(1, 2, 3)
  .map(_ * 3)
  .map(_ + 1)
```

```
res17_0: List[Int] = List(4, 7, 10)
res17_1: List[Int] = List(4, 7, 10)
```

Comprehensions

Scala has a loop-like form called a *comprehension* that translates to a `map` call:

```scala
for (i <- List(1, 2, 3)) yield i * 3
List(1, 2, 3) map { i => i * 3 }
```

```
res18_0: List[Int] = List(3, 6, 9)
res18_1: List[Int] = List(3, 6, 9)
```

A *comprehension* draws from set-comprehensions in mathematics:

$$\{i \cdot 3 \mid i \in \{1, 2, 3\}\}$$

And Python has similar syntax for list-comprehensions:

Listing 24.1 Python

```python
[i * 3 for i in [1, 2, 3]]
```

Comprehensions with constraints in mathematics and Python are also common:

$$\{i \cdot 3 \mid i \in \{1, 2, 3\} \text{ s.t. } i \bmod 2 = 1\}$$

and is supported in Scala:

Listing 24.2 `Python`

```
[i * 3 for i in [1, 2, 3] if i % 2 == 1]
```

```
for (i <- List(1, 2, 3) if i % 2 == 1) yield i * 3
```

```
res19: List[Int] = List(3, 9)
```

A constraint corresponds to first applying a filter:

```
List(1, 2, 3) filter { i => i % 2 == 1 } map { i => i * 3 }
```

```
res20: List[Int] = List(3, 9)
```

Because filtering and then mapping is common, Scala implements an optimization to record the filter to apply during the map.

```
List(1, 2, 3) filter { i => i % 2 == 1 }
List(1, 2, 3) filter { i => i % 2 == 1 } map { i => i }

List(1, 2, 3) withFilter { i => i % 2 == 1 }
List(1, 2, 3) withFilter { i => i % 2 == 1 } map { i => i}
```

```
res21_0: List[Int] = List(1, 3)
res21_1: List[Int] = List(1, 3)
res21_2: collection.WithFilter[Int, List[_]] = scala.collection.IterableOps$WithFilter@5181b2
res21_3: List[Int] = List(1, 3)
```

The `for-if-yield` comprehension translates to a call of `withFilter` and then `map`:

```
for (i <- List(1, 2, 3) if i % 2 == 1) yield i * 3
List(1, 2, 3) withFilter { i => i % 2 == 1 } map { i => i}
```

```
res22_0: List[Int] = List(3, 9)
res22_1: List[Int] = List(1, 3)
```

Pattern Matching on the Formal Parameter

While using `map`, we often want to pattern match in the parameter of the callback. For example,

```
List(None, Some(3), Some(4), None) map { iopt => iopt match {
  case None => 0
  case Some(i) => i + 1
} }
```

```
res23: List[Int] = List(0, 4, 5, 0)
```

We can drop the the **`match`** part to get the same behavior:

```
List(None, Some(3), Some(4), None) map {
  case None => 0
  case Some(i) => i + 1
}
```

```
res24: List[Int] = List(0, 4, 5, 0)
```

In actuality, the version without **`match`** the Scala syntax for defining "partial functions," which is a more specific version of "functions."

24.2.2 FlatMap

A slight generalization of `map` and `filter` together is called `flatMap`. Compare and contrast the type and implementations of `map` and `flatMap`:

```
def map[A, B](l: List[A])(f: A => B): List[B] = l match {
  case Nil => Nil
  case h :: t => f(h) :: map(t)(f)
}

def flatMap[A, B](l: List[A])(f: A => List[B]): List[B] = l match {
  case Nil => Nil
  case h :: t => f(h) ::: flatMap(t)(f)
}
```

```
defined function map
defined function flatMap
```

A `flatMap` takes a callback function argument `f: A => List[B]`, allowing us to define, for example, `duplicate`:

```
def duplicate[A](l: List[A]) = l flatMap { a => List(a, a) }
duplicate(List(1, 2, 3))
```

```
defined function duplicate
res26_1: List[Int] = List(1, 1, 2, 2, 3, 3)
```

The `flatMap` method takes its name from being a combination of `map` and `flatten`:

```
val mapped = List(1, 2, 3) map { a => List(a, a) }
val flattened = mapped.flatten
```

```
mapped: List[List[Int]] = List(List(1, 1), List(2, 2), List(3, 3))
flattened: List[Int] = List(1, 1, 2, 2, 3, 3)
```

While a direct implementation of `map` and `filter` is more efficient, we can see that `flatMap` is a generalization by defining `map` and `filter` using `flatMap`:

Exercise 24.1. Define `map` in terms of `flatMap`.

```
def map[A, B](l: List[A])(f: A => B): List[B] = ???
```

```
defined function map
```

Exercise 24.2. Define `filter` in terms of `flatMap`.

```
def filter[A](l: List[A])(f: A => Boolean): List[A] = ???
```

```
defined function filter
```

24.2.3 FoldRight

The `map` and `flatMap` offer transformations that stay within in the `List` type constructor. Let us look at examples `addList` and `multList` that summarize lists defined by direct recursion:

```scala
def addList(l: List[Int]): Int = l match {
  case Nil => 0
  case h :: t => h + addList(t)
}
addList(List(1, 2, 3, 4))
```

```
defined function addList
res30_1: Int = 10
```

```scala
def multList(l: List[Int]): Int = l match {
  case Nil => 1
  case h :: t => h * multList(t)
}
multList(List(1, 2, 3, 4))
```

```
defined function multList
res31_1: Int = 24
```

We recognize this summarization pattern: we use a binary operator to *fold* the recursively *accumulation* with the current element:

```scala
def foldRight[A, B](l: List[A])(z: B)(bop: (A, B) => B): B = l match {
  case Nil => z
  case h :: t => bop(h, foldRight(t)(z)(bop))
}
```

```
defined function foldRight
```

And we can then define the `addList` and `multList` as a clients of the `foldRight` function:

```scala
def addList(l: List[Int]): Int = foldRight(l)(0) { (h, acc) => h + acc }
addList(List(1, 2, 3, 4))

def multList(l: List[Int]): Int = foldRight(l)(1) { (h, acc) => h * acc }
multList(List(1, 2, 3, 4))
```

```
defined function addList
res33_1: Int = 10
defined function multList
res33_3: Int = 24
```

Like `map`, `foldRight` is defined as a method on `List[A]` in Scala:

```
List(1, 2, 3, 4).foldRight(0) { (h, acc) => h + acc }
List(1, 2, 3, 4).foldRight(1) { (h, acc) => h * acc }
```

```
res34_0: Int = 10
res34_1: Int = 24
```

Catamorphism

Take a closer look at the `foldRight` implementation:

```
def foldRight[A, B](l: List[A])(z: B)(bop: (A, B) => B): B = l match {
  case Nil => z
  case h :: t => bop(h, foldRight(t)(z)(bop))
}
```

```
defined function foldRight
```

and we see that it abstracts exactly structural recursion over the inductive data type `List[A]` where the `z` parameter corresponds to `Nil` constructor and the `bop` parameter to the `::` constructor. This pattern called a *catamorphism* can be translated into any inductive data type that abstracts the structural recursion with a parameter for each constructor. We say that `foldRight` is the catamorphism for `List`.

It is good practice to structural recursive functions using `foldRight`:

Exercise 24.3. Define `map` in terms of `foldRight`

```
def map[A,B](l: List[A])(f: A => B): List[B] = ???
```

```
defined function map
```

Exercise 24.4. Define `append: (List[A], List[A]) => List[A]` that appends together two lists into one list (i.e., returns `l1` follows by `l2`) in terms of `foldRight`:

```
def append[A](l1: List[A], l2: List[A]): List[A] = ???
```

```
defined function append
```

24.2.4 Other Folds and Reduce

With lists, we have another common pattern: tail-recursive iteration. This pattern is abstracted with the `foldLeft` function:

```
def foldLeft[A, B](l: List[A])(z: B)(bop: (B, A) => B): B = {
  def loop(acc: B, l: List[A]): B = l match {
    case Nil => acc
    case h :: t => loop(bop(acc, h), t)
  }
  loop(z, l)
}
```

```
defined function foldLeft
```

Because multiplication is associative, we can also use the tail-recursive `foldLeft` to multiply the elements of an integer list:

```
List(1, 2, 3, 4).foldLeft(1) { (acc, h) => acc * h }
```

```
res39: Int = 24
```

The mnemonic for `foldRight` versus `foldLeft` is that `foldRight` accumulates from the right of the list, while `foldLeft` accumulates from the left.

A good exercise is to write tail-recursive iteration lists functions using `foldLeft`.

Exercise 24.5. Define `reverse` of a list in terms of `foldLeft`.

```
def reverse[A](l: List[A]): List[A] = ???
```

```
defined function reverse
```

Reduce

When the order does not matter because the binary operator associative, using `fold` method allows the library to do whatever is most efficient:

```
List(1, 2, 3, 4).fold(1)(_ * _)
```

```
res41: Int = 24
```

A further special case of using an associative operator binary on a non-empty list is `reduce`:

```
List(1, 2, 3, 4).reduce(_ * _)
```

```
res42: Int = 24
```

that picks an element as the starting accumulator.

24.3 Abstract Data Types

We have seen that `Map` and `Set` data types are unlike `List` are *abstract data types* where we cannot get at the underlying representation. They prevent the client from direct access to underlying balanced search tree representation to be able to maintain the balance and search invariants, allowing for efficient key-based lookup.

At the same time, higher-order functions enables them to present the same collection view as lists with `map`, `flatMap`, `foldRight`, and `foldLeft`:

```
val m = Map(2 -> List("two", "dos", "  "), 10 -> List("ten", "diez", "  "))
```

```
m: Map[Int, List[String]] = Map(
  2 -> List("two", "dos", "\u4e8c"),
  10 -> List("ten", "diez", "\u5341")
)
```

```
m map { case k -> words => k -> words.head }
```

```
res44: Map[Int, String] = Map(2 -> "two", 10 -> "ten")
```

```
m.foldRight(Nil: List[String]) {
  case (_ -> words, acc) => words.head :: acc
}
```

```
res45: List[String] = List("two", "ten")
```

Parallel and Distributed

This decoupling of the concrete representation from the higher-order accessor view is incredibly powerful. For example, the same client code using `map` and `reduce` on sequential collections can be re-used by loading a parallel collections library:

> **ℹ Scala Parallel Collections Library**
>
> Run the following cell to load the Scala Parallel Collections library.
>
> **Listing 24.3** `scala.collection.parallel.CollectionConverters._`
>
> ```
> import $ivy.`org.scala-lang.modules::scala-parallel-collections:1.0.4`, scala.collection.pa
> ```
>
> ```
> import $ivy.$                                                        , scala.collection.pa
> ```

```
val par0to9999 = (0 to 9999).toList.par
val sum = par0to9999.map(_ + 1).reduce(_ + _)
assert(sum == 50005000)
```

```
par0to9999: collection.parallel.immutable.ParSeq[Int] = ParVector(0, 1, 2, 3, 4, 5, 6, 7, 8,
sum: Int = 50005000
```

This same idea underlies big-data applications where the library takes care of scheduling distributed jobs with client code that also works in the small locally in memory.

25 Exercise: Higher-Order Functions

Learning Goals

The primary learning goal of this exercise is to get experience programming with higher-order functions.

Instructions

This assignment asks you to write Scala code. There are restrictions associated with how you can solve these problems. Please pay careful heed to those. If you are unsure, ask the course staff.

Note that `???` indicates that there is a missing function or code fragment that needs to be filled in. Make sure that you remove the `???` and replace it with the answer.

Use the test cases provided to test your implementations. You are also encouraged to write your own test cases to help debug your work. However, please delete any extra cells you may have created lest they break an autograder.

Imports

```
import $ivy.$                                        , org.scalatest._, events._, flatspec._
```

```
defined function report
defined function assertPassed
defined function passed
defined function test
```

Listing 25.1 `org.scalatest._`

```
// Run this cell FIRST before testing.
import $ivy.`org.scalatest::scalatest:3.2.19`, org.scalatest._, events._, flatspec._
def report(suite: Suite): Unit = suite.execute(stats = true)
def assertPassed(suite: Suite): Unit =
  suite.run(None, Args(new Reporter {
    def apply(e: Event) = e match {
      case e @ (_: TestFailed) => assert(false, s"${e.message} (${e.testName})")
      case _ => ()
    }
  }))
def passed(points: Int): Unit = {
  require(points >=0)
  if (points == 1) println("***   Tests Passed (1 point) ***")
  else println(s"***   Tests Passed ($points points) ***")
}
def test(suite: Suite, points: Int): Unit = {
  report(suite)
  assertPassed(suite)
  passed(points)
}
```

25.1 Collections

When working with and organizing data, we primarily use collections from Scala's standard library. One of the most fundamental operations that one needs to perform with a collection is to iterate over the elements. Like many other languages with first-class functions (e.g., Python, ML), the Scala library provides various iteration operations via *higher-order functions*. Higher-order functions are functions that take functions as parameters. The function parameters are often called *callbacks*, and for collections, they typically specify what the library client wants to do for each element. We have seen examples of these functions in class. In the following questions, we practice both writing such higher-order functions in a library and using them as a client.

25.1.1 Lists

First, we will implement functions that eliminate consecutive duplicates of list elements. If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed. For example, the following `List(1, 2, 2, 3, 3, 3)` should be converted to `List(1,2,3)`.

Exercise 25.1 (5 points)**.** Write a function `compressRec` that implements this behavior. Implement the function by direct recursion (e.g., pattern match on `l` and call `compressRec` recursively). Do not call any `List` library methods.

Edit this cell:

```
def compressRec[A](l: List[A]): List[A] = l match {
  case Nil | _ :: Nil =>
    ???
  case h1 :: (t1 @ (h2 :: _)) =>
    ???
}
```

```
defined function compressRec
```

Notes

- This exercise is from Ninety-Nine Scala Problems. Some sample solutions are given there, which you are welcome to view. However, we *strongly* recommend you attempt this exercise before looking there. The purpose of the exercise is to get some practice for the later part of this homework. Note that the solutions there do not satisfy the requirements here, as they use library functions. If at some point you feel like you need more practice with collections, this site is a good resource.

Tests

Exercise 25.2 (5 points)**.** Write a different function `compressFold` that re-implements the behavior of `compressRec` using the `foldRight` method from the `List` library. The call to `foldRight` has been provided for you. Do not call `compressFold` recursively or any other `List` library methods.

Edit this cell:

```
def compressFold[A](l: List[A]): List[A] = l.foldRight(Nil: List[A]){ (h, acc) =>
  ???
}
```

```
defined function compressFold
```

Tests

Exercise 25.3 (5 points)**.** Explain in 1–2 sentences the similarities and differences between your two implementations: `compressRec` and `compressFold`.

Edit this cell:

???

Exercise 25.4 (5 points)**.** Implement a higher-order recursive function `mapFirst` that finds the first element of `l: List[A]` where `f: A => Option[A]` applied to it returns `Some(a)` for some value `a`. The function should replace that element with `a` and leave `l` the same everywhere else. For example,

```
mapFirst(List(1,2,-3,4,-5)) { i => if (i < 0) Some(-i) else None }
```

should result in `List[Int] = List(1, 2, 3, 4, -5)`.

Edit this cell:

Tests

Exercise 25.5 (5 points)**.** Write a function `composeMap` that sequentially applies a list of functions of type `A => A` to all the elements of a `List[A]`. For example, if we have a list of functions `List(f1, f2, f3)`, and a list `List(a, b)`, we want to output `List(f3(f2(f1(a))), f3(f2(f1(b))))`.

Edit this cell:

```
def composeMap[A](functions: List[A => A])(l: List[A]): List[A] =
  ???
```

```
defined function composeMap
```

Tests

25.1.2 Maps

Recall the `Map[A, B]` data structure from class. Also, recall the higher-order function `map`. To avoid confusion we will use the upper case `Map` to refer to the type, and the lowercase `map` to refer to the higher-order function.

Exercise 25.6 (5 points)**.** Implement a function `mapValues` that takes a `Map[A,B]` and a callback function `f: B => C`, that applies `f` to all the values in the `Map`. Your function should use the Scala collections `map` method. Do not use the standard library method `mapValues` on `Map` (however, note that the behavior of your function should be exactly the same as the `mapValues` standard library method).

Edit this cell:

Tests

Exercise 25.7 (5 points)**.** As we mentioned above, we just reimplemented `mapValues` using the `map` method from the standard library. Earlier, we implemented higher-order functions on `List`s recursively. Could we have implemented `mapValues` on `Map`s recursively? If yes, give an example implementation. If no, explain why we cannot, and what makes `Map` different from `List` in this case.

Edit this cell:

???

25.1.3 Trees

Recall the binary tree data type:

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

```
defined trait Tree
defined object Empty
defined class Node
```

Exercise 25.8 (10 points). Implement a higher-order function `foldLeft` that performs an in-order traversal of the input `t: Tree`, calling the callback `f` starting with `z` to accumulate a result. For example, suppose the in-order traversal of the input tree `t` yields the sequence of data values: $d_1, d_2, \ldots, d_n$. Then, `foldLeft(t)(z)(f)` yields $f(f(\ldots(f(f(z, d_1), d_2))\ldots d_{n-1}), d_n)$

Edit this cell:

```
def foldLeft[A](t: Tree)(z: A)(f: (A, Int) => A): A = {
  def loop(acc: A, t: Tree): A = t match {
    case Empty =>
      ???
    case Node(l, d, r) =>
      ???
  }
  ???
}
```

```
defined function foldLeft
```

We have provided a test client `sum` that computes the sum of all of the data values in the tree using your `foldLeft` method, along with some helper functions to build trees more easily. Feel free to use them to write more test cases for your code.

```
// An example use of foldLeft
def sum(t: Tree): Int = foldLeft(t)(0){ (acc, d) => acc + d }

// In-order insertion into a binary search tree
def insert(t: Tree, n: Int): Tree = t match {
  case Empty => Node(Empty, n, Empty)
  case Node(l, d, r) =>
    if (n < d) Node(insert(l, n), d, r) else Node(l, d, insert(r, n))
}

// Create a tree from a list. An example use of the List.foldLeft method.
def treeFromList(l: List[Int]): Tree =
  l.foldLeft(Empty: Tree){ (acc, i) => insert(acc, i) }
```

```
defined function sum
defined function insert
defined function treeFromList
```

Tests

Exercise 25.9 (10 points)**.** Using your `foldLeft` function, write a client function `strictlyOrdered` that checks if the data values of an in-order traversal of `t` are in strictly ascending order. For example, suppose the in-order traversal of the input tree `t` yields the sequence of data values: $d_1, d_2, \ldots, d_n$, the `strictlyOrdered` should return `true` iff $d_1 < d_2 < \cdots < d_n$.

Edit this cell:

```
def strictlyOrdered(t: Tree): Boolean = {
  val (b, _) = foldLeft(t)((true, None: Option[Int])) {
    ???
  }
  b
}
```

```
defined function strictlyOrdered
```

Tests

Now, we will write a higher-order function that uses our recent higher-order functions as a client.

Exercise 25.10 (5 points)**.** Implement a function `foldLeftTrees` that take as input a `lt: List[Tree]`, and applies a callback `f` to all of their nodes in-order (of both the `List` and the nested `Trees`) starting with initial value `z` to accumulate a result.

For example, calling `foldLeftTrees` on

```
val lt =
  List(
    Node(Node(Empty, 1, Empty), 2, Node(Empty, 3, Empty)),
    Node(Node(Empty, 4, Empty), 5, Node(Empty, 6, Empty))
  )
```

```
lt: List[Node] = List(
  Node(
    l = Node(l = Empty, d = 1, r = Empty),
    d = 2,
    r = Node(l = Empty, d = 3, r = Empty)
```

```
  ),
  Node(
    l = Node(l = Empty, d = 4, r = Empty),
    d = 5,
    r = Node(l = Empty, d = 6, r = Empty)
  )
)
```

and a function that sums integers should return `21`.

Use only `foldLeft` on `Tree` that you have defined above and `foldLeft` on `List[Tree]` provided in the Scala standard library.

Edit this cell:

```
def foldLeftTrees[B](lt: List[Tree])(z: B)(f: (B, Int) => B): B =
  ???
```

```
defined function foldLeftTrees
```

Tests

25.2 flatMap

Exercise 25.11 (5 points)**.** Recall the `flatMap` function from class on `List`. Write a new function `flatMapNoRec` that implements the same behavior without direct recursion; instead, use `foldRight` and `:::`.

Edit this cell:

Tests

Now, we will try to use `flatMap` to do something useful.

Exercise 25.12 (5 points)**.** Write a function `getAllWords` that takes in a `List[String]` of sentences and returns a `List[String]` of all the words in the sentences. For simplicity, our sentences will have no punctuation, and all words will be separated by a single space. For example, given the input `List("I love 3155", "Anish is the best TA")`, the `getAllWords` function should return `List("I", "love", "3155", "Anish", "is", "the", "best", "TA")`.

Use the String method `split` to separate a sentence into its component words:

```scala
"Functions are values".split(" ").toList
```

```
res22: List[String] = List("Functions", "are", "values")
```

Edit this cell:

Tests

26 Static Type Checking

When we considered just a single type (e.g., numbers) in Section 21.2, we defined a one-step reduction relation such that for any closed expression e: either e $\textsf{value}$ (i.e., e is a value) or $e \longrightarrow e'$ for some e' (i.e., e can take a step to some e'). This property is very nice, but as soon as we added another type of value, things got messy. We considered different possible designs:

1. We defined conversions between different types of values (e.g., coercing values v to numbers n with the judgment form $v \rightsquigarrow n$).
2. We defined dynamic type checking with the judgment form $e \longrightarrow r$ where step-result $r ::= \textsf{typeerror}\, e \mid e'$ is either a type-error result or a one-step reduced expression.

While conversions preserve the nice property that every expression is either a value or has a next step, a drawback with conversions is that some types of values simply do not have sensible conversions. For example, how should the number `3` convert to a function value?

Dynamic type checking changes the judgment form $e \longrightarrow r$ so that the next step could be to a $\textsf{typeerror}$, but that adds complexity for identifying and propagating errors.

Either choice comes at a cost in complexity.

26.1 JavaScripty: Numbers and Functions

26.1.1 Syntax

Let us consider JavaScripty just with number literals, anonymous function literals, and function call expressions:

$$
\begin{array}{rrcl}
\text{values} & v & ::= & n \mid \texttt{(}x\texttt{)} \Rightarrow e_1 \\
\text{expressions} & e & ::= & n \mid \texttt{(}x\texttt{)} \Rightarrow e_1 \mid x \mid e_1\texttt{(}e_2\texttt{)} \\
\text{variables} & x & &
\end{array}
$$

Figure 26.1: Syntax of JavaScripty with number literals, function literals, and function call expressions.

```scala
trait Expr                                       // e
case class N(n: Double) extends Expr             // e ::= n
case class Fun(x: String, e1: Expr) extends Expr // e ::= (x) => e1
case class Var(x: String) extends Expr           // e ::= x
case class Call(e1: Expr, e2: Expr) extends Expr // e ::= e1(e2)

def isValue(e: Expr): Boolean = e match {
  case N(_) | Fun(_, _) => true
  case _ => false
}
```

```
defined trait Expr
defined class N
defined class Fun
defined class Var
defined class Call
defined function isValue
```

26.1.2 Small-Step Operational Semantics

The small-step operational semantics just consists of reducing function call expressions:

$$\boxed{e \longrightarrow e'} \qquad \frac{\textsc{DoCall}}{((x) \Rightarrow e_1)(v_2) \longrightarrow [v_2/x]e_1} \qquad \frac{\overset{\textsc{SearchCall1}}{e_1 \longrightarrow e_1'}}{e_1(e_2) \longrightarrow e_1'(e_2)} \qquad \frac{\overset{\textsc{SearchCall2}}{e_2 \longrightarrow e_2'}}{v_1(e_2) \longrightarrow v_1(e_2')}$$

Figure 26.2: Small-step operational semantics with number literals, function literals, and function call expressions.

```scala
def subst(v: Expr, x: String, e: Expr) = {
  def subst(e: Expr): Expr = e match {
    case N(_) => e
    case Fun(y, e1) => if (x == y) e else Fun(y, subst(e1))
    case Var(y) => if (x == y) v else e
    case Call(e1, e2) => Call(subst(e1), subst(e2))
  }
  subst(e)
}
```

```scala
def step(e: Expr): Expr = e match {
  // DoCall
  case Call(Fun(x, e1), v2) if isValue(v2) => subst(v2, x, e1)
  // SearchCall2
  case Call(v1, e2) if isValue(v1) => Call(v1, step(e2))
  // SearchCall1
  case Call(e1, e2) => Call(step(e1), e2)
}
```

```
defined function subst
defined function step
```

26.2 Getting Stuck

In Figure 26.2, we restate the small-step operational semantics rules for reducing function calls. Observe in the DoCall rule that a reduction 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 **?@sec-call-by-name**.

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

$$e_{\text{illtyped}} : \quad \texttt{3(4)}$$

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

In implementation, we get undefined behavior (e.g., crashing with a `MatchError`):

```scala
val e_illtyped = Call(N(3), N(4))
```

```
e_illtyped: Call = Call(e1 = N(n = 3.0), e2 = N(n = 4.0))
```

```scala
step(e_stuck)
```

26.3 Dynamic Typing

As we saw in our introduction to dynamic typing (Section 21.3), another formalization and implementation choice would be to make such ill-typed expressions step to an error token with an updated the judgment form $e \longrightarrow r$:

$$\text{step-results} \quad r \quad ::= \quad \mathsf{typeerror}\, e \mid e'$$

$$\boxed{e \longrightarrow r} \qquad \frac{v_1 \neq x^?(y) \Rightarrow e_1}{v_1\,(e_2) \longrightarrow \mathsf{typeerror}(v_1\,(e_2))}\;\textsc{TypeErrorCall} \qquad \frac{e_1 \longrightarrow \mathsf{typeerror}\, e}{e_1\,(e_2) \longrightarrow \mathsf{typeerror}\, e}\;\textsc{PropagateCall1}$$

$$\frac{e_2 \longrightarrow \mathsf{typeerror}\, e}{v_1\,(e_2) \longrightarrow \mathsf{typeerror}\, e}\;\textsc{PropagateCall2}$$

Figure 26.3: Extending the small-step semantics from Figure 26.2 with dynamic type errors.

```
case class DynamicTypeError(e: Expr)
```

```
defined class DynamicTypeError
```

```
def step(e: Expr): Either[DynamicTypeError, Expr] = e match {
  case Call(v1, v2) if isValue(v1) && isValue(v2) => v1 match {
    // DoCall
    case Fun(x, e1) => Right( subst(v2, x, e1) )
    // TypeErrorCall
    case _ => Left( DynamicTypeError(e) )
  }
  // SearchCall2 and PropagateCall2
  case Call(v1, e2) if isValue(v1) => step(e2) map { e2 => Call(v1, e2) }
  // SearchCall1 and PropagateCall1
  case Call(e1, e2) => step(e1) map { e1 => Call(e1, e2) }
}
```

```
defined function step
```

An ill-typed function call now step to typeerror with rule TypeErrorCall. We also need to extend rules for evaluating other all other expression forms that propagate the typeerror token if one is encountered in searching for a redex. We show PropagateCall1 and PropagateCall2, which are two such rules, that correspond to SearchCall1 and SearchCall2, 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

$$e_{\text{open}}: \quad \texttt{f(4)}$$

should get stuck. Since our one-step evaluation relation is intended for closed expressions, we should view this as an internal error of the interpreter implementation rather than an error in the input program.

```
step(e_illtyped)
```

```
res6: Either[DynamicTypeError, Expr] = Left(
  value = DynamicTypeError(e = Call(e1 = N(n = 3.0), e2 = N(n = 4.0)))
)
```

```
val e_open = Call(Var("f"), N(4))
```

```
e_open: Call = Call(e1 = Var(x = "f"), e2 = N(n = 4.0))
```

```
step(e_open)
```

26.4 Static Typing

We saw how "bad" expressions, such as,

$$e_{\text{illtyped}}: \quad \texttt{3(4)}$$

are erroneous in that they "get stuck" according to our simpler small-step operational semantics or result in a typeerror according to our semantics with dynamic typing. This expression is "bad" because a call expression $e_1(e_2)$ is 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 characterizes 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 (written with judgment form $e : \tau$), we mean that if e evaluates to a value v, then that value v should be of type τ. In this way, a type system predicts some property about how an expression evaluates at run-time.

What is interesting is that we can design a type system to rule out ill-typed expressions before evaluation. If we only permit well-typed expressions to run, then we can use our simpler small-step operational semantics and know that we never get stuck getting to a value. This observation leads to the well-known quote:

> Well-typed programs can't go wrong. — Robin Milner [3]

Correspondingly, we can use the simpler interpreter implementation and know that a crash corresponds to an internal error of the implementation rather than an error in the input program.

26.5 TypeScripty: Numbers and Functions

Let us consider a statically-typed language that we affectionately call TypeScripty. In fact, like with JavaScripty and JavaScript, we make TypeScripty a subset of TypeScript whenever possible or indicate when we make a choice to make them differ.

26.5.1 Syntax

Let us consider the abstract syntax of TypeScripty with numbers and functions:

$$\begin{array}{rrcl} \text{types} & \tau, t & ::= & \texttt{number} \mid \texttt{(}x\texttt{:}\,\tau\texttt{)} \Rightarrow \tau' \\ \text{values} & v & ::= & n \mid \texttt{(}x\texttt{:}\,\tau\texttt{)} \Rightarrow e_1 \\ \text{expressions} & e & ::= & n \mid \texttt{(}x\texttt{:}\,\tau\texttt{)} \Rightarrow e_1 \mid x \mid e_1\texttt{(}e_2\texttt{)} \end{array}$$

Figure 26.4: Syntax of TypeScripty with number literals, function literals, and function call expressions.

In Figure 26.4, we show a language of types τ that includes base types for numbers `number` and a constructed type for function values

$$\texttt{(}x\texttt{:}\,\tau\texttt{)} \Rightarrow \tau'$$

A function type $\texttt{(}x\texttt{:}\,\tau\texttt{)} \Rightarrow \tau'$ classifies function values that has a parameter x of type τ to produce a return value of type τ'. Compared with JavaScripty syntax (Figure 26.1), our expression language e has been modified just slightly to add type annotations to function parameters. Take note of the parallel between values v and types τ—specifically, each type of value has a form in the type language τ.

We can represent the abstract syntax of TypeScripty with functions in Scala as follows:

```
trait Typ                                                //
case object TNumber extends Typ                          //   ::= number
case class TFun(xt: (String, Typ), tret: Typ) extends Typ //   ::= (x:  ) => '

trait Expr                                              // e
case class N(n: Double) extends Expr                    // e ::= n
case class Fun(xt: (String, Typ), e1: Expr) extends Expr // e ::= (x:  ) => e1
case class Var(x: String) extends Expr                  // e ::= x
case class Call(e1: Expr, e2: Expr) extends Expr        // e ::= e1(e2)

def isValue(e: Expr): Boolean = e match {
  case N(_) | Fun(_, _) => true
  case _ => false
}
```

```
defined trait Typ
defined object TNumber
defined class TFun
defined trait Expr
defined class N
defined class Fun
defined class Var
defined class Call
defined function isValue
```

For the `Expr` type, the only change compared with the abstract syntax representation of JavaScripty (Section 26.1.1) is this additional `Typ` parameter in the `Fun` constructor.

26.5.2 Small-Step Operational Semantics

Let us consider the small-step operational semantics of TypeScripty with functions:

The small-step operational semantics of TypeScripty here with functions are the same as JavaScripty with functions (Figure 26.2).

$$\boxed{e \longrightarrow e'} \qquad \frac{\textsc{DoCall}}{((x\colon \tau) \Rightarrow e_1)(v_2) \longrightarrow [v_2/x]e_1} \qquad \frac{\overset{\textsc{SearchCall1}}{e_1 \longrightarrow e_1'}}{e_1(e_2) \longrightarrow e_1'(e_2)} \qquad \frac{\overset{\textsc{SearchCall2}}{e_2 \longrightarrow e_2'}}{v_1(e_2) \longrightarrow v_1(e_2')}$$

Figure 26.5: Small-step operational semantics of TypeScripty with number literals, function literals, and function call expressions.

```
def subst(v: Expr, x: String, e: Expr) = {
  def subst(e: Expr): Expr = e match {
    case N(_) => e
    case Fun(yt @ (y, _), e1) => if (x == y) e else Fun(yt, subst(e1))
    case Var(y) => if (x == y) v else e
    case Call(e1, e2) => Call(subst(e1), subst(e2))
  }
  subst(e)
}

def step(e: Expr): Expr = {
  require(!isValue(e))
  e match {
    // DoCall
    case Call(Fun((y, _), e1), v2) => subst(v2, y, e1)
    // SearchCall2
    case Call(v1, e2) if isValue(v1) => Call(v1, step(e2))
    // SearchCall1
    case Call(e1, e2) => Call(step(e1), e2)
  }
}
```

```
defined function subst
defined function step
```

26.6 Typing Judgment

We want to judge when an expression e will evaluate to a value v of a particular type τ. We have already seen that this judgment form

$$e : \tau$$

that says, "Expression e has type τ." What we mean by "has type" is that the expression e evaluates to a value of the corresponding type τ.

Like with evaluation, a typing judgment form is defined by a set of *typing rules* that is the first step towards defining a *type checking* algorithm. Hence, typing is often called the *static semantics* of a language, while evaluation is the *dynamic semantics*.

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 expressions.

Recall from our earlier discussion on binding that to give a semantics and hence a type to an expression with free variable, we need an environment. For example, consider the expression

```
f(4)
```

Is this expression well-typed? It depends. If in the environment, the variable `f` is stated to have type $(x\colon \texttt{number}) \Rightarrow \tau'$, 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:

$$\text{type environments} \quad \Gamma, \mathit{tenv} \quad ::= \quad \cdot \mid \Gamma, x : \tau$$

```scala
type TEnv = Map[String, Typ] // Γ
```

```
defined type TEnv
```

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 $E \vdash e \Downarrow v$ from Section 18.5. This parallel is more than a mere 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 26.6, we define typing of TypeScripty with number literals, function literals, and function call expressions. The first two rules TypeNumber and TypeFunction describe the types of values. The type of the number literal n is `number` as expected. The TypeFunction rule is more interesting:

$$\boxed{\Gamma \vdash e : \tau} \qquad \frac{\textsc{TypeNumber}}{n : \texttt{number}} \qquad \textsc{TypeFunction}\ \frac{\Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{(}x\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ e_1 : \texttt{(}y\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ \tau'} \qquad \textsc{TypeVar}\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{TypeCall}\ \frac{\Gamma \vdash e_1 : \texttt{(}x\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau'}$$

Figure 26.6: Typing of TypeScripty with number literals, function literals, and function call expressions.

$$\textsc{TypeFunction}\ \frac{\Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{(}x\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ e_1 : \texttt{(}y\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ \tau'}$$

A function value has a function type $\texttt{(}y\texttt{:}\ \tau\texttt{)}\ \texttt{=>}\ \tau'$ (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 $\Gamma, x : \tau$. In TypeScript, the parameter name y in the function type $\texttt{(}y\texttt{:}\ \ldots\texttt{)}\ \texttt{=>}\ \ldots$ is essentially inconsequential and does not need to match the parameter name x in the function literal $\texttt{(}x\texttt{:}\ \ldots\texttt{)}\ \texttt{=>}\ \ldots$.

For a closed expression e, we write $e : \tau$ for $\cdot \vdash e : \tau$, that is, well-typed with an empty type environment:

$$\frac{\cdot \vdash e : \tau}{e : \tau}$$

We can translate the rules defining the typing judgment forms $e : \tau$ and $\Gamma \vdash e : \tau$ into a Scala implementation as follows:

```
def hastype(e: Expr): Typ = {
  def hastype(tenv: TEnv, e: Expr): Typ = e match {
    // TypeNumber
    case N(_) => TNumber
    // TypeFunction
    case Fun((x,t), e1) => TFun((x,t), hastype(tenv + (x -> t), e1))
    // TypeVar
    case Var(x) => tenv(x)
    // TypeCall
    case Call(e1, e2) => hastype(tenv, e1) match {
      case TFun((_,t), tret) if t == hastype(tenv, e2) => tret
    }
```

```
  }
  hastype(Map.empty, e)
}
```

```
defined function hastype
```

To test, let us consider the ill-typed expression from above, as well as a well-typed one:

$$e_{\text{welltyped}}: \quad \texttt{((i: number) => i)(4)} \qquad\qquad e_{\text{illtyped}}: \quad \texttt{3(4)}$$

```
val e_welltyped = Call(Fun(("i", TNumber), Var("i")), N(4))
val e_illtyped = Call(N(3), N(4))
```

```
e_welltyped: Call = Call(
  e1 = Fun(xt = ("i", TNumber), e1 = Var(x = "i")),
  e2 = N(n = 4.0)
)
e_illtyped: Call = Call(e1 = N(n = 3.0), e2 = N(n = 4.0))
```

```
hastype(e_welltyped)
```

```
res14: Typ = TNumber
```

That is, we infer that in the empty environment, `e_welltyped` has type `TNumber`, corresponding to the judgment

$$\texttt{((i: number) => i)(4)} : \texttt{number}$$

holding.

In the case of the ill-typed term, this implementation of `hastype` simply crashes with a `MatchError`:

```
hastype(e_illtyped)
```

To give a better error message to the programmer, we may want to identify the unexpected, bad type `tbad` for a particular sub-expression `esub` of input expression `e`:

```
case class StaticTypeError(tbad: Typ, esub: Expr, e: Expr) extends Exception {
  override def toString: String = s"invalid type ${tbad} for sub-expression ${esub} in ${e}"
}

def hastype(e: Expr): Typ = {
  def hastype(tenv: TEnv, e: Expr): Typ = e match {
    // TypeNumber
    case N(_) => TNumber
    // TypeFunction
    case Fun((x,t), e1) => TFun((x,t), hastype(tenv + (x -> t), e1))
    // TypeVar
    case Var(x) => tenv(x)
    // TypeCall
    case Call(e1, e2) => hastype(tenv, e1) match {
      case TFun((_,t), tret) if t == hastype(tenv, e2) => tret
      case tbad => throw StaticTypeError(tbad, e1, e)
    }
  }
  hastype(Map.empty, e)
}
```

```
defined class StaticTypeError
defined function hastype
```

```
hastype(e_illtyped)
```

26.7 Type Soundness

Our goal has been to design a type system such that whenever we say an expression e is well-typed (i.e., has a type τ for some type τ), then it can never get stuck in a step (i.e., $e \longrightarrow e'$ for some reduced expression e') in reducing to a value. This proposition is a meta-property relating our typing judgment form $e : \tau$ and our reduction-step judgment form $e \longrightarrow e'$ that we break up into two parts, *progress* and *preservation*:

Proposition 26.1 (Progress)**.** *If $e : \tau$, then $e \longrightarrow e'$ for some expression e'.*

Proposition 26.2 (Preservation)**.** *If $e \longrightarrow e'$ and $e : \tau$, then $e' : \tau$.*

If we can prove these propositions, then we say that our type system is *sound*, that is, it correctly classifies expressions that on evaluation, will never get stuck or result in an error. If our type system correctly claims that an expression e is well-typed, then every step in iteratively reducing e to a value will remain well-typed according to Progress and Preservation.

While the typically production scenario is to apply type checking statically before evaluating at full speed without checking, we can operationize this property to test our interpreter implementation. Let us instrument the reduction to a value judgment form $e \hookrightarrow_\tau v$ to say, "Expression e reduces to a value v using some number of steps while checking the preservation of type τ at each step.":

$$\boxed{e \hookrightarrow_\tau v} \qquad \frac{e\ \mathsf{value}}{e \hookrightarrow_\tau e}\ \textsc{ReducesValue} \qquad \frac{e \longrightarrow e' \qquad \cdot \vdash e' : \tau \qquad e' \hookrightarrow_\tau e''}{e \hookrightarrow_\tau e''}\ \textsc{ReducesProgressAndPreservation}$$

We define a generic `iterate` function and a `iterateStepPAP` function that implements the $e \hookrightarrow_\tau v$ judgment form:

```
def iterate[A](acc: A)(step: A => Option[A]): A = {
  def loop(acc: A): A = step(acc) match {
    case None => acc
    case Some(acc) => loop(acc)
  }
  loop(acc)
}

def iterateStepPAP(e: Expr): Expr = {
  // Check e is well-typed in that it doesn't throw StaticTypeError.
  val ty = hastype(e)
  // Iterate step while checking type preservation.
  iterate(e) {
    // ReducesValue
    case v if isValue(v) => None
    // ReducesProgressAndPreservation
    case e => {
      val e_ = step(e)
      val ty_ = hastype(e_)
      require(ty == ty_)
      Some(e_)
    }
  }
}
```

```
defined function iterate
defined function iterateStepPAP
```

in contrast to the "production" `iterateStep` that type checks statically before evaluation at full speed:

```
def iterateStep(e: Expr): Expr = {
  // Check e is well-typed in that it doesn't throw StaticTypeError.
  val _ = hastype(e)
  // Iterate step at full speed.
  iterate(e) {
    // ReducesToValue
    case v if isValue(v) => None
    // ReducesToStep
    case e => Some(step(e))
  }
}
```

```
defined function iterateStep
```

They should evaluate a given expression to the same value:

```
val v_welltyped_step = iterateStep(e_welltyped)
val v_welltyped_steppap = iterateStepPAP(e_welltyped)
assert(v_welltyped_step == v_welltyped_steppap)
```

```
v_welltyped_step: Expr = N(n = 4.0)
v_welltyped_steppap: Expr = N(n = 4.0)
```

27 Lazy Evaluation

We have seen short-circuiting evaluation (Section 21.6), which is a particular instance of *lazy evaluation* where some sub-expression is conditionally evaluated.

In this chapter, we consider *call-by-name*, which is another of form of lazy evaluation in defining the semantics of function call $e_1\texttt{(}e_2\texttt{)}$. In contrast to *call-by-value*, call-by-name semantics does not evaluate the function argument to a value before starting to evaluate the function body. Instead, it takes the unevaluated argument expression and substitutes it for the formal parameter.

Consider two possible DoCall rules:

$$\frac{}{\texttt{((}x\texttt{) => }e_1\texttt{)(}v_2\texttt{)} \longrightarrow [v_2/x]e_1}\ \textsc{DoCallByValue} \qquad \frac{}{\texttt{((}x\texttt{) => }e_1\texttt{)(}e_2\texttt{)} \longrightarrow [e_2/x]e_1}\ \textsc{DoCallByName}$$

It is one tiny difference on paper that is a substantively different semantically. The DoCallByValue rule requires that the argument be eagerly evaluated to a value before applying the substitution, while the DoCallByName rule does not. Call-by-name is lazy in that if e_1 does not end up using the parameter x in the subsequent evaluation, then e_2 will not be evaluated (i.e., like being "short-circuited").

Formalization - formalize when an expression is reducible given the parameter passing mode

28 Lab: Static Type Checking

Learning Goals

The primary goals of this lab are:

- Programming with higher-order functions.
- Static type checking and understanding the interplay between type checking and evaluation.

Functional Programming Skills Higher-order functions with collections and callbacks.
Programming Language Ideas Static type checking and type safety. Records.

Instructions

A version of project files for this lab resides in the public pppl-lab4 repository. Please follow separate instructions to get a private clone of this repository for your work.

You will be replacing ??? or `case _ => ???` in the `Lab4.scala` file with solutions to the coding exercises described below.

Your lab will not be graded if it does not compile. You may check compilation with your IDE, `sbt compile`, or with the "sbt compile" GitHub Action provided for you. Comment out any code that does not compile or causes a failing assert. Put in `???` as needed to get something that compiles without error.

You may add additional tests to the `Lab4Spec.scala` file. In the `Lab4Spec.scala`, there is empty test class `Lab4StudentSpec` that you can use to separate your tests from the given tests in the `Lab4Spec` class. You are also likely to edit `Lab4.worksheet.sc` for any scratch work. You can also use `Lab4.worksheet.ts` to write and experiment in a JavaScript file that you can then parse into a TypeScripty AST (see `Lab4.worksheet.sc`).

If you like, you may use this notebook for experimentation. However, **please make sure your code is in `Lab4.scala`; code in this notebook will not graded.**

28.1 Static Typing: TypeScripty: Functions and Objects

Static Typing

As we have seen in the prior labs, dealing with coercions and checking for dynamic type errors complicate the interpreter implementation (i.e., `step`). Some languages restrict the possible programs that it will execute to ones that it can guarantee will not result in a dynamic type error. This restriction of programs is enforced with an analysis phase after parsing but before evalation known as *type checking*. Such languages are called *statically-typed*. In this lab, we implement a statically-typed version of JavaScripty that we affectionately call TypeScripty. We will not permit *any* type coercions and simultaneously guarantee the absence of dynamic type errors.

Multi-Parameter Recursive Functions

Using our skills working with higher-order functions on collections from previous assignments, we now consider functions with zero-or-more parameters (instead of exactly one):

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & (\overline{y\colon\tau}) \Rightarrow \tau' \\
\text{values} & v & ::= & x^{?}(\overline{y\colon\tau})\tau^{?} \Rightarrow e_1 \\
\text{expressions} & e & ::= & x^{?}(\overline{y\colon\tau})\tau^{?} \Rightarrow e_1 \mid e_1(e_2) \\
\text{optional variables} & x^{?} & ::= & x \mid \varepsilon \\
\text{optional type annotations} & \tau^{?} & ::= & :\tau \mid \varepsilon
\end{array}
$$

We write a sequence of things using either an overbar or dots (e.g., $\overline{y}$ or $y_1, \dots, y_n$ for a sequence of variables). Functions can now take any number of parameters $\overline{y\colon\tau}$. We have a language of types τ and function parameters $\overline{y}$ are annotated with types $\overline{\tau}$.

Functions can be named or unnamed $x^{?}$ and can be annotated with a return type or unannotated $\tau^{?}$. To define recursive functions, the function needs to be named *and* annotated with a return type.

To represent an arbitrary number of function parameters or function call arguments in Scala, we use an appropriate `List`:

```
trait Typ  // t
case class TFun(yts: List[(String,Typ)], tret: Typ) extends Typ // t ::= (yts) => tret

trait Expr // e
case class Fun(xopt: Option[String], yts: List[(String,Typ)], tretopt: Option[Typ], e1: Expr
case class Call(e0: Expr, es: List[Expr]) extends Expr
```

```
defined trait Typ
defined class TFun
defined trait Expr
defined class Fun
defined class Call
```

Immutable Objects (Records)

Similarly, we now consider immutable objects that can have an arbitrary number of fields:

$$\begin{array}{rcll} \text{types} & \tau & ::= & \{\overline{f : \tau}\} \\ \text{values} & v & ::= & \{\overline{f : v}\} \\ \text{expressions} & e & ::= & \{\overline{f : e}\} \mid e_1 . f \end{array}$$

An object literal expression

$$\{f_1 : e_1 , \ldots , f_n : e_n\}$$

is a comma-separated sequence of field names with initialization expressions surrounded by braces. Objects here are more like records in other programming languages compared to actual JavaScript objects, as we do not have any form of mutation or dynamic extension. Fields here correspond to what JavaScript calls properties but which can be dynamically added or removed from objects. We use the term fields to emphasize that they are fixed based on their type:

$$\{f_1 : e_1 , \ldots , f_n : e_n\} \quad : \quad \{f_1 : \tau_1 , \ldots , f_n : \tau_n\}$$

Note that an object value is an object literal expression where each field is a value:

$$\{f_1 : v_1 , \ldots , f_n : v_n\}$$

The field read expression $e_1 . f$ evaluates e_1 to an object value and then looks up the field named f.

To represent object types and object literal expressions in Scala, we use an appropriate `Map`:

```scala
case class TObj(fts: Map[String, Typ]) extends Typ    // t ::= { fts }
case class Obj(fes: Map[String,Expr]) extends Expr    // e ::= { fes }
case class GetField(e1: Expr, f: String) extends Expr // e ::= e1.f
```

```
defined class TObj
defined class Obj
defined class GetField
```

Otherwise, we consider our base JavaScripty language that has numbers with arithmetic expressions, booleans with logic and comparison expressions, strings with concatenation, **undefined** with printing, and **const**-variable declarations. In summary, the type language τ includes base types for numbers, booleans, strings, and **undefined**, as well as constructed types for functions and objects described above:

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \texttt{number} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{Undefined} \\ & & \mid & (\overline{y\colon \tau}) \Rightarrow \tau' \mid \{\overline{f\colon \tau}\} \end{array}$$

As an aside, we have chosen a concrete syntax that is compatible with the TypeScript language that adds typing to JavaScript. TypeScript is a proper superset of JavaScript, so it is not as strictly typed as TypeScripty is here in this lab.

28.2 Interpreter Implementation

We break our interpreter implementation into evaluation and type checking.

Small-Step Reduction

For evaluation, we continue with implementing a small-step operational semantics with a `step` that implements a single reduction step $e \longrightarrow e'$ on closed expressions. Because of the static type checking, the reduction-step cases can be greatly simplified: we eliminate performing all coercions, and what's cool is that we no longer need to represent the possibility of a dynamic typeerror (e.g., with a `Either[DynamicTypeError,Expr]`).

We can use the more basic type signature for `step`:

```
def step(e: Expr): Expr = ???
```

```
defined function step
```

corresponding to the more basic judgment form $e \longrightarrow e'$ (given in the subsequent sections).

To make easier to identify implementation bugs, we introduce another Scala exception type to throw when there is no possible next step.

```
case class StuckError(e: Expr) extends Exception
```

```
defined class StuckError
```

However, the intent of this exception is that it should get thrown at run-time! If it does get thrown, that signals a bug in our interpreter implementation rather than an error in the TypeScripty test input.

In particular, if the TypeScripty expression `e` passed into `step` is closed and well-typed (i.e., `inferType(e)` does not throw `StaticTypeError`), then `step` should never throw a `StuckError`. This property is *type safety*.

Recall that to implement `step`, we need to implement a substitution function `substitute` corresponding to $[v/x]e$ that we use to eagerly apply variable bindings:

```
def substitute(v: Expr, x: String, e: Expr) = ???
```

```
defined function substitute
```

Static Type Checking

We implement a static type checker that up front rules out programs that would get stuck in taking reduction steps. This type checker is very similar to a big-step interpreter. Instead of computing the value of an expression by recursively computing the value of each sub-expression, we infer the type of an expression, by recursively inferring the type of each sub-expression. An expression is *well-typed* if we can infer a type for it.

Given its similarity to big-step evaluation, we formalize a type inference algorithm in a similar way. That is, we define the judgment form $\Gamma \vdash e : \tau$, which says, "In type environment Γ, expression e has type τ." We then implement a function `hastype`:

```
type TEnv = Map[String, Typ]
def hastype(tenv: TEnv, e: Expr): Typ = ???
```

```
defined type TEnv
defined function hastype
```

that corresponds directly to this judgment form. It takes as input a type environment `tenv: TEnv` (Γ) and an expression `e: Expr` (e) to return a type `Typ` (τ). It is informative to compare the rules defining typing with a big-step operational semantics.

To signal a type error, we will use a Scala exception

```
case class StaticTypeError(tbad: Typ, esub: Expr, e: Expr) extends Exception
```

```
defined class StaticTypeError
```

where `tbad` is the type that is inferred sub-expression `esub` of input expression `e`. These arguments are used to construct a useful error message. We also provide a helper function `err` to simplify throwing this exception.

While it is possible to implement iterative reduction via `step` and type inference via`hastype` independently, it is generally easier to "incrementally grow the language" by going language-feature by-language-feature for all functions rather than function-by-function. In the subsequent steps, we describe the small-step operational semantics and the static typing semantics together incrementally by language feature.

Notes

For testing your implementation, there are some interface functions defined that calls your `step` and `hastype` implementations with some debugging information:

- The `iterateStep: Expr => Expr` function repeatedly calls your `step` implementation until reaching a value.
- The `inferType: Expr => Typ` function calls your `hastype` function with an empty type environment.

Note that the provided tests are minimal. **You will want to add your own tests to cover most language features.**

28.3 Base TypeScripty

28.3.1 Small-Step Reduction

We consider the base TypeScripty that has numbers with arithmetic expressions, booleans with logic and comparison expressions, strings with concatenation, **undefined** with printing, and **const**-variable declarations from previous assignments and remove all coercions.

Exercise 28.1 (Small-Step Reduction for Base TypeScripty)**.** Implement `step` for base TypeScripty following the small-step operational semantics in Figure 28.1 defining the reduction-step judgment form $e \longrightarrow e'$.

Note that your task here is simpler than what you have done before in previous assignments. There are no judgment forms or rules defining coercions (e.g., `toBoolean`) or stepping to a typeerror result (e.g., `Left(DynamicTypeError(e))`).

You will need to implement a helper function `substitute` for base TypeScripty to perform scope-respecting substitution $[v/x]e$ as in previous assignments.

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

$$\textsc{DoNeg}\quad \frac{n' = -n_1}{\texttt{-}\, n_1 \longrightarrow n'}$$

$$\textsc{DoArith}\quad \frac{n' = n_1 \; bop \; n_2 \qquad bop \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{n_1 \; bop \; n_2 \longrightarrow n'}$$

$$\textsc{DoPlusString}\quad \frac{str' = str_1 str_2}{str_1 \,\texttt{+}\, str_2 \longrightarrow str'}$$

$$\textsc{DoInequalityNumber}\quad \frac{b' = n_1 \; bop \; n_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{n_1 \; bop \; n_2 \longrightarrow b'}$$

$$\textsc{DoInequalityString}\quad \frac{b' = str_1 \; bop \; str_2 \qquad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{str_1 \; bop \; str_2 \longrightarrow b'}$$

$$\textsc{DoEquality}\quad \frac{b' = (v_1 \; bop \; v_2) \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{v_1 \; bop \; v_2 \longrightarrow b'}$$

$$\textsc{DoNot}\quad \frac{b' = \neg b_1}{\texttt{!}\, b_1 \longrightarrow b'}$$

$$\textsc{DoAndTrue}\quad \frac{}{\mathbf{true} \;\texttt{\&\&}\; e_2 \longrightarrow e_2}$$

$$\textsc{DoAndFalse}\quad \frac{}{\mathbf{false} \;\texttt{\&\&}\; e_2 \longrightarrow \mathbf{false}}$$

$$\textsc{DoOrTrue}\quad \frac{}{\mathbf{true} \;\texttt{||}\; e_2 \longrightarrow \mathbf{true}}$$

$$\textsc{DoOrFalse}\quad \frac{}{\mathbf{false} \;\texttt{||}\; e_2 \longrightarrow e_2}$$

$$\textsc{DoIfTrue}\quad \frac{}{\mathbf{true} \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_2}$$

$$\textsc{DoIfFalse}\quad \frac{}{\mathbf{false} \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_3}$$

$$\textsc{DoSeq}\quad \frac{}{v_1 \;\texttt{,}\; e_2 \longrightarrow e_2}$$

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

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

$$\textsc{SearchUnary}\quad \frac{e_1 \longrightarrow e_1'}{uop \, e_1 \longrightarrow uop \, e_1'}$$

$$\textsc{SearchBinary1}\quad \frac{e_1 \longrightarrow e_1'}{e_1 \; bop \; e_2 \longrightarrow e_1' \; bop \; e_2}$$

$$\textsc{SearchBinary2}\quad \frac{e_2 \longrightarrow e_2'}{v_1 \; bop \; e_2 \longrightarrow v_1 \; bop \; e_2'}$$

$$\textsc{SearchIf}\quad \frac{e_1 \longrightarrow e_1'}{e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \longrightarrow e_1' \;\texttt{?}\; e_2 \;\texttt{:}\; e_3}$$

$$\textsc{SearchPrint}\quad \frac{e_1 \longrightarrow e_1'}{\texttt{console.log(}e_1\texttt{)} \longrightarrow \texttt{console.log(}e_1'\texttt{)}}$$

$$\textsc{SearchConst}\quad \frac{e_1 \longrightarrow e_1'}{\mathbf{const}\; x \;\texttt{=}\; e_1 \texttt{;}\; e_2 \longrightarrow \mathbf{const}\; x \;\texttt{=}\; e_1' \texttt{;}\; e_2}$$

Figure 28.1: Small-step operational semantics of base TypeScripty, including numbers with arithmetic expressions, booleans with logic and comparison expressions, strings with concatenation, **undefined** with printing, and **const**-variable declarations.

Notes

- You may use (or ignore) the provided helper function `doInequality` to implement the DoInequalityNumber and DoInequalityString rules.

28.3.2 Static Type Checking

We define static typing with the judgment form $\Gamma \vdash e : \tau$ of base TypeScripty that has numbers with arithmetic expressions, booleans with logic and comparison expressions, strings with concatenation, **undefined** with printing, and **const**-variable declarations.

Observe how closely the Type rules align with the Do rules in Figure 28.1, except for having a big-step evaluation structure with types.

Exercise 28.2 (Static Type Checking for Base TypeScripty)**.** Implement a function `hastype` for base TypeScript following the static typing semantics in Figure 28.2 defining the typing judgment form $\Gamma \vdash e : \tau$.

```
type TEnv = Map[String, Typ]
def hastype(tenv: TEnv, e: Expr): Typ = ???
```

```
defined type TEnv
defined function hastype
```

28.4 Immutable Objects (Records)

Next, we extend our interpreter implementation for immutable objects. We consider the implementation for immutable objects next, as it is a bit simpler than that for multi-parameter functions.

28.4.1 Small-Step Reduction

We extend the reduction-step judgment form $e \longrightarrow e'$ for immutable objects:

Exercise 28.3 (Small-Step Reduction for Immutable Objects)**.** Implement the cases in `step` for reducing immutable object expressions using the rules given in Figure 28.3 for the reduction-step judgment form $e \longrightarrow e'$.

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

$$\textsc{TypeNumber}\quad \frac{}{\Gamma \vdash n : \texttt{number}} \qquad \textsc{TypeString}\quad \frac{}{\Gamma \vdash \mathit{str} : \texttt{string}} \qquad \textsc{TypeNeg}\quad \frac{\Gamma \vdash e_1 : \texttt{number}}{\Gamma \vdash \texttt{-}\,e_1 : \texttt{number}}$$

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

$$\textsc{TypePlusString}\quad \frac{\Gamma \vdash e_1 : \texttt{string} \qquad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1 \,\texttt{+}\, e_2 : \texttt{string}}$$

$$\textsc{TypeInequalityNumber}\quad \frac{\Gamma \vdash e_1 : \texttt{number} \qquad \Gamma \vdash e_2 : \texttt{number} \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{bool}}$$

$$\textsc{TypeInequalityString}\quad \frac{\Gamma \vdash e_1 : \texttt{string} \qquad \Gamma \vdash e_2 : \texttt{string} \qquad \mathit{bop} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{bool}}$$

$$\textsc{TypeEquality}\quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{bool}} \qquad \textsc{TypeBool}\quad \frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \textsc{TypeNot}\quad \frac{\Gamma \vdash e_1 : \texttt{bool}}{\Gamma \vdash \texttt{!}\,e_1 : \texttt{bool}}$$

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

$$\textsc{TypeIf}\quad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 : \tau} \qquad \textsc{TypeUndefined}\quad \frac{}{\Gamma \vdash \textbf{undefined} : \texttt{Undefined}}$$

$$\textsc{TypeSeq}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ \texttt{,}\ e_2 : \tau_2} \qquad \textsc{TypePrint}\quad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \texttt{console.log(}e_1\texttt{)} : \texttt{Undefined}} \qquad \textsc{TypeVar}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{TypeConstDecl}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{const}\ x \,\texttt{=}\, e_1\texttt{;}\ e_2 : \tau_2}$$

Figure 28.2: Typing of base TypeScripty, including numbers with arithmetic expressions, booleans with logic and comparison expressions, strings with concatenation, **undefined** with printing, and **const**-variable declarations.

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

$$\text{DoGetField}\quad \frac{}{\{f_1: v_1, \ldots, f_i: v_i, \ldots, f_n: v_n\}.f_i \longrightarrow v_i}$$

$$\text{SearchObject}\quad \frac{e_i \longrightarrow e_i' \qquad e_j = v_j \qquad \text{for all } j < i}{\{f_1: e_1, \ldots, f_i: e_i, \ldots\} \longrightarrow \{f_1: e_1, \ldots, f_i: e_i', \ldots\}}$$

$$\text{SearchGetField}\quad \frac{e_1 \longrightarrow e_1'}{e_1.f \longrightarrow e_1'.f}$$

Figure 28.3: Small-step operational semantics of TypeScripty with immutable objects.

Notes

- Field names f are different than variable names x, even though they are both represented in Scala with a `String`. Object expressions are not variable binding constructs—what does that mean about `substitute` for them?
- For SearchObject, you should make the reduction step apply to the first non-value as given by the left-to-right iteration of the collection using the `find` method on `Map`s:

```
(m: Map[K,V]).find(f: ((K,V)) => Boolean): Option[(K,V)]
```

- Other helpful Scala library methods not previously mentioned to use here include the following:

```
(m: Map[K,V]).get(k: K): Option[V]
```

28.4.2 Static Type Checking

We extend the static typing judgment form $\Gamma \vdash e : \tau$ for immutable objects:

$$\text{TypeObject}\quad \frac{\Gamma \vdash e_i : \tau_i \qquad \text{for all } i}{\Gamma \vdash \{\ldots, f_i: e_i, \ldots\} : \{\ldots, f_i: \tau_i, \ldots\}}$$

$$\text{TypeGetField}\quad \frac{\Gamma \vdash e : \{\ldots, f: \tau, \ldots\}}{\Gamma \vdash e.f : \tau}$$

Figure 28.4: Typing of TypeScripty with immutable objects.

Exercise 28.4 (Static Type Checking for Immutable Objects)**.** Implement the cases in `hastype` for typing immutable object expressions using the rules given in Figure 28.4 defining the typing judgment form $\Gamma \vdash e : \tau$.

Notes

- Other helpful Scala library methods not previously mentioned to use here include the following:

```
(m: Map[K,V]).map(f: ((K,V)) => (J,U)): Map[J,U]
```

28.5 Multi-Parameter Recursive Functions

Finally, we extend our interpreter implementation for multi-parameter recursive functions.

28.5.1 Small-Step Reduction

We extend the reduction-step judgment form $e \longrightarrow e'$ for multi-paramter recursive functions:

$$\boxed{e \longrightarrow e'} \qquad \frac{\textsc{DoCall}}{((y_1 : \tau_1\,,\,\ldots\,,y_n : \tau_n)\tau^? \Rightarrow e)(v_1, \ldots v_n) \longrightarrow [v_1/y_1]\cdots[v_n/y_n]e}$$

$$\textsc{DoCallRec}\;\frac{v = (x(y_1 : \tau_1\,,\,\ldots\,,y_n : \tau_n) : \tau' \Rightarrow e)}{v(v_1, \ldots v_n) \longrightarrow [v/x][v_1/y_1]\cdots[v_n/y_n]e} \qquad \textsc{SearchCall1}\;\frac{e \longrightarrow e'}{e(e_1, \ldots, e_n) \longrightarrow e'(e_1, \ldots, e_n)}$$

$$\textsc{SearchCall2}\;\frac{e_i \longrightarrow e_i' \qquad e_j = v_j \qquad \text{for all } j < i}{v(e_1, \ldots, e_i, \ldots, e_n) \longrightarrow v(e_1, \ldots, e_i', \ldots, e_n)}$$

Figure 28.5: Small-step operational semantics for TypeScripty with multi-parameter recursive functions.

Exercise 28.5 (Small-Step Reduction for Multi-Parameter Recursive Functions)**.** Implement the cases in `step` for reducing multi-parameter recursive functions using the rules given in Figure 28.5 for the reduction-step judgment form $e \longrightarrow e'$.

Notes

- Other helpful Scala library methods not previously mentioned to use here include the following:

```
(l: List[A]).map(f: A => B): List[B]
(l: List[A]).exists(f: A => Boolean): Boolean
(la: List[A]).zip(lb: List[B]): List[(A,B)]
(l: List[A]).forall(f: A => Boolean): Boolean
(l: List[A]).foldRight(f: (A,B) => B): B
```

- You may want to use the `zip` method for the DoCall and DoCallRec cases to match up formal parameters and actual arguments.

- You might want to use your `mapFirst` function from Homework 4 here.

28.5.2 Static Type Checking

We extend the static typing judgment form $\Gamma \vdash e : \tau$ for multi-parameter recursive functions:

$$\text{TypeCall}\quad \frac{\Gamma \vdash e : (y_1 : \tau_1, \ldots, y_n : \tau_n) \Rightarrow \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$$\text{TypeFunction}\quad \frac{\Gamma, y_1 : \tau_1, \cdots, , y_n : \tau_n \vdash e' : \tau'}{\Gamma \vdash (\overline{y : \tau}) \Rightarrow e' : (\overline{y : \tau}) \Rightarrow \tau'} \qquad \text{TypeFunctionAnn}\quad \frac{\Gamma, y_1 : \tau_1, \cdots, , y_n : \tau_n \vdash e' : \tau'}{\Gamma \vdash (\overline{y : \tau}) : \tau' \Rightarrow e' : (\overline{y : \tau}) \Rightarrow \tau'}$$

$$\text{TypeFunctionRec}\quad \frac{\Gamma, x : \tau_x, y_1 : \tau_1, \cdots, , y_n : \tau_n \vdash e' : \tau' \qquad \tau_x = (\overline{y : \tau}) \Rightarrow \tau'}{\Gamma \vdash x(\overline{y : \tau}) : \tau' \Rightarrow e' : \tau_x}$$

Figure 28.6: Typing of TypeScripty with multi-parameter recursive functions.

Exercise 28.6 (Static Type Checking for Immutable Objects)**.** Implement the cases in `hastype` for typing multi-parameter recursive function expressions using the rules given in Figure 28.6 defining the typing judgment form $\Gamma \vdash e : \tau$.

Notes

- Other helpful Scala library methods not previously mentioned to use here include the following:

```
(l: List[A]).foldLeft(f: (B,A) => B): B
(l: List[A]).foreach(f: A => Unit): Unit
(l: List[A]).length: Int
(m1: Map[K,V]).++(m2: Map[K,V]): Map[K,V]
```

- The ++ method on `Map`s appends two `Map`s together.

29 Review: Higher-Order Functions and Static Checking

Instructions

This assignment is a review exercise in preparation for a subsequent assessment activity.

This is a peer-quizzing activity with two students. Each section has an even number of exercises. Student A quizzes Student B on the odd numbered exercises, and Student B quizzes Student A on the even numbered exercises.

To the best of your ability, give feedback using the learning-levels rubric below on where your peer is in reaching or exceeding Proficient (P) on each question live. Guidance of what a Proficient (P) answer looks like are given.

There may or may not be a member of the course staff assigned to your slot. It is expected that regardless of whether a member of the course staff is present, this is a peer-quizzing activity. If a member of the course staff is present, you may ask for their help and guidance on answering the questions and/or their assessment of where you are at in your learning level.

It is not expected that you can complete all exercises in the allotted time. You and your partner may pick and choose which sections you want to focus on and use the remaining questions as a study guide. You and your partner may, of course, continue working together after the scheduled session.

At the same time, most questions can be answered in a few minutes with a Proficient (P) level of understanding. Aim for 3–4 sections in 30 minutes.

Your submission for this session is an overall assessment of where your partner is in their reaching-or-exceeding-proficiency level. Be constructive and honest. **Neither your nor your partners grade will depend on your learning-level assessment.** Instead, your score for this assignment will be based on the thoughtfulness of your feedback to your partner.

Submit on Gradescope as a pair. That is, use Gradescope's group assignment feature to submit as a group. The submission form has a spot for each of you to provide your assessment and feedback for each other.

Please proactively fill slots with an existing sign-up to have a partner. In case your peer does not show up to the slot, try to join another slot happening at the same time from the course calendar. If that fails and a course staff member is present, you may do the exercise with the staff member and get credit. If there is no staff member present, you may try to find a slot at a later time if you like or else write to the Course Manager on Piazza timestamped during the slot.

Learning-Levels Rubric

4 - Exceeding (E) Student demonstrates synthesis of the underlying concepts. Student can go beyond merely describing the solution to explaining the underlying reasoning and discussing generalizations.

3 - Proficient (P) Student is able to explain the overall solution and can answer specific questions. While the student is capable of explaining their solution, they may not be able to confidently extend their explanation beyond the immediate context.

2 - Approaching (A) Student may able to describe the solution but has difficulty answering specific questions about it. Student has difficulty explaining the reasoning behind their solution.

1 - Novice (N) Student has trouble describing their solution or responding to guidance. Student is unable to offer much explanation of their solution.

29.1 Higher-Order Functions

Exercise 29.1. Suppose you have a very large list of floating-point numbers stored in a parallel sequence, for example, `ParSeq(0.1, 12, -500, 76.33, 0, -9.9)`. Write a function `squareRootSum` that computes the sum of the square roots of all the positive numbers in this sequence. You can use the `scala.math.sqrt` function to compute square roots.

```
import scala.math.sqrt
import $ivy.`org.scala-lang.modules::scala-parallel-collections:1.0.4`, scala.collection.para

def squareRootSum(l: ParSeq[Double]): Double = ???
```

```
import scala.math.sqrt

import $ivy.$                                                              , scala.collection.para


defined function squareRootSum
```

Note that `ParSeq` is an abstract data type that has the same higher-order iteration methods as `List`.

A Proficient (P) answer will recall the higher-order functions `filter`, `map`, and one of `foldLeft`, `foldRight` or `reduce`, and use them to filter only positive numbers, compute the square root, and sum the list elements respectively.

```
def squareRootSum(l: ParSeq[Double]): Double =
  l filter {_ > 0} map {sqrt} reduce {_ + _}
```

```
defined function squareRootSum
```

Exercise 29.2. Consider the following Scala expression. Can you figure out its type?

```
List(1,2,2,3,3,3,4,4,4,4).foldRight[(List[Int], List[List[Int]])]((Nil, Nil)) {
  (h, acc) => acc match {
    case (Nil, pacc) => (h :: Nil, pacc)
    case (lacc @ (p :: _), pacc) =>
      if (h == p) (h :: lacc, pacc) else (h :: Nil, lacc :: pacc)
  }
}
```

A Proficient (P) answer will look at the value of the second case, and see that we are always consing a `List[Int]` onto another list deduce that the answer is `(List[Int], List[List[Int]])`.

Another Proficient (P) answer may see observe that the type argument of `foldRight` is also its return type, and therefore deduce the answer is `(List[Int], List[List[Int]])`.

Exercise 29.3. Consider the same Scala expression above. Explain what the `foldRight` call with the given callback function does. What value will the `foldRight` call return?

Hint: You could step through the first several iterations of the folding function and see what the value of `acc` becomes each time.

A Proficient (P) answer will recall the type deduced above, and then step through the code and see that the left side of the accumulator remembers the current list of consecutively equal-valued integers in the input, while the right side collects such lists when the the consecutive integer values in the input differ. It will then extrapolate to get the final output of `(List(1), List(List(2, 2), List(3, 3, 3), List(4, 4, 4, 4)))`.

Selection sort is a sorting algorithm that repeatedly removes the smallest element of an unsorted list to create a new sorted list. In the next few exercises, we will implement selection sort to sort a list `l: List[A]` by a custom specified `le: (A, A) => Boolean` comparison function. The `le(x,y)` comparison function returns `true` if `x` is less-than-or-equal to `y`.

```
def sortBy[A](l: List[A], le: (A, A) => Boolean): List[A] = ???
```

```
defined function sortBy
```

Exercise 29.4. First, let's write a function `findMin` to find the minimum element of `l` according to `le` if the `l` is not empty. The `findMin` function returns `None` if `l` is `Nil` and otherwise `Some(a)` for the minimum element according to `le`. Do not use recursion and instead use higher-order iteration methods.

```
def findMin[A](l: List[A], le: (A, A) => Boolean): Option[A] = ???
```

```
defined function findMin
```

You may pattern match for `Nil` or use `isEmpty: List[A] => Boolean` to detect if `l` is empty.

A Proficient (P) answer might use `foldRight` or `foldLeft`:

```
def findMin[A](l: List[A], le: (A, A) => Boolean): Option[A] =
  l.foldRight(None: Option[A]) {
    case (x, None) => Some(x)
    case (x, Some(y)) => if (le(x, y)) Some(x) else Some(y)
  }
```

```
defined function findMin
```

An Exceeding (P) answer might use `reduceRight` instead to apply `le` to each pair of elements starting from the right, and take the minimum of them to compare to the next one.

```
def findMin[A](l: List[A], le: (A, A) => Boolean): Option[A] =
  if (l.isEmpty) None
  else Some(l reduceRight { (x, y) => if (le(x, y)) x else y })
```

```
defined function findMin
```

Exercise 29.5. Next, write a function to remove any specified element `e` from `l` if it exists. Again, do not use recursion and instead use higher-order iteration methods.

```
def removeOne[A](e: A, l: List[A]): List[A] = ???
```

```
defined function removeOne
```

Hint: You can write a recursive version of `removeOne` and then translate it into a `foldRight` or `foldLeft`.

> A Proficient (P) answer will observe that we need a flag to determine whether we have removed an element already or not:

```
def removeOne[A](e: A, l: List[A]): List[A] = {
  val (_, accl) = l.foldRight((false, Nil: List[A])) {
    case (x, (false, accl)) if x == e => (true, accl)
    case (x, (removed, accl)) => (removed, x :: accl)
  }
  accl
}
```

```
defined function removeOne
```

Exercise 29.6. Finally, combine both of these functions to complete the implementation of `sortBy`. You may define `sortBy` using recursion.

```
def sortBy[A](l: List[A], le: (A, A) => Boolean): List[A] = ???
```

```
defined function sortBy
```

> A Proficient (P) answer will combine the two functions to first find the min, put it at the head of a list, then recursively sort the remainder of a list with that element removed.

```
def sortBy[A](l: List[A], le: (A, A) => Boolean): List[A] = findMin(l, le) match {
  case None => Nil
  case Some(min) => min :: sortBy(removeOne(min, l), le)
}
```

```
defined function sortBy
```

29.2 Static Typing

Consider the syntax of TypeScripty with base values, functions, and immutable objects:

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & \texttt{number} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{Undefined} \\
 & & \mid & (\overline{y\colon \tau}) \Rightarrow \tau' \mid \{\overline{f\colon \tau}\} \\
\text{expressions} & e & ::= & n \mid b \mid \mathit{str} \mid \textbf{undefined} \mid \mathit{uop}\, e_1 \mid e_1\ \mathit{bop}\ e_2 \mid \cdots \\
 & & \mid & (\overline{y\colon \tau}) : \tau' \Rightarrow e_1 \mid e_1(e_2) \mid \{\overline{f\colon e}\} \mid e_1 . f \\
\text{variables} & x, y & & \\
\text{fields} & f & & \\
\text{numbers} & n & & \\
\text{booleans} & b & & \\
\text{strings} & \mathit{str} & &
\end{array}
$$

For simplicity, observe that function literals are anonymous and always have annotated return type.

Exercise 29.7. Define the values of this language (assuming the operational semantics is defined via substitution).

A Proficient (P) answer states a value form for each type:

$$
\text{values} \quad v \quad ::= \quad n \mid b \mid \mathit{str} \mid \textbf{undefined} \mid (\overline{y\colon \tau}) : \tau' \Rightarrow e_1 \mid \{\overline{f\colon v}\}
$$

An object value is one where each component of an object literal is a value.

An Exceeding (E) answer considers whether a function literal or a closure is the value form for function types. However, because the question states the semantics is defined via substitution, the answer can consider function literals as values.

An Exceeding (E) answer may also state that the above grammar for v is a short-hand for the unary judgment form e value that is analogous to how we implement in Scala with the `isValue` function:

$$
\boxed{e\ \mathsf{value}}
\qquad
\textsc{NumVal}\ \frac{}{n\ \mathsf{value}}
\qquad
\textsc{BoolVal}\ \frac{}{b\ \mathsf{value}}
\qquad
\textsc{StringVal}\ \frac{}{\mathit{str}\ \mathsf{value}}
\qquad
\textsc{UndefinedVal}\ \frac{}{\textbf{undefined}\ \mathsf{value}}
$$

$$
\textsc{FunctionVal}\ \frac{}{(\overline{y\colon \tau}) : \tau' \Rightarrow e_1\ \mathsf{value}}
\qquad
\textsc{ObjectVal}\ \frac{e_1\ \mathsf{value} \quad \cdots \quad e_n\ \mathsf{value}}{\{f_1 : e_1, \ldots, f_n : e_n\}\ \mathsf{value}}
$$

The e value judgment form makes it evident that with object literals, it must be an inductively-defined relation.

Exercise 29.8. Give typing rules for the function and object expressions forms:

$$(\overline{y\colon\tau})\colon\tau' \Rightarrow e_1 \qquad e_1(e_2) \qquad \{\overline{f\colon e}\} \qquad e_1.f$$

A Proficient (P) answer gives four typing rules: one for each form (e.g., TypeFunction, TypeCall, TypeObject, and GetField) from the preceding chapters.

An Exceeding (E) answer might note that with this simplified expression language for function literals, we only need one rule for function literals (corresponding to TypeFunctionAnn in the preceding chapters).

Exercise 29.9. Consider the following JavaScripty code:

```
const x = 7;
const y = "hello";
const b = x < 0;
const f = (n) => n + 5;
const test = {
    a: b,
    b: {z: y},
    c: b,
};
test.b.z
```

Suppose we want to refactor it into TypeScripty. What do we need to do?

A Proficient (P) answer will recognize that we need to annotate the function literal with types. Specifically, we need to update the line binding `f` as follows:

```
const f = (n: number): number => n + 5;
```

Exercise 29.10. The above code refactored into TypeScripty is well-typed. Consider this judgment that is in a sub-derivation of type checking the above TypeScripty code:

$$\Gamma \vdash \texttt{test.b.z} : \tau$$

What is the type environment Γ and the result type τ for this judgment?

A Proficient (P) answer will recognize Γ as resulting from the **const** bindings:

$$\Gamma: \quad \texttt{x} : \texttt{number}, \texttt{y} : \texttt{string}, \texttt{b} : \texttt{bool}, \texttt{f} : (\texttt{n}: \texttt{number}) \Rightarrow \texttt{number}, \\ \texttt{test} : \{\texttt{a}: \texttt{bool}, \texttt{b}: \{\texttt{z}: \texttt{string}\}, \texttt{c}: \texttt{bool}\}$$

which shows that τ is `string`.

Exercise 29.11. Now consider the following JavaScripty code snippet instead. Can we refactor this code snippet to TypeScripty? If yes, give their new types. If no, explain why, and give a possible solution.

```
const x = 7;
const y = "hello";
const b = x < 0;
const f = (n, m) => n + m;
const test = {
    a: b,
    b: {z: y},
    c: b,
};
const r1 = f(test.b.z, y);
const r2 = f(1, x);
const r3 = f(test.b.z, x);
```

A Proficient (P) answer will note that we can no longer annotate the type of `(n, m) => n + m`, since it is ambiguous whether it is concatenating strings or adding numbers now. It should recognize that + is being used for concatenating strings on the line binding `r1`, + is being used for adding numbers on the line binding `r2`, and + is being used for concatenating strings with a number to string coercion on the line binding `r3`. A Proficient (P) answer may say that it is not just not possible to do this refactoring.

An Exceeding (E) answer may say that you can create two function literals `(n: number, m: number): number => n + m` and `(n: string, m: string): string => n + m` for the `r1` and `r2` lines, respectively, but `(n: string, m: number): string => n + m` will not type check for the `r3` line. An Exceeding (E) answer may consider the possible solution of creating a type that includes both `number` and `string` and update typing to allow for coercions between numbers and strings. As a comment for an accelerated student, there are multiple ways to allow for this *polymorphism*, including subtyping and union types.

Recall that our type checking rule for `&&` and `||` was as follows:

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

Suppose we instead replaced this rule with the following four rules to try to more closely match our standard small-step operational semantics (cf. Figure 28.1 in Section 28.3) that does short-circuiting evaluation:

$$\frac{}{\Gamma \vdash \textbf{false} \texttt{ \&\& } e_2 : \texttt{bool}}\ \textsc{TypeAndFalseShort} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{true} \texttt{ \&\& } e_2 : \tau_2}\ \textsc{TypeAndTrueShort}$$

$$\frac{}{\Gamma \vdash \textbf{true} \texttt{ || } e_2 : \texttt{bool}}\ \textsc{TypeOrTrueShort} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{false} \texttt{ || } e_2 : \tau_2}\ \textsc{TypeOrFalseShort}$$

Exercise 29.12. Are these new rules **unsound**? Unsound means that it allows expressions to be type-checked that would then get stuck during evaluation using our standard small-step operational semantics (see Section 26.7 for further discussion about soundness). If you say they are sound, explain why our standard small-step interpreter cannot get stuck. If you say they are unsound, give an expression that type checks but would get stuck during evaluation.

> A Proficient (P) answer will explain that the rules are in fact still sound because the four rules correspond to the four Do rules that implement short-circuiting evaluation of `&&` and `||`.

Exercise 29.13. Compare this new set of rules (i.e., TypeAndFalseShort, TypeAndTrueShort, TypeOrTrueShort, TypeOrFalseShort) with the original set (i.e., TypeAndOr) for type checking. Are there expressions where the old set would allow but the new rules would not? If so, give an example expression and explain briefly. What about vice versa?

> An Exceeding (E) answer will see that the expressions that can type checked with the two sets are incomparable. For example, the **false** `&& 1` is well-typed with the new set but not the old set. And vice versa, the `(`**false** `&&` **false**`) &&` **true** is well-typed in the old set but not the new set.
>
> A Proficient (P) answer will likely see one direction but not the other. An example like **false** `&& 1` that is well-typed with the new set but not the old set is typically easier to see.

Exercise 29.14. Suppose that our type system is sound (i.e., our judgment form $\Gamma \vdash e : \tau$ is sound with respect to our small-step operational semantics $e \longrightarrow e'$). Further suppose that our implementations of these two judgment forms `hastype` and `step`, respectively, are correct. Now suppose that we have a closed TypeScripty expression `e` that type checks with `hastype` (i.e., `hastype(Map.empty, e)` does not throw `StaticTypeError`). Will iteratively running `step` on `e` always terminate in a value, or is it still possible that it results in an error? If you think it terminates in a value, justify why. If you think it could still encounter an error, give an example and explain what types of errors are possible.

> A Proficient (P) answer may note that there can still be other run-time issues like division by 0 or infinite loops. Therefore, there is no guarantee that the program will now terminate in a value.
>
> An alternative Proficient (P) answer can state that type checking will eliminate typing errors that would otherwise result in a `StuckError` or `MatchError` at run time. This is what is stated as soundness in Exercise 29.12.
>
> An Exceeding (E) answer will distinguish between typing errors that are all soundly caught at compile time and other run-time issues like division by 0 or infinite loops. The answer may state this is what is stated by the progress and preservation properties of a sound type system.

Part VI

Imperative Computation

30 Encapsulating Effects

In this chapter, we explore the ideas of abstract data types further. In particular, we see that we can generalize from collections with higher-order methods to other kinds of data structures with such methods that intuitively *encapsulate computational effects*.

30.1 Abstract Data Types

Recall that an *abstract data type* is a data type whose representation is abstract and unavailable to the client.

It is a concept that we have seen multiple times, for example, the `Map` and `Set` types in the Scala standard library abstracts the interface of a mathematical finite map or a finite set, respectively, whose actual representation is abstracted from the client of the library. This enables the library to maintain a representation invariant on behalf of the client (e.g., representing a finite map as a balanced binary search tree to maintain logarithmic lookup, insertion, and deletion).

We have seen that a library can provide higher-order methods to expose a view of the data type without exposing the internal representation (see Section 24.3). For example, using the `map` method on a `List` is a convenience

```
def inc(l: List[Int]): List[Int] = l.map(_ + 1)
inc(List(1, 2, 3))
```

```
defined function inc
res0_1: List[Int] = List(2, 3, 4)
```

over direct recursion

```
def inc(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => (h + 1) :: inc(t)
}
inc(List(1, 2, 3))
```

```
defined function inc
res1_1: List[Int] = List(2, 3, 4)
```

The view of a `Set` as a collection is only accessible via `map` and related higher-order methods:

```scala
def inc(s: Set[Int]): Set[Int] = s.map(_ + 1)
inc(Set(1, 2, 3))
```

```
defined function inc
res2_1: Set[Int] = Set(2, 3, 4)
```

30.2 Error Effects

Recall that distinction between effect-free and effect-ful computation. An *effect-free* (or *pure* or *referentially transparent*) computation is an evaluation of an expression that does not do anything external to its final value, while an *effect-ful* (or *side-effecting*) computation does do something that is not visible in its final value.

Perhaps the most basic *effect* is the possibility of error. For example, consider a function

```scala
toDoubleException: String => Double
```

that parses a string as a floating-point number and converts it into a double:

```scala
def toDoubleException(s: String): Double = s.toDouble

toDoubleException("1")
toDoubleException("4.2")
```

```
defined function toDoubleException
res3_1: Double = 1.0
res3_2: Double = 4.2
```

Of course, not all strings correspond to floating-point numbers, so `toDoubleException` throws an exception if it is unable to recognize the string as a floating-point number:

```scala
toDoubleException("hello")
```

This throwing of an exception is a side-effect because it is not captured in the return type `Double`. That is, all Scala expressions have the possibility of throwing an exception to bypass the expected type of the expression, so Scala is not pure language.

30.2.1 Option

With some discipline, we can attempt to program in a pure subset of Scala where we make explicit the possibility of error. For example, we can instead define

```
toDoubleOption: String => Option[Double]
```

to return an `Option`:

```
def toDoubleOption(s: String): Option[Double] =
  try { Some(s.toDouble) } catch { case _: NumberFormatException => None }

toDoubleOption("1")
toDoubleOption("4.2")
toDoubleOption("hello")
```

```
defined function toDoubleOption
res5_1: Option[Double] = Some(value = 1.0)
res5_2: Option[Double] = Some(value = 4.2)
res5_3: Option[Double] = None
```

An `Option[A]` type represents an optional value, and it is often used to represent possible error in computing a result of type `A`. That is, either return `Some(a)` of the result `a` of type `A` or `None` to indicate error.

The trade-off is that working with an `Option[Double]` is different than working with a `Double`. For example, suppose we define a function `toDoubleNoNaNOption` that excludes different versions of `"NaN"` as a string before calling `toDoubleOption`:

```
def toDoubleNoNaNOption(s: String): Option[Double] =
  // Do some work: trim the spaces from the end of s
  s.trim match {
    // Check for an error condition: if now the string is empty
    case s if s.length == 0 => None
    // Continue with some work: normalize to upper case
    case s => s.toUpperCase match {
      // Check for an error condition: the trimmed and upper-cased string is "NAN"
      case s if s == "NAN" => None
      // Continue with some work: convert to an Option[Double]
      case s => toDoubleOption(s)
    }
```

```
  }

toDoubleNoNaNOption("nan")
toDoubleNoNaNOption(" nan ")
toDoubleNoNaNOption("NaN")
toDoubleNoNaNOption(" NaN ")
```

```
defined function toDoubleNoNaNOption
res6_1: Option[Double] = None
res6_2: Option[Double] = None
res6_3: Option[Double] = None
res6_4: Option[Double] = None
```

Or as another example, suppose we define a function `addToDoubleOption` to convert two floating-point strings excluding `"NaN"` and then add them:

```
def addToDoubleOption(s1: String, s2: String): Option[Double] =
  toDoubleNoNaNOption(s1) match {
    // If we get None, then we return None indicating error.
    case None => None
    // If we get Some, then we can continue to do work.
    case Some(d1) => toDoubleNoNaNOption(s2) match {
      // If we get None, then we return None indicating error.
      case None => None
      // If we get Some, then we can continue to do work.
      case Some(d2) => Some(d1 + d2)
    }
  }

addToDoubleOption("1", "4.2")
addToDoubleOption("1", "hello")
addToDoubleOption("1", " nan")
```

```
defined function addToDoubleOption
res7_1: Option[Double] = Some(value = 5.2)
res7_2: Option[Double] = None
res7_3: Option[Double] = None
```

This works well in carefully avoiding errors with support from the Scala type checker to make sure we check for `None`. At the same time, there is a lot of `None` handling scaffolding mixed in with the “work”.

Now, let's think of `Option[A]` as zero-or-one element list. That is, `None` is the zero element list and `Some(a)` is the one element list with an `a: A`. Now, we rewrite these two functions using the higher-order methods that we are used to using on lists:

```
def toDoubleNoNaNOption(s: String): Option[Double] =
  Some(s)
  // Do some work: trim the spaces from the end of s
  .map(_.trim)
  // Check for an error condition: if now the string is empty
  .filter(_.length != 0)
  // Continue with some work: normalize to upper case
  .map(_.toUpperCase)
  // Check for an error condition: the trimmed and upper-cased string is "NAN"
  .filter(_ != "NAN")
  // Continue with some work: convert to an Option[Double]
  .flatMap(toDoubleOption(_))
```

```
defined function toDoubleNoNaNOption
```

```
def addToDoubleOption(s1: String, s2: String): Option[Double] =
  toDoubleNoNaNOption(s1) flatMap { d1 =>
    // If we get Some, then we can continue to do work.
    toDoubleNoNaNOption(s2) map { d2 =>
      // If we get Some, then we can continue to do work.
      d1 + d2
    }
  }

addToDoubleOption("1", "4.2")
addToDoubleOption("1", "hello")
addToDoubleOption("1", " nan")
```

```
defined function addToDoubleOption
res9_1: Option[Double] = Some(value = 5.2)
res9_2: Option[Double] = None
res9_3: Option[Double] = None
```

Wow, the `None` scaffolding is gone! The `None` handling scaffolding is precisely what is factored into the `map`, `filter`, and `flatMap` library methods. It is a good exercise to define these library methods:

Exercise 30.1. Implement `map` for `Option[A]`s:

```scala
def map[A, B](opt: Option[A])(f: A => B): Option[B] = ???
```

```
defined function map
```

Exercise 30.2. Implement `filter` for `Option[A]`s:

```scala
def filter[A](opt: Option[A])(f: A => Boolean): Option[A] = ???
```

```
defined function filter
```

Exercise 30.3. Implement `flatMap` for `Option[A]`s:

```scala
def flatMap[A, B](opt: Option[A])(f: A => Option[B]): Option[B] = ???
```

```
defined function flatMap
```

Comprehensions

Note the pattern of using a `map` nested in a `flatMap` is so common that a `for-yield` expression with multiple binders translates to exactly that. So for example, we can define `addToDoubleOption` as follows:

```scala
def addToDoubleOption(s1: String, s2: String): Option[Double] =
  for {
    d1 <- toDoubleNoNaNOption(s1)
    d2 <- toDoubleNoNaNOption(s2)
  } yield d1 + d2

addToDoubleOption("1", "4.2")
addToDoubleOption("1", "hello")
addToDoubleOption("1", " nan")
```

```
defined function addToDoubleOption
res13_1: Option[Double] = Some(value = 5.2)
res13_2: Option[Double] = None
res13_3: Option[Double] = None
```

We can see the sequence of binders <- corresponds to a sequential composition where we get first the double `d1` corresponding to `s1` and second the double `d2` corresponding to `s2` to add them. If either step errors (i.e., results in a `None`), then the whole **`for-yield`** expression evaluates to `None`.

In either case of using `flatMap` and `map` or the **`for-yield`** expressions, we have mostly recovered the minimal scaffolding from effect-ful exceptions while being effect-free with explicit `Options`.

30.2.2 Either

We have seen that `Either[Err, A]` is another data type that is often used for representing error effects where the error-case has some data of type `Err`. For example, we can use an `Either` to save the exception in the error case:

```
def toDoubleEither(s: String): Either[NumberFormatException, Double] =
  try { Right(s.toDouble) } catch { case e: NumberFormatException => Left(e) }
```

```
defined function toDoubleEither
```

And observe the code that uses `map` and `flatMap` works with either type:

```
def addToDoubleEither(s1: String, s2: String): Either[NumberFormatException, Double] =
  toDoubleEither(s1) flatMap { d1 =>
    toDoubleEither(s2) map { d2 =>
      d1 + d2
    }
  }

addToDoubleEither("1", "4.2")
addToDoubleEither("1", "hello")
```

```
defined function addToDoubleEither
res15_1: Either[NumberFormatException, Double] = Right(value = 5.2)
res15_2: Either[NumberFormatException, Double] = Left(
  value = java.lang.NumberFormatException: For input string: "hello"
)
```

```scala
def addToDoubleEither(s1: String, s2: String): Either[NumberFormatException, Double] =
  for {
    d1 <- toDoubleEither(s1)
    d2 <- toDoubleEither(s2)
  } yield d1 + d2

addToDoubleEither("1", "4.2")
addToDoubleEither("1", "hello")
```

```
defined function addToDoubleEither
res16_1: Either[NumberFormatException, Double] = Right(value = 5.2)
res16_2: Either[NumberFormatException, Double] = Left(
  value = java.lang.NumberFormatException: For input string: "hello"
)
```

30.2.3 Try

The Scala standard library has a data type `Try` specifically for representing exception effects:

```scala
import scala.util.Try
def toDoubleTry(s: String): Try[Double] =
  Try(s.toDouble)
```

```
import scala.util.Try

defined function toDoubleTry
```

```scala
def addToDoubleTry(s1: String, s2: String): Try[Double] =
  toDoubleTry(s1) flatMap { d1 =>
    // If we get Success, then we can continue to do work.
    toDoubleTry(s2) map { d2 =>
      // If we get Success, then we can continue to do work.
      d1 + d2
    }
  }

addToDoubleTry("1", "4.2")
addToDoubleTry("1", "hello")
```

```
defined function addToDoubleTry
res18_1: Try[Double] = Success(value = 5.2)
res18_2: Try[Double] = Failure(
  exception = java.lang.NumberFormatException: For input string: "hello"
)
```

```scala
def addToDoubleTry(s1: String, s2: String): Try[Double] =
  for {
    d1 <- toDoubleTry(s1)
    d2 <- toDoubleTry(s2)
  } yield d1 + d2

addToDoubleTry("1", "4.2")
addToDoubleTry("1", "hello")
```

```
defined function addToDoubleTry
res19_1: Try[Double] = Success(value = 5.2)
res19_2: Try[Double] = Failure(
  exception = java.lang.NumberFormatException: For input string: "hello"
)
```

30.3 Non-Determinism Effects

Another kind of effect is computation that is non-deterministic:

```scala
val rand = new scala.util.Random(0)
val r1 = rand.between(1,10)
val r2 = rand.between(1,10)
val r3 = rand.between(1,10)
val r4 = rand.between(1,10)
```

```
rand: scala.util.Random = scala.util.Random@3651e44b
r1: Int = 7
r2: Int = 8
r3: Int = 5
r4: Int = 3
```

While we do not necessarily see computations that return `List` as representing effects, we can represent non-determinism effects with a sequence:

```scala
val r = List(r1, r2, r3, r4)
```

```
r: List[Int] = List(7, 8, 5, 3)
```

And thus computations on top of non-deterministic computations correspond to applying `List` list methods. For example, let us show all pairs of results:

```scala
for {
  i <- r
  j <- r
} yield (i, j)
```

```
res22: List[(Int, Int)] = List(
  (7, 7),
  (7, 8),
  (7, 5),
  (7, 3),
  (8, 7),
  (8, 8),
  (8, 5),
  (8, 3),
  (5, 7),
  (5, 8),
  (5, 5),
  (5, 3),
  (3, 7),
  (3, 8),
  (3, 5),
  (3, 3)
)
```

30.4 Mutation Effects

The hallmark of imperative computation is *mutation* (or sometimes called assignment or imperative update). For example, suppose we define a function `freshVarImperative` that creates a globally unique variable name by keeping a counter:

```scala
var counter: Int = 0
def freshVarImperative: String = {
  val x = s"x${counter}"
  counter += 1
  x
}

val x0 = freshVarImperative
val x1 = freshVarImperative
val x2 = freshVarImperative
```

```
counter: Int = 3
defined function freshVarImperative
x0: String = "x0"
x1: String = "x1"
x2: String = "x2"
```

To represent a mutation effect, we see that what `freshVar` needs is the current counter that we view as input-output *state* of the `freshVar` function:

```scala
def freshVar: Int => (Int, String) = { counter =>
  val x = s"x${counter}"
  (counter + 1, x)
}

val counter = 0
val (counter_, x0) = freshVar(counter)
val (counter__, x1) = freshVar(counter_)
val (counter___, x2) = freshVar(counter__)
```

```
defined function freshVar
counter: Int = 0
counter_: Int = 1
x0: String = "x0"
counter__: Int = 2
x1: String = "x1"
counter___: Int = 3
x2: String = "x2"
```

We see the `Int` as the input-out state where a call to `freshVar` "updates" the state. Here, the "state" is the `Int` representing the next available variable number. The contract of the freshVar function is that counter on input is the next available variable number to return the next variable name s"x${counter}". It also returns counter + 1 that is the next available variable number, conceptually "allocating" the next variable number.

In the imperative version freshVarImperative, we have to be careful about what code mutates counter. However, in the functional version freshVar, we have to be careful to thread the right version of counter.

We can improve this slightly with careful use of shadowing counter:

```
val counter = 0
val (x0, x1, x2) = freshVar(counter) match {
  case (counter, x0) => freshVar(counter) match {
    case (counter, x1) => freshVar(counter) match {
      case (counter, x2) => (x0, x1, x2)
    }
  }
}
```

```
counter: Int = 0
x0: String = "x0"
x1: String = "x1"
x2: String = "x2"
```

However, it is still arguably messy.

30.5 Encapsulating Mutation Effects

When there is repeated boilerplate that would be error-prone to get right each time, good engineers will implement a library so that they can type less boilerplate and more importantly never get it wrong.

Thus, a seemingly crazy idea is to ask, "Can we abstract a generic state-transforming function S => (S, A) as a collection-like data type?" Let us call this data type a DoWith[S, A]:

```
type DoWith[S, A] = S => (S, A)
```

```
defined type DoWith
```

which is a function that returns a result of type `A` while computing "with" a state type `S`.

Observe that `freshVar` is a function of type `DoWith[Int, String]`:

```
freshVar: DoWith[Int, String]
```

```
res27: Int => (Int, String) = ammonite.$sess.cmd24$Helper$$Lambda$2351/0x0000000800bd3840@27
```

Let us see a `DoWith[S, A]` like a data type that stores a way to compute an `A` using an input-output state of type `S`. We see it is like a collection similar to `Option[A]`, `List[A]`, or `Either[Err, A]` in that we can define `map`:

```
def map[S, A, B](doer: DoWith[S, A])(f: A => B): DoWith[S, B] = { (s: S) =>
  val (s_, a) = doer(s)
  (s_, f(a))
}
```

```
defined function map
```

This `map` function transforms a `DoWith[S, A]` to a `DoWith[S, B]` using a callback function `f: A => B`. We see that the implementation of this generic function is to create a function that when called with an `s: S`, applies `doer` to get an updated state `s_` and an `a: A` value to then call `f(a)` to get a value of type `B` to return with the updated state `s_`. This function implements that careful threading of state that we saw above with `counter` and `freshVar`.

And we can similarly implement a `flatMap`:

```
def flatMap[S, A, B](doer: DoWith[S, A])(f: A => DoWith[S, B]): DoWith[S, B] = { (s: S) =>
  val (s_, a) = doer(s)
  f(a)(s_)
}
```

```
defined function flatMap
```

that carefully threads the state values `s` and `s_` of type `S`.

We can then use `flatMap` and `map` to create a function that threads the state of `counter` through calls to `freshVar` that is then called with the initial counter-state of `0`.

```
val counter = 0
val (counter___, (x0, x1, x2)) =
  (flatMap(freshVar) { x0 =>
    flatMap(freshVar) { x1 =>
      map(freshVar) { x2 =>
        (x0, x1, x2)
      }
    }
  })(counter)
```

```
counter: Int = 0
counter___: Int = 3
x0: String = "x0"
x1: String = "x1"
x2: String = "x2"
```

Let us now implement a library class `DoWith[S, A]` that encapsulates a function of type `S => (S, A)` with `map` and `flatMap` methods following the above:

```
sealed class DoWith[S, A] private (doer: S => (S, A)) {
  def map[B](f: A => B): DoWith[S, B] = new DoWith[S, B]({
    (s: S) => {
      val (s_, a) = doer(s)
      (s_, f(a))
    }
  })

  def flatMap[B](f: A => DoWith[S, B]): DoWith[S, B] = new DoWith[S, B]({
    (s: S) => {
      val (s_, a) = doer(s)
      f(a)(s_)
    }
  })

  def apply(s: S): (S, A) = doer(s)
}

object DoWith {
  def doget[S]: DoWith[S, S] = new DoWith[S, S]({ s => (s, s) })
  def doput[S](s: S): DoWith[S, Unit] = new DoWith[S, Unit]({ _ => (s, ()) })
  def doreturn[S, A](a: A): DoWith[S, A] = new DoWith[S, A]({ s => (s, a) })
```

```
  def domodify[S](f: S => S): DoWith[S, Unit] = new DoWith[S, Unit]({ s => (f(s), ()) })
}

import DoWith._
```

```
defined class DoWith
defined object DoWith
import DoWith._
```

In the above definition of the `DoWith[S, A]` library class, we encapsulate the state-transforming function `doer: S => (S, A)`. We go one step further in preventing implementation errors by requiring the client create `DoWith[S, A]` objects using only `doget`, `doput`, `doreturn`, `domodify`, `map`, and `flatMap`. That is, the client cannot directly construct `DoWith[S, A]` objects with `new` because the constructor is marked `private` but instead has to use one of those six methods.

The `doget[S]` method constructs a `DoWith[S, S]` that makes the "current" state the result (i.e., `s => (s, s)`). Intuitively, it "gets" the state.

The `doput[S](s: S)` method constructs a `DoWith[S, Unit]` that makes the given state `s` the "current" state (i.e., `_ => (s, ())`). Intuitively, it "puts" `s` into the state.

The `doreturn[S, A](a: A)` method constructs a `DoWith[S, A]` that leaves the "current" state as-is and returns the given result `a` (i.e., `s => (s, a)`). It technically does not need to be given in the library, as it can be defined in terms of `doget` and `map`.

The `domodify[S](f: S => S)` method constructs a `DoWith[S, Unit]` that "modifies" the state using the given function `f: S => S` (i.e., `s => (f(s), ())`). It technically does not need to be given in the library, as it can be defined in terms of `doget`, `doput`, and `flatMap`.

Let us now define `freshVar` as a `DoWith[Int, String]`:

```
def freshVar: DoWith[Int, String] = doget flatMap { counter =>
  doput(counter + 1) map { _ => s"x${counter}" }
}

freshVar(0)
```

```
defined function freshVar
res32_1: (Int, String) = (1, "x0")
```

Or we can use `for-yield` expressions:

```
def freshVar: DoWith[Int, String] =
  for {
    counter <- doget
    _ <- doput(counter + 1)
  } yield s"x${counter}"

freshVar(0)
```

```
defined function freshVar
res33_1: (Int, String) = (1, "x0")
```

And we can get our three fresh variables:

```
val counter = 0
val (counter___, (x0, x1, x2)) =
  (freshVar flatMap { x0 =>
    freshVar flatMap { x1 =>
      freshVar map { x2 =>
        (x0, x1, x2)
      }
    }
  })(counter)
```

```
counter: Int = 0
counter___: Int = 3
x0: String = "x0"
x1: String = "x1"
x2: String = "x2"
```

```
val counter = 0
val (counter___, (x0, x1, x2)) =
  (for {
    x0 <- freshVar
    x1 <- freshVar
    x2 <- freshVar
  } yield (x0, x1, x2))(counter)
```

```
counter: Int = 0
counter___: Int = 3
x0: String = "x0"
x1: String = "x1"
x2: String = "x2"
```

Note that DoWith[S, A] is often called State[S, A] (e.g., in the Scala Cats library).

30.6 Monads

Data types Option[A], Either[Err,A], Try[A], List[A], and DoWith[S, A] are similar in that they all have a flatMap method. Having a flatMap method corresponds to being able to sequentially compose them:

```
def getOption: Option[Int] = Some(1)
def getEither: Either[String, Int] = Right(2)
def getTry: Try[Int] = Try(3)
def getList: List[Int] = List(4, 5)
def getDoWith: DoWith[String, Int] = doreturn(6)

for { i1 <- getOption; i2 <- getOption } yield (i1, i2)
for { i1 <- getEither; i2 <- getEither } yield (i1, i2)
for { i1 <- getTry; i2 <- getTry } yield (i1, i2)
for { i1 <- getList; i2 <- getList } yield (i1, i2)

val doer = for { i1 <- getDoWith; i2 <- getDoWith } yield (i1, i2)
doer("")
```

```
defined function getOption
defined function getEither
defined function getTry
defined function getList
defined function getDoWith
res36_5: Option[(Int, Int)] = Some(value = (1, 1))
res36_6: Either[String, (Int, Int)] = Right(value = (2, 2))
res36_7: Try[(Int, Int)] = Success(value = (3, 3))
res36_8: List[(Int, Int)] = List((4, 4), (4, 5), (5, 4), (5, 5))
doer: DoWith[String, (Int, Int)] = ammonite.$sess.cmd31$Helper$DoWith@2cbfa83f
res36_10: (String, (Int, Int)) = ("", (6, 6))
```

A type constructor M for a parametrized data type M[A] that has a flatMap method, as well as method to construct a M[A] from an A is called a *monad*. We see that Option[_], Either[Err,_], Try[_], List[_], and DoWith[S,_] are monads where we write _ for the parametrized type.

Unfortunately, there are lots of confusing descriptions of monads out there. For our purposes, the essence is simply observing that it is a design pattern for data types. Defining a flatMap method

```scala
class M[A] {
  def flatMap[B](f: A => M[B]): M[B] = ???
}
```

```
defined class M
```

makes it possible to sequentially compose computations using that data type.

30.6.1 Monad Interface

It is possible to take this one step further in defining an interface for a type constructor that has a monad interface:

```scala
trait Monad[M[_]] {
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def pure[A](a: A): M[A]
}
```

```
defined trait Monad
```

The `M[_]` parameter to `Monad` says that `M` is a type constructor with one parameter.

The `pure[A]` method injects an `A` into an `M[A]` (e.g., `Some`, `Right`, `Try`, `List`, or `doreturn`).

The following objects witness that `Option` and `List` satisfy the `Monad` interface:

```scala
object optionMonad extends Monad[Option] {
  def flatMap[A, B](opt: Option[A])(f: A => Option[B]): Option[B] = opt.flatMap(f)
  def pure[A](a: A): Option[A] = Some(a)
}

object listMonad extends Monad[List] {
  def flatMap[A, B](l: List[A])(f: A => List[B]): List[B] = l.flatMap(f)
  def pure[A](a: A): List[A] = List(a)
}
```

```
defined object optionMonad
defined object listMonad
```

We can now define functions that are generic over type constructors that satisfy the `Monad` interface:

```
def cross[M[_], A, B](ma: M[A], mb: M[B])(m: Monad[M]): M[(A, B)] =
  m.flatMap(ma) { a => m.flatMap(mb) { b => m.pure(a, b) } }

cross[Option, Int, String](Some(1), Some("hello"))(optionMonad)
cross[List, Int, String](List(2, 3), List("hola", "bonjour"))(listMonad)
```

```
defined function cross
res40_1: Option[(Int, String)] = Some(value = (1, "hello"))
res40_2: List[(Int, String)] = List(
  (2, "hola"),
  (2, "bonjour"),
  (3, "hola"),
  (3, "bonjour")
)
```

For a type constructor to be a proper monad, the `flatMap` and `pure` should be satisfy some expected consistency conditions (cf. monad laws). In particular, we see that `pure` is a kind of a no-op, so we should be able to remove it in a `flatMap` sequence without changing the result. And since `flatMap` as a kind of sequencing operator, so we should be able to change the grouping of the sequence without changing the result.

In other contexts, `flatMap` is sometimes called "bind" or written as the `>>=` operator, and `pure` is sometimes called "return".

30.6.2 Contextual Abstraction

It seems somewhat onerous to explicitly pass the `optionMonad` or `listMonad` instances to `cross` when the type signature already says `Option` or `List`:

```
cross[Option, Int, String](Some(1), Some("hello"))(optionMonad)
cross[List, Int, String](List(2, 3), List("hola", "bonjour"))(listMonad)
```

```
res41_0: Option[(Int, String)] = Some(value = (1, "hello"))
res41_1: List[(Int, String)] = List(
  (2, "hola"),
  (2, "bonjour"),
  (3, "hola"),
  (3, "bonjour")
)
```

Scala does have an advanced feature to automatically pass particular given values of particular types (cf. contextual parameters). In particular, we expect `optionMonad` and `listMonad` to be the only instances of `Monad[Option]` and `Monad[List]` that would make sense, respectively. With contextual parameters, we can instruct the Scala compiler to pass either `optionMonad` or `listMonad` whenever an instance of type `Monad[Option]` or `Monad[List]` is needed, respectively:

```scala
trait Monad[M[_]] {
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def pure[A](a: A): M[A]
}

implicit object optionMonad extends Monad[Option] {
  def flatMap[A, B](opt: Option[A])(f: A => Option[B]): Option[B] = opt.flatMap(f)
  def pure[A](a: A): Option[A] = Some(a)
}

implicit object listMonad extends Monad[List] {
  def flatMap[A, B](l: List[A])(f: A => List[B]): List[B] = l.flatMap(f)
  def pure[A](a: A): List[A] = List(a)
}
```

```
defined trait Monad
defined object optionMonad
defined object listMonad
```

The **`implicit`** keyword states that `optionMonad` and `listMonad` are these canonical instances of type `Monad[Option]` and `Monad[List]`, respectively.

We then state that `cross` takes in a parameter of type `Monad[M]` implicitly:

```scala
def cross[M[_]: Monad, A, B](ma: M[A], mb: M[B]): M[(A, B)] = {
  val m = implicitly[Monad[M]]
  m.flatMap(ma) { a => m.flatMap(mb) { b => m.pure(a, b) } }
}
```

```
defined function cross
```

The `M[_]: Monad` declaration says `M` has a canonical `Monad[M]` instance that should be passed as an argument to `cross`. The method call to `implicitly[Monad[M]]` gets that implicit parameter (though it is also possible to explicitly declare `m` as an implicit parameter of `cross`).

The result is that we can call `cross` without explicitly passing the `optionMonad` or `listMonad` instances:

```
cross[Option, Int, String](Some(1), Some("hello"))
cross[List, Int, String](List(2, 3), List("hola", "bonjour"))
```

```
res44_0: Option[(Int, String)] = Some(value = (1, "hello"))
res44_1: List[(Int, String)] = List(
  (2, "hola"),
  (2, "bonjour"),
  (3, "hola"),
  (3, "bonjour")
)
```

Note that the **`implicit`** keyword is, unfortunately, overloaded for many things in Scala 2. This language feature has been significantly revised and improved on in Scala 3.

31 Exercise: Programming with Encapsulated Effects

Learning Goals

The primary learning goal of this exercise is to get experience programming with encapsulated effects.

We will also consider the idea of transforming code represented as an abstract syntax tree to make it easier to implement subsequent passes like interpretation.

Instructions

This assignment asks you to write Scala code. There are restrictions associated with how you can solve these problems. Please pay careful heed to those. If you are unsure, ask the course staff.

Note that `???` indicates that there is a missing function or code fragment that needs to be filled in. Make sure that you remove the `???` and replace it with the answer.

Use the test cases provided to test your implementations. You are also encouraged to write your own test cases to help debug your work. However, please delete any extra cells you may have created lest they break an autograder.

Imports

```
import $ivy.$                                   , org.scalatest._, events._, flatspec._, matche

import $ivy.$                                                 , org.scalatestplus.scalacheck._
```

```
defined function report
defined function assertPassed
defined function passed
defined function test
```

Listing 31.1 `org.scalatest._`

```
// Run this cell FIRST before testing.
import $ivy.`org.scalatest::scalatest:3.2.19`, org.scalatest._, events._, flatspec._, matche
import $ivy.`org.scalatestplus::scalacheck-1-18:3.2.19.0`, org.scalatestplus.scalacheck._
def report(suite: Suite): Unit = suite.execute(stats = true)
def assertPassed(suite: Suite): Unit =
  suite.run(None, Args(new Reporter {
    def apply(e: Event) = e match {
      case e @ (_: TestFailed) => assert(false, s"${e.message} (${e.testName})")
      case _ => ()
    }
  }))
def passed(points: Int): Unit = {
  require(points >= 0)
  if (points == 1) println("***   Tests Passed (1 point) ***")
  else println(s"***   Tests Passed ($points points) ***")
}
def test(suite: Suite, points: Int): Unit = {
  report(suite)
  assertPassed(suite)
  passed(points)
}
```

31.1 TypeScripty: Numbers, Booleans, and Functions

31.1.1 Syntax

In this assignment, we consider a simplified TypeScripty with numbers, booleans, and functions types.

Function literals $x(y\colon\tau)\colon\tau' \Rightarrow e_1$ have exactly one parameter, are always named, and must have a return type annotation. The name may be used to define recursive functions.

The expressions include variable uses x, variable binding **const** x = e_1; e_2, unary $uop\, e_1$ and binary e_1 bop e_2 expressions, if-then-else e_1 ? e_2 : e_3, and function call $e_1(e_2)$. The unary operators are limited to number negation `-` and boolean negation `!`, and the binary operators are limited to number addition `+`, number times `*`, and equality `===`:

We give a Scala representation of the abstract syntax, along with an implementation of computing the free variables of an expression and determining an expression is closed:

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & \texttt{number} \mid \texttt{bool} \mid \texttt{(}y\texttt{:}\,\tau\texttt{)} \Rightarrow \tau' \\
\text{values} & v & ::= & n \mid b \mid x\texttt{(}y\texttt{:}\,\tau\texttt{)}\texttt{:}\,\tau' \Rightarrow e_1 \\
\text{expressions} & e & ::= & x \mid \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 \\
 & & \mid & n \mid uop\, e_1 \mid e_1\ bop\ e_2 \\
 & & \mid & b \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \\
 & & \mid & x\texttt{(}y\texttt{:}\,\tau\texttt{)}\texttt{:}\,\tau' \Rightarrow e_1 \mid e_1\texttt{(}e_2\texttt{)} \\
\text{unary operators} & uop & ::= & \texttt{-} \mid \texttt{!} \\
\text{binary operators} & bop & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{===}
\end{array}
$$

Figure 31.1: Syntax of TypeScripty with recursive functions and limited arithmetic-logical expressions.

```
trait Typ                                                 // t
case object TNumber extends Typ                           // t ::= number
case object TBool extends Typ                             // t ::= bool
case class TFun(yt: (String,Typ), tret: Typ) extends Typ // t ::= (y: t) => tret

trait Expr                                                        // e
case class Var(x: String) extends Expr                            // e ::= x
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr // e ::= const x = e1; e2

case class N(n: Double) extends Expr                         // e ::= n
case class Unary(uop: Uop, e1: Expr) extends Expr            // e ::= uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr // e ::= e1 bop b2

case class B(b: Boolean) extends Expr                     // e ::= b
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr // e ::= e1 ? e2 : e3

case class Fun(x: String, yt: (String, Typ), tret: Typ, e1: Expr) extends Expr // e ::= x(y:
case class Call(e1: Expr, e2: Expr) extends Expr                               // e ::= e1(e

trait Uop                   // uop
case object Neg extends Uop // uop ::= -
case object Not extends Uop // uop ::= !

trait Bop                     // bop
case object Plus extends Bop  // bop ::= *
case object Times extends Bop // bop ::= *
case object Eq extends Bop    // bop ::= ===
```

```scala
def freeVars(e: Expr): Set[String] = e match {
  case Var(x) => Set(x)
  case ConstDecl(x, e1, e2) => freeVars(e1) | (freeVars(e2) - x)
  case N(_) | B(_) => Set.empty
  case Unary(_, e1) => freeVars(e1)
  case Binary(_, e1, e2) => freeVars(e1) | freeVars(e2)
  case If(e1, e2, e3) => freeVars(e1) | freeVars(e2) | freeVars(e3)
  case Fun(x, (y, _), _, e1) => freeVars(e1) - x - y
  case Call(e1, e2) => freeVars(e1) | freeVars(e2)
}

def isClosed(e: Expr): Boolean = freeVars(e).isEmpty
```

```
defined trait Typ
defined object TNumber
defined object TBool
defined class TFun
defined trait Expr
defined class Var
defined class ConstDecl
defined class N
defined class Unary
defined class Binary
defined class B
defined class If
defined class Fun
defined class Call
defined trait Uop
defined object Neg
defined object Not
defined trait Bop
defined object Plus
defined object Times
defined object Eq
defined function freeVars
defined function isClosed
```

31.1.2 Static Type Checking

The typing judgment $\Gamma \vdash e : \tau$ says, "In typing environment Γ, expression e has type τ" where the typing environment $\Gamma ::= \cdot \mid \Gamma, x : \tau$ is a finite map from variables to types, assigning types to the free variables of e. We give the expected typing rules that we have seen before, restricted to this simpler language:

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

$$\textsc{TypeVar}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{TypeConstDecl}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{const}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 : \tau_2}$$

$$\textsc{TypeNumber}\quad \frac{}{\Gamma \vdash n : \texttt{number}}$$

$$\textsc{TypeNeg}\quad \frac{\Gamma \vdash e_1 : \texttt{number}}{\Gamma \vdash \texttt{-}\, e_1 : \texttt{number}}$$

$$\textsc{TypeArith}\quad \frac{\Gamma \vdash e_1 : \texttt{number} \qquad \Gamma \vdash e_2 : \texttt{number} \qquad \mathit{bop} \in \{\texttt{+}, \texttt{*}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{number}}$$

$$\textsc{TypeBool}\quad \frac{}{\Gamma \vdash b : \texttt{bool}}$$

$$\textsc{TypeEq}\quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ \texttt{===}\ e_2 : \texttt{bool}}$$

$$\textsc{TypeIf}\quad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}$$

$$\textsc{TypeFunctionRec}\quad \frac{\Gamma, x : \texttt{(}y\texttt{:}\ \tau\texttt{)} \Rightarrow \tau', y : \tau \vdash e' : \tau'}{\Gamma \vdash x\texttt{(}y\texttt{:}\ \tau\texttt{)}\texttt{:}\ \tau' \Rightarrow e' : \texttt{(}y\texttt{:}\ \tau\texttt{)} \Rightarrow \tau'}$$

$$\textsc{TypeCall}\quad \frac{\Gamma \vdash e_1 : \texttt{(}y\texttt{:}\ \tau\texttt{)} \Rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau'}$$

Figure 31.2: Typing of TypeScripty with recursive functions and limited arithmetic-logical expressions.

31.2 Error Effects

As we have seen, the most direct way to implement a type checker following the typing judgment $\Gamma \vdash e : \tau$ is a function `hastype: (Map[String, Typ], Expr) => Typ` that takes an typing environment Γ represented by a `Map[String, Typ]` and an expression e represented by a `Expr` and returns a type τ represented by a `Typ`:

```
def hastype(tenv: Map[String, Typ], e: Expr): Typ = ???
```

However, with this function signature for `hastype`, we necessarily have to throw an exception or crash to indicate that an expression e is ill-typed. An expression is ill-typed when there are no rules that allow us to derive a judgment $\Gamma \vdash e : \tau$ for any τ and Γ.

In this exercise, we implement a version of `hastype` that makes explicit the possibility of a type error.

31.2.1 Type-Error Result

We first extend our typing judgment form $\Gamma \vdash e : r$ where a type-checker result r is either a type τ or a type-error result *typerr*:

type-checker result $\quad r ::= \mathit{typerr} \mid \tau \qquad$ type-error result $\quad \mathit{typerr} ::= \mathsf{typerr}(e : \tau)$

This extended judgment form makes explicit when an expression is ill-typed. A type-error result

$$\mathsf{typerr}(e : \tau)$$

includes an expression e with its inferred type τ but is used in a context another type is expected. We can consider such a type-error result being used to give a descriptive error message to the programmer.

We now give some rules that introduce type-error results and propagates them:

$$\boxed{\Gamma \vdash e : r} \qquad \textsc{TypeErrorNeg}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \neq \texttt{number}}{\Gamma \vdash \texttt{-}\, e_1 : \mathsf{typerr}(e_1 : \tau_1)} \qquad \textsc{PropagateNeg}\ \frac{\Gamma \vdash e_1 : \mathit{typerr}}{\Gamma \vdash \texttt{-}\, e_1 : \mathit{typerr}}$$

$$\textsc{TypeErrorEq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \neq \tau_2}{\Gamma \vdash e_1 \texttt{ === } e_2 : \mathsf{typerr}(e_2 : \tau_2)} \qquad \textsc{PropagateEq1}\ \frac{\Gamma \vdash e_1 : \mathit{typerr}}{\Gamma \vdash e_1 \texttt{ === } e_2 : \mathit{typerr}} \qquad \textsc{PropagateEq2}\ \frac{\Gamma \vdash e_2 : \mathit{typerr}}{\Gamma \vdash e_1 \texttt{ === } e_2 : \mathit{typerr}}$$

Observe that the TypeErrorEq blames e_2: it infers the type τ_1 for e_1 and then expects that e_2 has type τ_1.

Exercise 31.1 (4 points)**.** Define the TypeErrorNot rule that introduces a type-error result when the boolean negation expression $\texttt{!}\, e_1$ is ill-typed:

Edit this cell:

???

Notes

- Hint: Use TypeErrorNeg as a model.
- You may give the rule in LaTeX math or as plain text (ascii art) approximating the math rendering. For example,

```
TypeErrorNeg
Gamma |- e1 : tau1  tau1 != number
-----------------------------------------
Gamma |- -e1 : typerr(e1 : tau1 != number)
```

The LaTeX code for the rendered TypeErrorNeg rule above is as follows:

```
\inferrule[TypeErrorNeg]{
  \Gamma \vdash e_1 : \tau_1
  \and
  \tau_1 \neq \texttt{number}
}{
  \Gamma \vdash \mathop{\texttt{-}} e_1 :  \mathop{\mathsf{typerr}}(e_1 : \tau_1 \neq \textt
}
```

Exercise 31.2 (8 points)**.** Define **two** TypeErrorIf rules that introduces a type-error result when the if-then-else expression e_1 ? e_2 : e_3 is ill-typed:

Edit this cell:

???

Notes

- Hint: Use TypeErrorNeg and TypeErrorEq as a model for the two rules, respectively.

31.2.2 Implementation

We represent a type-error result r as an `Either[StaticTypeError, Typ]`.

Exercise 31.3 (26 points)**.** Complete the following implementation of

`hastype: (Map[String, Typ], Expr) => Either[StaticTypeError, Typ]`

corresponding to the judgment form $\Gamma \vdash e : r$.

Edit this cell:

```
case class StaticTypeError(tbad: Typ, esub: Expr, e: Expr) {
  override def toString = s"StaticTypeError: invalid type $tbad for sub-expression $esub in $
}

def hastype(tenv: Map[String, Typ], e: Expr): Either[StaticTypeError, Typ] = {

  def err[T](esub: Expr, tgot: Typ): Either[StaticTypeError, T] =
    Left(StaticTypeError(tgot, esub, e))

  def typecheck(tenv: Map[String, Typ], e: Expr, tshould: Typ): Either[StaticTypeError, Unit]
    hastype(tenv, e) flatMap { tgot =>
      if (tgot == tshould)
        ???
      else err(e, tgot)
    }

  e match {
    case Var(x) => Right(tenv(x))
    case ConstDecl(x, e1, e2) =>
      ???
    case N(_) => Right(TNumber)
    case B(_) => Right(TBool)
    case Unary(Neg, e1) => typecheck(tenv, e1, TNumber) map { _ => TNumber }
    case Unary(Not, e1) =>
      ???
    case Binary(Plus|Times, e1, e2) =>
      ???
    case Binary(Eq, e1, e2) =>
      hastype(tenv, e1) flatMap { t1 =>
        hastype(tenv, e2) flatMap { t2 =>
          if (t1 == t2) Right(TBool) else err(e2, t2)
        }
      }
    case If(e1, e2, e3) => typecheck(tenv, e1, TBool) flatMap { _ =>
      ???
    }
    case Fun(x, yt @ (y, t), tret, e1) => {
      ???
    }
    case Call(e1, e2) =>
      ???
```

```
  }
}

def inferType(e: Expr): Either[StaticTypeError, Typ] = {
  require(isClosed(e), s"$e should be closed")
  hastype(Map.empty, e)
}
```

```
defined class StaticTypeError
defined function hastype
defined function inferType
```

Notes

- The `err` helper function is simply a shortcut to construct `StaticTypeError` with the input expression `e`.
- The `typecheck` helper function implements a common functionality where we want to call `hastype` recursively with a sub-expression to infer a type `tgot` and check it is equal to an expected type `tshould`. If `tgot != tshould`, we return an `err`; otherwise, we return success.
- Hint: Start by implementing the Type rules from Figure 31.2 without worrying about type errors. Then, add TypeError and Propagate cases.
- Note that not all the TypeError and Propagate rules are given, but they follow the patterns given above. If you get stuck to implement the code for TypeError for a particular construct, try writing out the rule.
- It is fine to initially pattern match on `Either[StaticTypeError, Expr]` values that result from calls to `hastype` and `typecheck` (i.e., for `Left` and `Right` values). However, challenge yourself to replace them with `map` or `flatMap` calls to minimize your typing and opportunities for bugs to creep in. It is possible to replace all pattern matches for `Left` and `Right` values with calls to `map` and `flatMap` calls.
- Use the given cases (e.g., TypeNeg, TypeErrorNeg, PropagateNeg) as a model. Identify how TypeNeg, TypeErrorNeg, and PropagateNeg manifests in the code.
- For accelerated students, you can make the code even more compact (and arguably more readable) by replacing `flatMap` and `map` calls with `for-yield` expressions. Note that not all cases can be implemented with `for-yield` expressions.

Tests

31.3 Mutation Effects

31.3.1 Defining Generic `DoWith` Methods

Recall the idea of encapulating a state transforming function `S => (S, A)` as a collection-like data type (Section 30.5), which we call a `DoWith[S, A]`:

```
sealed class DoWith[S, A] private (doer: S => (S, A)) {
  def map[B](f: A => B): DoWith[S, B] = new DoWith[S, B]({
    (s: S) => {
      val (s_, a) = doer(s)
      (s_, f(a))
    }
  })

  def flatMap[B](f: A => DoWith[S, B]): DoWith[S, B] = new DoWith[S, B]({
    (s: S) => {
      val (s_, a) = doer(s)
      f(a)(s_)
    }
  })

  def apply(s: S): (S, A) = doer(s)
}

object DoWith {
  def doget[S]: DoWith[S, S] = new DoWith[S, S]({ s => (s, s) })
  def doput[S](s: S): DoWith[S, Unit] = new DoWith[S, Unit]({ _ => (s, ()) })
}

import DoWith._
```

```
defined class DoWith
defined object DoWith
import DoWith._
```

Consider a `DoWith[S,A]` as a "collection" holding an `A` somewhat like `List[A]` or `Option[A]`. While `List[A]` encapsulates a sequence of elements of type `A` elements and `Option[A]` encapsulates an optional `A`, a `DoWith[S, A]` encapsulates a *computation* that results in an `A`, using an input-output state `S` (i.e., a `S => (S, A)` function). It is collection-like because it also has `map` and `flatMap` methods to transform that computation.

We also define two functions for a client to create a `DoWith[S, A]` encapsulating particular `S => (S, A)`{.scala functions}:

- The `doget[S]` method constructs a `DoWith[S, S]` that makes the "current" state the result (i.e., `s => (s, s)`). Intuitively, it "gets" the state.
- The `doput[S](s: S)` method constructs a `DoWith[S, Unit]` that makes the given state `s` the "current" state (i.e., `_ => (s, ())`). Intuitively, it "puts" `s` into the state.

We require the client to create `DoWith[S, A]` values using only the `doget` and `doput` constructors.

Exercise 31.4 (4 points)**.** Read the implementations `map[B]` and `flatMap[B]` methods of `DoWith[S, A]` above. Explain how they transform the encapsulated `doer: S => (S, A)` function. Compare and contrast them—what do they do that is same, and what is the key difference between them?

Edit this cell:

???

As a `DoWith[S, A]` encapsulates any function of type `S => (S, A)`, there are other commonly needed functions of this type. We implement methods for constructing `DoWith[S, A]` encapsulating two more commonly-needed computations in terms of `doget` and `doput`.

- The `doreturn[S, A](a: A)` method constructs a `DoWith[S, A]` that leaves the "current" state as-is and returns the given result `a` (i.e., `s => (s, a)`). It technically does not need to be given in the library, as it can be defined in terms of `doget` and `map`.
- The `domodify[S](f: S => S)` method constructs a `DoWith[S, Unit]` that "modifies" the state using the given function `f: S => S` (i.e., `s => (f(s), ())`). It technically does not need to be given in the library, as it can be defined in terms of `doget`, `doput`, and `flatMap`.

Exercise 31.5 (4 points)**.** Implement `doreturn[S, A](a: A)` that creates a `DoWith[S, A]` that encapsulates the computation `(s: S) => (s, a)` using `doget`, `doput`, `map`, and/or `flatMap`.

Edit this cell:

```scala
def doreturn[S, A](a: A): DoWith[S, A] =
  ???
```

```
defined function doreturn
```

Tests

Exercise 31.6 (4 points)**.** Implement `domodify[S](f: S => S)` that creates a `DoWith[S, Unit]` that encapsulates the computation `(s: S) => (f(s), ())` using `doget`, `doput`, `map`, and/or `flatMap`.

Edit this cell:

```
def domodify[S](f: S => S): DoWith[S, Unit] =
  ???
```

```
defined function domodify
```

Notes

- Hint: To define `doreturn` and `domodify`, it might help first to do type-directed programming where you ignore what a `DoWith[S, A]` actually is and think of it as `Either[S, A]` or abstractly as `M[S, A]` for an unknown type constructor `M`. There only limited things you can do by composing `doget`, `doput`, `map`, and `flatMap`.

Tests

31.3.2 Renaming Bound Variables

Recall that with static scoping, we see expressions as being equivalent up to the renaming of bound variables (Section 14.7). For example, we see the following two expressions as being equivalent for the purposes of the language implementation:

```
const four = (2 + 2); (four + four)
```

```
const x = (2 + 2); (x + x)
```

even though the concrete syntax for the human user is different in their choice of variable names.

While being able to choose variable names that shadows another variable is important for the human user, we have seen that clashing variable names adds complexity to the language implementation. For example, mishandling of clashing variable names results in accidental dynamic scoping (Section 19.2) and substitution has to account for variable shadowing (Section 21.9).

Observing that renaming bound variables preserves meaning for the language implementation, one approach for simplifying the handling of variable scope is to implement a *lowering* pass that renames bound variables in a consistent manner so that variable names are globally unique. For example, we could lower the following expression

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

by renaming bound variables to, for example,

```
const a_0 = ((a_1) => a_1)(1) + ((a_2) => a_2)(2); a_0
```

We will implement the above lowering pass in a few steps.

Exercise 31.7 (4 points)**.** Write a function `freshVar(x: String)` that returns a function of type `Int => (Int, String)`, which takes a current index-state `i: Int` and returns the pair of the next index state `i + 1` and the string `x` with the index appended and separated by `"_"` (specifically, `s"${x}_${i}"`):

Edit this cell:

```
def freshVar(x: String): Int => (Int, String) =
  { i =>
      ???
  }
```

```
defined function freshVar
```

Tests

Aside: Writing lots of tests is painful. In the above, we use an idea called *property-based testing* to make it less painful and more effective. A property-based testing library enables specifying a property on inputs like in the above for `x: String` and `i: Int`, and the library will choose lots of random inputs with which to check the property. Importantly, the library also enables the client to specify how to generate inputs (e.g., what range, what distribution), though we do not do anything special to control the input generation in the above.

Exercise 31.8 (4 points)**.** Write a function `freshVarWith(x: String): DoWith[Int, String]` that behaves like `freshVar(x: String): Int => (Int, String)` in Exercise 31.7.

Edit this cell:

```scala
def freshVarWith(x: String): DoWith[Int, String] =
  ???
```

```
defined function freshVarWith
```

Notes

- Hint: Remember that `DoWith[Int, String]` encapsulates a function of type `Int => (Int, String)`. Use `freshVar` as a guide to creating the `DoWith[Int, String]` using `doget`, `doput`, `map`, and/or `flatMap`.
- For accelerated students, you may try to use a `for-yield` expression to replace your `map` and `flatMap` calls.

A lowering pass is a transformation function from `Expr => Expr`. To rename bound variables in a consistent manner, we need an enviroment to remember what the name should be for a free variable use in the input `Expr`. Thus, we need to define a helper function `rename(env: Map[String, String], e: Expr)` that takes as input an `env: Map[String, String]` that maps original names to new names for the free variable uses in `e`.

We also need a way to choose how to rename bound variables so that they are unique. The client of `rename` might want to rename variables uniquely in different ways (e.g., using an integer counter, using the original name with an integer counter). Thus, we add an additional callback parameter

```scala
fresh: String => DoWith[S, String]
```

for the client to specify how they want to specify the fresh name given the original name where they can choose a state type `S` to use through renaming.

Tests

Exercise 31.9 (24 points)**.** Implement a function `rename` that renames variable names in `e` consistently using the given callback to `fresh` to generate fresh names for bound variables and `env` to rename free variable uses.

Edit this cell:

Notes

- Hint: Use the given cases for `N` and `Call` as models. The only methods for manipulating `DoWith` objects needed in this exercise are `doreturn`, `map`, and `flatMap`.
- **Accelerated**: You may try to use `for-yield` expressions to replace your `map` and `flatMap` calls.

31.3.3 Test

Exercise 31.10 (4 points)**.** Finally, implement the lowering-transformation function `uniquify: Expr => Expr` on closed expressions using a call to `rename` to rename bound variables consistently with the `freshVarWith: String => DoWith[Int, String]` function defined above. Start the `Int` counter-state at `0`.

Edit this cell:

Notes

- Hint: The `rename` and `freshVarWith` functions constructs `DoWith` objects. The `uniquify` finally uses a `DoWith[Int, String]` object. How? By calling it with the initial state `0`.

Test

31.3.4 `DoWith` with Collections

Recall the higher-order function `map` we considered previously for `List`s.

```
def map[A, B](l: List[A])(f: A => B): List[B] =
  l.foldRight[List[B]](
    Nil
  ) { (h, acc) =>
    f(h) :: acc
  }
```

```
defined function map
```

We may want to use `DoWith` to encapsulate some stateful computation with other data structures, like `List`s. For example, we want a version of `map` that can take a stateful callback:

Exercise 31.11 (4 points)**.** Implement a function `mapWith` for `List` that takes a stateful callback function `f: A => DoWith[S, B]`:

Edit this cell:

```
def mapWith[S, A, B](l: List[A])(f: A => DoWith[S, B]): DoWith[S, List[B]] =
  l.foldRight[DoWith[S, List[B]]](
    ???
  ) { (h, acc) =>
    ???
  }
```

```
defined function mapWith
```

Tests

Also recall the `mapFirst` function we defined previously that replaces the first element in `l` where `f` returns `Some(a)` with `a`:

```
def mapFirst[A](l: List[A])(f: A => Option[A]): List[A] = l match {
  case Nil => Nil
  case h :: t =>
    f(h) match {
      case None => h :: mapFirst(t)(f)
      case Some(h) => h :: t
    }
}
```

```
defined function mapFirst
```

Exercise 31.12 (4 points)**.** Implement a function `mapFirstWith` for `List` that instead takes a stateful callback function `f: A => Option[DoWith[S, B]]`:

Edit this cell:

```
def mapFirstWith[S, A](l: List[A])(f: A => Option[DoWith[S, A]]): DoWith[S, List[A]] = l mat
  case Nil =>
    ???
  case h :: t =>
    ???
}
```

```
defined function mapFirstWith
```

Tests

32 Mutable State

Recall that the most characteristic feature of imperative computation is *mutation*—that is, executing *assignment* for its side *effect* that updates a *memory* (cf. Section 3.1).

32.1 JavaScripty: Mutable Variables

32.1.1 Syntax

To introduce mutable state, we introduce *mutable variables* declared as follows:

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

A **var** declaration creates a new mutable variable and assigns it an initial value. We also introduce an assignment expression:

$$e_1 = e_2$$

that writes the value of e_2 to a memory location named by expression e_1. Note that we use the C and JavaScript-style assignment operator =, which unfortunately looks like mathematical equality but is very different.

Let us consider JavaScripty with number literals and mutable variables:

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & n \mid x \mid \mathbf{var}\ x = e_1;\ e_2 \mid e_1 = e_2 \mid {*}e_1 \mid a \\
\text{variables} & x & & \\
\text{values} & v & ::= & n \\
\text{location values} & l & ::= & {*}a \\
\text{addresses} & a & &
\end{array}
$$

Figure 32.1: Abstract syntax of JavaScripty with number literals and mutable variables.

To focus on mutation, let us drop **const** $x = e_1;\ e_2$ constant-variable declarations for the moment and have only **var** $x = e_1;\ e_2$ mutable-variable declarations. The **var** $x = e_1;\ e_2$

mutable-variable declaration allocates a new memory cell with fresh address a, evaluates e_1 to a value v_1, stores value v_1 in the new memory cell, and evaluates e_2 with x in scope pointing to the new memory cell.

The assignment expression e_1 = e_2 evaluates e_1 to a location value l_1, e_2 to a value v_2, updates the memory cell named by l_1 with value v_2, and returns v_2.

We introduce the unary, pointer-dereference expression $*e_1$ in the abstract syntax to use as an intermediate expression during reduction. It is not present in the concrete syntax of JavaScript, though it corresponds to the pointer-dereference expression in the C and C++ languages.

An address a is a memory address for a memory cell that stores some content like values. A location value l is a reduced expression that names a memory location.

Note that we do not consider an address a to be a value here. In low-level languages like C and C++, an address is called a *pointer* and is a first-class value (i.e., an address is a a is a value). A location value is also called an *l-value* for the value of an expression on the left-hand–side of an assignment, while a value is called a *r-value* for the value of an expression on the right-hand–side of an assignment.

```
trait Expr                                                   // e
case class N(n: Double) extends Expr                         // e ::= n
case class Var(x: String) extends Expr                       // e ::= x
case class VarDecl(x: String, e1: Expr, e2: Expr) extends Expr // e ::= var x = e1; e2
case class Assign(e1: Expr, e2: Expr) extends Expr           // e ::= e1 = e2
case class Deref(e1: Expr) extends Expr                      // e ::= *e1
case class A(a: Int) extends Expr                            // e ::= a

def isValue(e: Expr): Boolean = e match {
  case N(_) => true
  case _ => false
}
```

```
defined trait Expr
defined class N
defined class Var
defined class VarDecl
defined class Assign
defined class Deref
defined class A
defined function isValue
```

We represent an address a in Scala as an `A(a)` for a positive integer `a`. We can think of the address `A(1)` corresponding to the hexadecimal `0x00000004` memory address on a 32-bit machine.

Let consider the example expression with assignment:

Listing 32.1 JavaScript

```
var i = 1;
i = 2
```

Listing 32.2 AST Representation in Scala

```
val e_assign =
  VarDecl("i", N(1),
    Assign(Var("i"), N(2))
  )
```

```
e_assign: VarDecl = VarDecl(
  x = "i",
  e1 = N(n = 1.0),
  e2 = Assign(e1 = Var(x = "i"), e2 = N(n = 2.0))
)
```

32.1.2 Small-Step Operational Semantics

32.1.2.1 Memories

A *mutable variable* is a variable that can be updated. We can think of a mutable variable as a box that can be filled with a value, and then the value can be updated by filling the box with a new value. A memory cell $[a \mapsto v]$ is such a box that has an address a to reference that box. A memory m is then a finite set of such memory cells:

$$\text{memories} \quad m \quad ::= \quad \cdot \mid m[a \mapsto v]$$

Figure 32.2: Memories for JavaScripty with mutable variables.

We also view a memory m as a finite map from addresses a to values v and write $m(a)$ for looking up the value v corresponding to address a in memory m.

32.1.2.2 Location Values

We have noted above that a location value l is a reduced expression that names a memory location. Like e $\textsf{value}$ defining values, we can view this as unary judgment form e $\textsf{location}$ on expressions:

$$\boxed{e\ \textsf{location}} \qquad \frac{\textsc{VarLocation}}{*a\ \textsf{location}}$$

Figure 32.3: Location values for JavaScripty with mutable variables.

In particular, the location value for a variable is given by the expression $*a$ for some address a.

32.1.2.3 Judgment Form for Imperative Computation

To define a small-step operational semantics with mutable variables, we need to update our reduction-step judgment form to include memories:

$$\langle e, m \rangle \longrightarrow \langle e', m' \rangle$$

that says, "Closed expression e with memory m reduces to closed expression e' with updated memory m'."

We can see the *state* of the machine as a pair of the expression e and the memory m:

$$\text{states} \quad \sigma \quad ::= \quad \langle e, m \rangle$$

and thus the small-step judgment form is $\sigma \longrightarrow \sigma'$. This machine state σ with a program e that we execute for its effects on an off-to-the-side memory m is the characteristic feature of imperative computation.

In pure functional computation, the machine state σ is just the program e that we evaluate to a value (i.e., iterate $e \longrightarrow e'$).

32.1.2.4 Inference Rules for Mutation

The DoDeref rule says that dereferencing an address $*a$ reduces to the value v stored in the memory cell named by a:

$$\textsc{DoDeref}\quad \frac{[a \mapsto v] \in m}{\langle *a, m\rangle \longrightarrow \langle v, m\rangle}$$

Note that it would be equivalent to write DoDeref as follows:

$$\textsc{DoDeref}\quad \frac{}{\langle *a, m\rangle \longrightarrow \langle m(a), m\rangle}$$

The DoAssignVar rule says that assigning value v to memory location $*a$ in memory m updates memory m with the cell $[a \mapsto v]$:

$$\textsc{DoAssignVar}\quad \frac{a \in \mathrm{dom}(m)}{\langle *a = v, m\rangle \longrightarrow \langle v, m[a \mapsto v]\rangle}$$

The side-condition that the address a is in the domain of memory m (i.e., $a \in \mathrm{dom}(m)$) says that there is already an allocated memory cell $[a \mapsto v_0]$ in m so that we are writing $m[a \mapsto v]$ to mean updating that cell from v_0 to v in memory m.

We shall see that memory addresses a are introduced by **var** allocation, so the alternative system that does not check this condition in the DoAssignVar rule

$$\textsc{DoAssignVar}\quad \frac{}{\langle *a = v, m\rangle \longrightarrow \langle v, m[a \mapsto v]\rangle}$$

is essentially the same (technically, called being *bisimilar*).

The DoVarDecl rules describes allocating a new memory cell for a mutable variable:

$$\textsc{DoVarDecl}\quad \frac{a \notin \mathrm{dom}(m)}{\langle \mathbf{var}\ x = v_1\,;\ e_2, m\rangle \longrightarrow \langle [*a/x]e_2, m[a \mapsto v_1]\rangle}$$

We choose a fresh address a, which we state with the side condition that $a \notin \mathrm{dom}(m)$ and thus the $[a \mapsto v_1]$ is a new cell in the updated memory $m[a \mapsto v_1]$. The reduced expression $[*a/x]e_2$ is interesting. The scope of variable x is the continuation expression e_2, so we must eliminate free-variable occurrences of x in e_2. We do this by substituting the location value $*a$ corresponding to x in e_2.

In the following, we repeat the above Do rules along with the needed Search for mutable variables:

$$\boxed{\langle e, m\rangle \longrightarrow \langle e', m'\rangle} \qquad \frac{}{\langle *a, m\rangle \longrightarrow \langle m(a), m\rangle}\ \textsc{DoDeref} \qquad \frac{}{\langle *a = v, m\rangle \longrightarrow \langle v, m[a \mapsto v]\rangle}\ \textsc{DoAssignVar}$$

$$\frac{a \notin \operatorname{dom}(m)}{\langle \mathbf{var}\ x = v_1\,;\ e_2, m\rangle \longrightarrow \langle [*a/x]e_2, m[a \mapsto v_1]\rangle}\ \textsc{DoVarDecl} \qquad \frac{\langle e_1, m\rangle \longrightarrow \langle e_1', m'\rangle \qquad e_1 \neq l_1}{\langle e_1 = e_2, m\rangle \longrightarrow \langle e_1' = e_2, m'\rangle}\ \textsc{SearchAssign1}$$

$$\frac{\langle e_2', m\rangle \longrightarrow \langle e_2', m'\rangle}{\langle l_1 = e_2, m\rangle \longrightarrow \langle l_1 = e_2', m'\rangle}\ \textsc{SearchAssign2} \qquad \frac{\langle e_1, m\rangle \longrightarrow \langle e_1', m'\rangle}{\langle \mathbf{var}\ x = e_1\,;\ e_2, m\rangle \longrightarrow \langle \mathbf{var}\ x = e_1'\,;\ e_2, m'\rangle}\ \textsc{SearchVarDecl}$$

Figure 32.4: Small-step operational semantics with mutable variables.

The SearchAssign1 rule says that if e_1 is not a location value, then we need to reduce it to be able to do the assignment:

$$\frac{e_1 \neq l_1 \qquad \langle e_1, m\rangle \longrightarrow \langle e_1', m'\rangle}{\langle e_1 = e_2, m\rangle \longrightarrow \langle e_1' = e_2, m'\rangle}\ \textsc{SearchAssign1}$$

Note that SearchAssign1 rule is needed if addresses were first-class values (cf. pointers in C and C++). However, we see that it is not actually needed for this variant of JavaScripty where addresses are not first-class. In this case, we can also restrict the syntax of assignment to

$$\text{expressions} \quad e \quad ::= \quad x = e_1$$

Studying the DoVarDecl rule, we see that assignment expressions $x = e_1$ where x is in scope would become $*a = e_1$ on substitution where either SearchAssign2 or DoAssignVar would apply.

Indeed, the actual concrete syntax of JavaScript is restricted such that only certain the expression forms like variables can be written on the left-hand–side of assignment.

32.1.3 Implementation

32.1.3.1 Memories

Let us define `Mem` as an abstract data type to represent a memory m (defined in Figure 32.2) in terms of a Scala `Map[A, Expr]`:

```
sealed class Mem private (m: Map[A, Expr], nextAddr: Int) {
  def apply(a: A): Expr = m(a)
  override def toString: String = m.toString

  def +(av: (A, Expr)): Mem = {
    val (a, _) = av
    require(m.contains(a))
    new Mem(m + av, nextAddr)
  }

  def alloc(v: Expr): (A, Mem) = {
    val fresha = A(nextAddr)
    (fresha, new Mem(m + (fresha -> v), nextAddr + 1))
  }
}

object Mem {
  val empty: Mem = new Mem(Map.empty, 1)
}
```

```
defined class Mem
defined object Mem
```

The `apply` and `toString` methods simply delegate to the corresponding methods on the underlying `m: Map[A, Expr]`.

In addition to underlying `m: Map[A, Expr]`, the additional `nextAddr: Int` field represents the next available address. The `+` method for updating the memory checks that the given address to update `a` is already in the map. The `alloc` method implements allocating a fresh address `fresha: A` by using the next available address, extending the map with the new cell `(fresha -> v)` using the given initialization value `v`, and advances the next available address to `nextAddr + 1`.

The abstract data type `Mem` thus maintains a consistency invariant between the map `m: Map[A, Expr]` and the next available address `nextAddr: Int`. As a client, any address `A` that we obtained from `alloc` must have a corresponding mapping in `m`:

```scala
val (a, m) = Mem.empty.alloc(N(42))
```

```
a: A = A(a = 1)
m: Mem = Map(A(1) -> N(42.0))
```

One further step we could take is to make `A` an abstract data type where the only way for a client to get an address `A` is via `alloc`.

32.1.3.2 Location Values

We define `isLValue` defining the expression forms that are location values following the e location judgment form (see Figure 32.3):

```scala
def isLValue(e: Expr): Boolean = e match {
  case Deref(A(_)) => true
  case _ => false
}
```

```
defined function isLValue
```

32.1.3.3 Substitution

Comparing **var** x = e_1 ; e_2 and **const** x = e_1 ; e_2, they are the same with respect to binding a variable whose scope is e_2. Thus, substitution works the same for both.

We define `substitute(with_e, x, in_e)` to implement

[`with_e` / `x`] `in_e`

assume that `with_e` and `in_e` have non-intersecting free variables (following Figure 21.1 in Section 21.9):

```scala
def substitute(with_e: Expr, x: String, in_e: Expr) = {
  // Assume that with_e and in_e have non-intersecting free variables.
  def $(in_e: Expr): Expr = in_e match {
    case N(_) | A(_) => in_e
    case Var(y) => if (x == y) with_e else in_e
    case VarDecl(y, e1, e2) => if (x == y) VarDecl(y, $(e1), e2) else VarDecl(y, $(e1), $(e2
    case Assign(e1, e2) => Assign($(e1), $(e2))
    case Deref(e1) => Deref($(e1))
  }
```

```
  $(in_e)
}
```

```
defined function substitute
```

32.1.3.4 Step

We can now define a `step` function following the $\langle e, m\rangle \longrightarrow \langle e', m'\rangle$ judgment form (see Figure 32.4).

32.1.3.4.1 Explicit State Passing

We first choose to define `step` with type `(Expr, Mem) => (Expr, Mem)`:

```scala
def step(e: Expr, m: Mem): (Expr, Mem) = e match {
  // DoDeref
  case Deref(a @ A(_)) => (m(a), m)
  // DoAssignVar
  case Assign(Deref(a @ A(_)), v) if isValue(v) => (v, m + (a -> v))
  // DoVarDecl
  case VarDecl(x, v1, e2) if isValue(v1) => {
    val (a, m_) = m.alloc(v1)
    (substitute(Deref(a), x, e2), m_)
  }
  // SearchAssign2
  case Assign(l1, e2) if isLValue(l1) => {
    val (e2_, m_) = step(e2, m)
    (Assign(l1, e2_), m_)
  }
  // Skip SearchAssign1
  // SearchVarDecl
  case VarDecl(x, e1, e2) => {
    val (e1_, m_) = step(e1, m)
    (VarDecl(x, e1_, e2), m_)
  }
}
```

```
defined function step
```

Let us test `step`:

```
val (e_assign_, m_)   = step(e_assign, Mem.empty)
val (e_assign__, m__) = step(e_assign_, m_)
```

```
e_assign_: Expr = Assign(e1 = Deref(e1 = A(a = 1)), e2 = N(n = 2.0))
m_: Mem = Map(A(1) -> N(1.0))
e_assign__: Expr = N(n = 2.0)
m__: Mem = Map(A(1) -> N(2.0))
```

32.1.3.4.2 Encapsulated State

While the above implementation of `step` faithfully implements the small-step operational semantics judgment form $\langle e, m\rangle \longrightarrow \langle e', m'\rangle$, we see it requires threading explicitly different versions of the memory state `m: Mem` (e.g., `m_`), which could be error prone.

Recall the idea of representing mutation effects by encapsulating a function of type `S => (S, A)` for a state type `S` and a main value type `A` into a data type `DoWith[S, A]` (see Section 30.5):

Listing 32.3 `DoWith._`

```
sealed class DoWith[S, A] private (doer: S => (S, A)) {
  def map[B](f: A => B): DoWith[S, B] = new DoWith[S, B]({ (s: S) => { val (s_, a) = doer(s)
  def flatMap[B](f: A => DoWith[S, B]): DoWith[S, B] = new DoWith[S, B]({ (s: S) => { val (s_
  def apply(s: S): (S, A) = doer(s)
}

object DoWith {
  def doget[S]: DoWith[S, S] = new DoWith[S, S]({ s => (s, s) })
  def doput[S](s: S): DoWith[S, Unit] = new DoWith[S, Unit]({ _ => (s, ()) })
  def doreturn[S, A](a: A): DoWith[S, A] = new DoWith[S, A]({ s => (s, a) })
  def domodify[S](f: S => S): DoWith[S, Unit] = new DoWith[S, Unit]({ s => (f(s), ()) })
}

import DoWith._
```

```
defined class DoWith
defined object DoWith
import DoWith._
```

Rearranging the type of `step` slightly, we see that we can implement the judgment form $\langle e, m \rangle \longrightarrow \langle e', m' \rangle$ using a `step` function of type `Expr => Mem => (Mem, Expr)` or `Expr => DoWith[Mem, Expr]`.

For convenience and to warm up, let us start by defining a helper function to the `alloc` method of `Mem` using a `DoWith[Mem, A]`:

```
def memalloc(v: Expr): DoWith[Mem, A] = doget flatMap { m =>
  val (a, m_) = m.alloc(v)
  doput(m_) map { _ => a }
}
```

```
defined function memalloc
```

We now translate the explicit state passing version of `step` from above (Section 32.1.3.4.1) to use an encapsulated `DoWith[Mem, Expr]` state as follows:

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
  // DoDeref
  case Deref(a @ A(_)) =>
    doget map { m => m(a) }

  // DoAssignVar
  case Assign(Deref(a @ A(_)), v) if isValue(v) =>
    domodify[Mem](m => m + (a -> v)) map { _ => v }

  // DoVarDecl
  case VarDecl(x, v1, e2) if isValue(v1) =>
    memalloc(v1) map { a => substitute(Deref(a), x, e2) }

  // SearchAssign2
  case Assign(l1, e2) if isLValue(l1) =>
    step(e2) map { e2 => Assign(l1, e2) }

  // Skip SearchAssign1

  // SearchVarDecl
  case VarDecl(x, e1, e2) =>
    step(e1) map { e1 => VarDecl(x, e1, e2) }
}
```

```
defined function step
```

We see that the memory state fades into the background, except where it is explicitly needed. It is in the implementation of the SEARCH rules where this fading into the background is particularly salient—whatever effect on memory happens in the recursive call to `step` is just passed along.

To use `step`, we can still run each step explicitly:

```
val (m_, e_assign_)   = step(e_assign)(Mem.empty)
val (m__, e_assign__) = step(e_assign_)(m_)
```

```
m_: Mem = Map(A(1) -> N(1.0))
e_assign_: Expr = Assign(e1 = Deref(e1 = A(a = 1)), e2 = N(n = 2.0))
m__: Mem = Map(A(1) -> N(2.0))
e_assign__: Expr = N(n = 2.0)
```

Observe that we have explicitly threaded the memory state in these top-level calls to `step` to show the intermediate memory state and expressions. That is, we called `step(e_assign)` to get a `DoWith[Mem, Expr]` that we then called with `Mem.empty` to get `(m_, e_assign_)` and then called the `DoWith[Mem, Expr]` resulting from `step(e_assign_)` with the current memory `m_`.

But we do not have to get the intermediate memory state `m_`. We can get the `DoWith[Mem, Expr]` for the two steps and then run it:

```
val (m__, e_assign__) = (step(e_assign) flatMap step)(Mem.empty)
```

```
m__: Mem = Map(A(1) -> N(2.0))
e_assign__: Expr = N(n = 2.0)
```

Or, we can rewrite the above make it more visible that `flatMap` is a sequential composition operator:

```
val (m__, e_assign__) = (doreturn(e_assign) flatMap step flatMap step)(Mem.empty)
```

```
m__: Mem = Map(A(1) -> N(2.0))
e_assign__: Expr = N(n = 2.0)
```

If desired, we can also use the `for-yield` expression syntax in Scala:

```scala
def step(e: Expr): DoWith[Mem, Expr] = e match {
  // DoDeref
  case Deref(a @ A(_)) =>
    for { m <- doget } yield m(a)

  // DoAssignVar
  case Assign(Deref(a @ A(_)), v) if isValue(v) =>
    for { _ <- domodify[Mem](m => m + (a -> v)) } yield v

  // DoVarDecl
  case VarDecl(x, v1, e2) if isValue(v1) =>
    for { a <- memalloc(v1) } yield substitute(Deref(a), x, e2)

  // SearchAssign2
  case Assign(l1, e2) if isLValue(l1) =>
    for { e2 <- step(e2) } yield Assign(l1, e2)

  // Skip SearchAssign1

  // SearchVarDecl
  case VarDecl(x, e1, e2) =>
    for { e1 <- step(e1) } yield VarDecl(x, e1, e2)
}
```

```
defined function step
```

The two `step` calls here look somewhat like having a side-effect on a mutable memory state, but in actuality, immutable memory states are threaded in the background:

```scala
val (m__, e_assign__) = (for {

  e_assign_  <- step(e_assign)
  e_assign__ <- step(e_assign_)

} yield e_assign_)(Mem.empty)
```

```
m__: Mem = Map(A(1) -> N(2.0))
e_assign__: Expr = Assign(e1 = Deref(e1 = A(a = 1)), e2 = N(n = 2.0))
```

TypeScripty - Formalize Type Checking

32.2 Other Effects

One might realize that before considering mutation in the above, we have considered another side-effecting JavaScripty expression in logging to the console:

```
console.log("Hello, World!")
```

The `console.log(` e `)` expression evaluates e to a value, logs that value to the console as a side-effect, and evaluates to **`undefined`**. Its effect is external to its final value **`undefined`**.

We gave this rule for DoPrint:

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

that states the logging effect informally with the "v_1 printed" condition.

If we want to describe explicitly that there is log of values (e.g., separated by linefeed characters LF), then we need to similarly reify a log state log

$$\text{logs} \quad \mathit{log} \quad ::= \cdot \mid \mathit{log} \,\textsc{lf}\, v$$

and extend our small-step judgment with a log state $\langle e, \mathit{log} \rangle \longrightarrow \langle e', \mathit{log}' \rangle$:

$$\textsc{DoPrint}\quad \frac{}{\langle \texttt{console.log}(v_1), \mathit{log} \rangle \longrightarrow \langle \textbf{undefined}, \mathit{log} \,\textsc{lf}\, v_1 \rangle}$$

References

[1] Dean, J. and Ghemawat, S. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM*. 51, 1 (2008), 107–113.

[2] Hoare, C.A.R. 2009. Null references: The billion dollar mistake. *QCon London* (2009).

[3] Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. DOI:https://doi.org/10.1016/0022-0000(78)90014-4.

[4] Odersky, M. et al. 2008. *Programming in Scala*. Artima.

[5] Odersky, M. et al. 2019. *Programming in Scala, fourth edition*. Artima.

[6] Python Software Foundation What is Python?