Concepts of Programming Languages

DRAFT

Boston University

September 17, 2025

Contents

Chapter 1

Introduction

When we write a program in our favorite programming language, we fill a file with a collection of symbols that we typed up using a computer keyboard. This is one of the beauties of programming: looking past the bells and whistles provided by editors and programming tools, a program is just a stream of characters. At some point in our programming workflow, we need to verify that what we've typed up works as expected, so we *run* our program. In some editors, there is literally a "play" button, but in many other cases this means opening a terminal and typing out a few commands. In either case, we're running a *different* program in order to run the program we've written. Our goal is to understand what's going on here: *What is this program doing? How does it do it? What sorts of data structures does this program use?* Our intuition should tell us this program is probably doing something fairly complicated.[1] So our basic question is: *How do we get from a stream of symbols representing a program to its output?*

1.1 The Interpretation Pipeline

The program to which we referred in the previous paragraph—the one that runs the programs we write—is called an *interpreter*. An interpreter takes as input a stream of characters, along with additional inputs (perhaps given as command line arguments or environment variables) and produces the output of our program. So another way of phrasing our basic question is simply: *what does an interpreter do?* As is typically the case, it's easier to answer a question like this by breaking it up into several subquestions. These subquestions correspond to passes in the *interpretation pipeline*, which is visualized in Figure 1.1. As we will understand it, an interpreter does four things:

1. It attempts to convert a sequence of characters (our program) into a sequence of *tokens*. We can think of tokens as the *units* or *atoms* of our program. This part of the interpreter is called **lexical analysis**, and its primary purpose is to simplify the process of analyzing the input program. It's easier, for example, to analyze a program if we know beforehand whether the character `1` which appears in our program is part of a number, a variable name, or a comment. Lexical analysis handles all these low-level syntactic concerns up front. In analogy with natural language, this is like the part of language processing in which your brain combines sequences of sounds or letters into whole words.

2. Once we have a sequence of tokens, our interpreter will attempt to convert them into an *abstract syntax tree (AST)*, which is a representation of our program as *hierarchical data*. This part of the

[1] Just think of how hard it is to follow directions, and how often we wish we read *all* the instructions before starting some task.

interpreter is called **syntactic analysis** or **parsing**. Hierarchical data is easier to analyze and evaluate, e.g., we'd like to know before evaluating our program if a variable `x` that might appear in our program is part of the expression we're evaluating or part of a new variable declaration. In analogy with natural language, this is like when your brain combines words into sub-phrases like prepositional phrases and verb phrases.

3. Not all programs we can write make sense. It doesn't make sense, for example, to add a number to a string.[2] The next part of an interpreter should analyze the hierarchical data from the previous part to verify that everything in it "looks reasonable." There are a couple ways to do this; this is generally a part of what is called **static (semantic) analysis**, but one of the most common form—and the way on which we'll focus because it's OCaml's approach—is **type checking**. Types help us describe what kind of things we're working with *before we start running our program*. They help us determine *before we start running our program* if we're working with data correctly. In analogy with natural language, this is like the uneasy feeling you get when you hear or see the sentence "the happiness is cold." The word "happiness" is not the right category of word for its location in the sentence, despite the fact that this sentence is grammatically correct. Not all languages have type checkers, but we maintain that "good" languages have type checkers.

4. Everything above is in service of *running* our program. This part of the interpreter is called **dynamic (semantic) analysis** or **evaluation**. This is the most intuitive part of an interpreter for programmers; learning to program is ultimately internalizing the evaluation rules of a programming language, so that we know what to write to accomplish a certain task.

This can be seen as our roadmap; we will consider each of these of parts in turn. For each part, there will be two perspectives: the theoretical perspective and the practical perspective. For example, the formal counterpart of syntactic analysis is **formal grammar** and the practical counterpart is **parsing**. We will build simple interpreters along the way, and by the end of the text we'll (hopefully) have an appreciation for the kinds of considerations that go into designing "good" programming languages.

Remark 1.1.1. It should be noted at the point that we will *not* consider **compilation**. In rough terms, compilation is the process of translating a program into another program in some target language, often a language that deals with low-level concerns like memory management and hardware architecture (e.g., assembly language or LLVM). We will always consider high-level semantics for our programming languages. And when we implement interpreters, we will work in high-level abstractions using OCaml.

[2] Even though some languages allow you to do this.

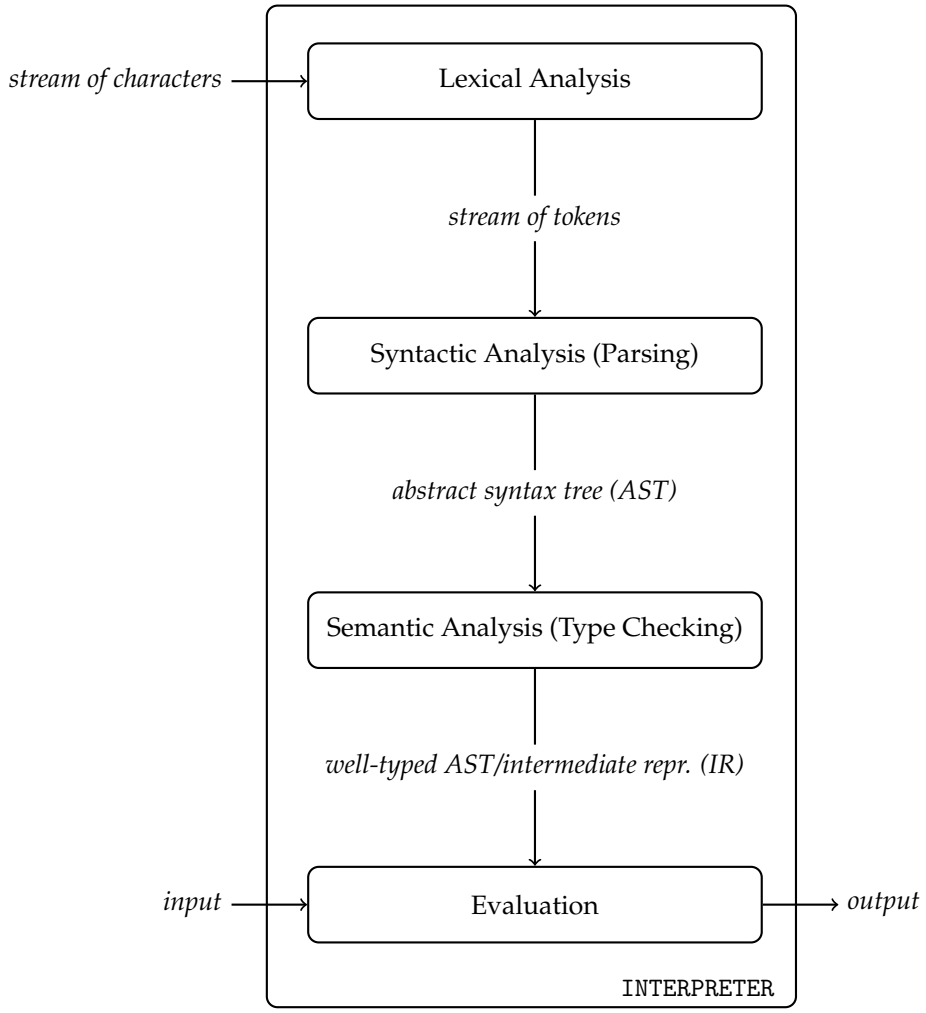


Figure 1.1: Visualization of the interpretation pipeline

Chapter 2

Inference Systems

The study of programming languages can be understood in part as the study of a collection of **inference systems**.[1] A programming language—in the sense of the techology that we use when we program—is just an *implementation* of this collection of inference systems.

Let's begin with a picture. As we said in Chapter 1, when we program we type a bunch of symbols into a file. Of the sequences of symbols we can type, only a handful of them are valid *sequences of tokens* of the language in which we're programming. For example, `let` is a keyword of OCaml, `1.223` is a floating point literal, `%` is an operator, and `yasn3__` is a valid variable name. These are all valid tokens in OCaml. Furthermore, OCaml requires that sequences of tokens are separated by whitespace, so any whitespace-separated combination of these is a valid sequence of tokens. But `,,A,,` is a meaningless sequences of symbols from OCaml's perspective, and cannot appear in a valid sequence of tokens.

Of the possible sequences of tokens, only a handful of those constitute *well-formed programs*. For example, `let f x = x + 2` is a valid OCaml program, whereas `let f rec = =` is not, though both are valid sequences of tokens.

Of the well-formed programs, only a handful make sense as programs; said another way, only a handful are *well-typed programs*. For example, `let rec f x = f x` is well-typed, whereas `let rec f x = f + x` is not, though both are well-formed programs.

And, of the well-typed programs, only a handful of *those* make sense as computations; said another way, only a handful can be successfully *evaluated*. This can be visualized as a nested collection of sets (Figure 2.1). Note that things get a bit interesting in the relationship between well-typed programs and programs that have values. We'll take this up in-depth when we cover type systems in **??**.

Our goal is to transform a sequence of symbols which make up our program into the *value* of our program. This is done by successive passes/transformations, each of which corresponds roughly to one of these nested sets:

- ▷ determining valid sequences of tokens in part of *lexical analysis*;
- ▷ determining well-formed programs is part of *parsing* or *syntactic analysis*;
- ▷ determining well-typed programs is part of *type-checking* or *static semantic analysis*;
- ▷ determining the value of a program is part of *evaluation* or *dynamic semantic analysis*.

[1] In some settings, these are called **formal systems** or **deductive systems**, though it's my sense that there's no real consensus as to what exactly these terms mean.

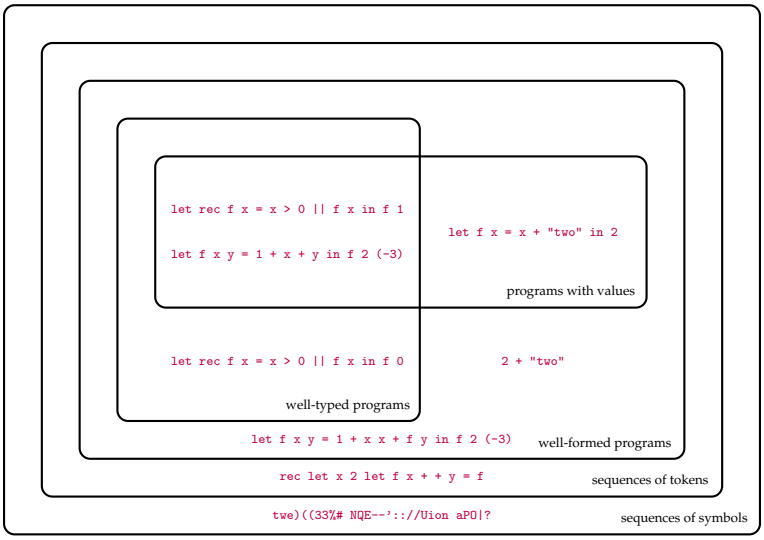


Figure 2.1: Visualization of nested classes of programs. In order to make the graphic simpler, the code examples are actually OCaml *expressions*, not programs.

And at each pass, there's an **inference system** that *formally* describes which inputs are "good" and which are "garbage."

In this chapter we will:

- ▷ formally define **inference rules**, i.e, the rules which govern when we can derive new judgments from judgements we've already derived within an inference system;
- ▷ use **derivations** to demonstrate that a judgment is derivable in an inference system;
- ▷ go through an extended example in which we define a collection of inference systems that define the syntax, typing behavior, and evaluation behavior of a simple calculator.

2.1 Inference Rules

An inference system is determined by a collection of **inference rules**. These inference rules describe what new information we're allowed to *infer* given previously inferred information. Inference rules will generally have the following form.

$$\frac{m \text{ is even} \qquad n = m + 2}{n \text{ is even}} \text{ (addTwo)}$$

In rough terms, this inference rule expresses that we can infer that a number n is even if we already know that $n - 2$ is even (i.e., if $n = m + 2$ and m is even). We begin by looking at the general anatomy of inference rules.

An inference system is defined over a fixed class of **judgments**. We think of the judgments as the things we can say while inferring something. Mathematically speaking, judgments can be anything we

want, elements of an arbitrary set. Practically speaking, we'll always take judgments to be *statements* parametrized by some other set.

Example 2.1.1. If we're interested in the parity of natural numbers, we may take $\mathcal{J}$, our set of judgments, to be all statements of the form

$$n \text{ is even} \qquad \text{or} \qquad n \text{ is odd}$$

So "7 is even" and "111 is odd" are both judgments in $\mathcal{J}$.

We'll call something like "n is even" a **parameterized judgment** and something like "4 is even" a **concrete judgment**, or just a judgment if it's clear from context that it's concrete. In particular, we'll say "110 is odd" is an **instance** of the parameterized judgment "n is odd."

Remark 2.1.1. We'll genearlly be a cavalier about this distinction between parametrized and concrete judgments. For example, we'll say that concrete judgments are also parameterized judgments, insofar as they're parametrized by no parameters. Likewise, we'll say that a parameterized judgment is in a set $\mathcal{J}$ when we really mean that its instances are in $\mathcal{J}$.

An **inference rule** is a way of describing what judgments we're allowed to infer given we've already inferred some other judgments. They also often include addition statements that need to hold in order for a rule to be applied; these are called **side conditions**.[2] The following is the general definition of an inference rule. It's important that we understand this definition, but not on our first pass; make sure to lean on the intuitions we've been trying to develop here as you work through this chapter, and come back to this definition potentially several times.

Definition 2.1.1. An **inference rule** named (ruleName) is a nonempty sequence of parameterized judgments $J_1, \dots J_k, J_{k+1}$ along with a collection of statements $S_1, \dots, S_l$ on the parameters used in those judgments. An inference rule is denoted by

$$\frac{J_1 \quad \dots \quad J_k \qquad S_1 \quad \dots \quad S_l}{J_{k+1}} \text{ (ruleName)}$$

We say that a concrete judgment J'_{k+1} **follows from** the concrete judgments $J'_1, \dots, J'_k$ **by (ruleName)** if J'_i is an instance of J_i for each index i and all statements $S_1, \dots, S_l$ hold for the parameters used in the given judgments. We'll often also denote this by

$$\frac{J'_1 \qquad \dots \qquad J'_k}{J'_{k+1}} \text{ (ruleName)}$$

and say that this is an **instance** of the rule (ruleName). We'll also call this simply an **inference**.

[2] In what follows we'll highlight side conditions to make clear that they are not formal judgments.

Example 2.1.2. The inference

$$\frac{10 \text{ is even}}{12 \text{ is even}} \text{ (addTwo)}$$

is an instance of the rule (addTwo) above. In particular, the side condition $12 = 10 + 2$ holds. Note that the side condition is not included in the instance, it's checked "offline" so to speak. Also note that

$$\frac{11 \text{ is even}}{13 \text{ is even}} \text{ (addTwo)}$$

is an instance of the rule (addTwo). It expresses that, hypothetically, if 11 is even, then it's reasonable to infer that 13 is even as well.

We call the judgments "above the line" **antecedences** and the judgment below called the **consequent**. It should be recognized that an inference rule may have no side-condition or no antecedents, it could even neither. An inference rule with no antecedents is called an **axiom**. As we'll see, an inference system can't be very interesting if it doesn't have any axioms; without them, nothing can be unconditionally inferred.q

2.2 Derivations

At this point, we can give a basic definition of an inference system.

Definition 2.2.1. An **inference system** over a set of judgment $\mathcal{J}$ is a collection of inference rules over $\mathcal{J}$.

Example 2.2.1. Let $\mathcal{J}$ be as in Example 2.1.1. We can consider the inference system given by the following two inference rules (note that these rules do not say anything about odd numbers).

$$\frac{}{0 \text{ is even}} \text{ (zero)} \qquad \frac{m \text{ is even} \qquad n = m + 2}{n \text{ is even}} \text{ (addTwo)}$$

What we're interested in with regards to inference systems is the collection of judgments that are *possible* to infer, i.e., the *closure* of the inference rules given in the system. We make this more concrete with the notion of a **derivation**, which is used to demonstrate that it's possible to infer a given judgment within an inference system by repeated invocations of its inference rules.

Definition 2.2.2. A **derivation** of the judgment J in an inference system $\mathcal{I}$ is a tree T with the following properties:

- ▷ The values of T are judgments from $\mathcal{J}$;
- ▷ $\mathsf{root}(T)$ is J;
- ▷ For every node of T of the form $\mathsf{node}(K, T_1, \ldots, T_k)$

$$\frac{\mathsf{root}(T_1) \qquad \ldots \qquad \mathsf{root}(T_k)}{K}$$

  is an instance an inference rule in $\mathcal{I}$.

In particular, the leaves of T are instances of axioms in $\mathcal{I}$.

It's in this definition that we see why we need axioms: without them, our derivations can't start anywhere. Figure 2.2 has a derivation of the judgment "8 is even" in the inference system from Example 2.2.1. It's not a terribly interesting derivation, but it captures the form of reasoning we're allowed to do in this system: 8 is even because 6 is even, which holds because 4 is even, which holds because 2 is even, which holds because 0 is even, which holds by definition.

Remark 2.2.1. Even though derivations are trees, we present them bottom-up as stacks of inferences. This is, in part, because it captures the structure of a proof more naturally, but it's also just historical. Gerhard Gentzen established much of the notation we use here (and in the mathematical field called *proof theory*) in the 1930s [1].

Let's consider a slightly more interesting inference system.

Example 2.2.2. We can define an alternative system which takes advantage of our ability to reason about odd numbers.

$$\frac{}{0 \text{ is even}}\ (\text{zero}) \qquad \frac{}{1 \text{ is odd}}\ (\text{one})$$

$$\frac{m \text{ is even} \qquad n = m + 1}{n \text{ is odd}}\ (\text{odd})$$

$$\frac{m \text{ is odd} \qquad n \text{ is odd} \qquad k = m + n}{k \text{ is even}}\ (\text{even})$$

Figure 2.3 is a derivation of "8 is even" is this alternative system. The primary takeaway: different inference systems provide us with different forms of reasoning. In this alternative system, we have to first separate 8 into two summands and reason about them separately. This system also differs from the one in Example 2.2.1 because a judgment may have multiple proofs.

Exercise 2.2.1. Give another derivation of "8 is even" in this system from Example 2.2.2.

Exercise 2.2.2. (Challenge) Prove that "n is even" is derivable in the system from Example 2.2.1 if and only if it's derivable in the system from Example 2.2.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \text{ is even}}\,(\text{zero})}{2 \text{ is even}}\,(\text{addTwo})}{4 \text{ is even}}\,(\text{addTwo})}{6 \text{ is even}}\,(\text{addTwo})}{8 \text{ is even}}\,(\text{addTwo})$$

Figure 2.2: A derivation of "8 is even" in the inference system from Example 2.2.1.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1 \text{ is odd}}\,(\text{one}) \quad \dfrac{}{1 \text{ is odd}}\,(\text{one})}{2 \text{ is even}}\,(\text{even})}{3 \text{ is odd}}\,(\text{odd}) \quad \dfrac{}{1 \text{ is odd}}\,(\text{one})}{4 \text{ is even}}\,(\text{even}) \quad \dfrac{\dfrac{}{1 \text{ is odd}}\,(\text{one}) \quad \dfrac{}{1 \text{ is odd}}\,(\text{one})}{2 \text{ is even}}\,(\text{even})}{5 \text{ is odd}}\,(\text{odd}) \quad 3 \text{ is odd}\,(\text{odd})}{8 \text{ is even}}\,(\text{even})$$

Figure 2.3: Derivation of "8 is even" in the inference system from Example 2.2.2

2.3 Extended Example: Calculator

We now consider several inference systems defined over judgments about expressions for a calculator with lisp-like syntax. Anticipating the next chapter, the formal syntax for such expressions is give in Figure 2.4. This syntax allows for only 8 possible symbols: `(`, `)`, `+`, `*`, `=`, `?`, `0`, and `1`. It is not expected at this point that you understand this specification; it's just meant to give a hint of what is to come.

We begin by defining an inference system that determines which sequences of symbols constitute well-formed expressions (Figure 2.5). This system is defined over judgments of the form "$e \in \mathsf{WF}$" where e is a sequence of symbols. It captures in its inference rules that operators appear before their arguments, and that all invocations of an operator are surrounded in parentheses. For example, `(* (+ 1 1) (+ 1 1))` is a well-formed expression, and we can derive this judgment formally (Figure 2.6).

```
<expr> ::= ( + <expr> <expr> )
         | ( * <expr> <expr> )
         | ( = <expr> <expr> )
         | ( ? <expr> <expr> <expr> )
         | 0 | 1
```

Figure 2.4: Lisp-like syntax for a simple calculator

$$\frac{}{\texttt{0} \in \mathsf{WF}}\ \text{(zero)} \qquad \frac{}{\texttt{1} \in \mathsf{WF}}\ \text{(one)}$$

$$\frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(}\ \texttt{+}\ e_1\ e_2\ \texttt{)} \in \mathsf{WF}}\ \text{(add)} \qquad \frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(}\ \texttt{*}\ e_1\ e_2\ \texttt{)} \in \mathsf{WF}}\ \text{(mul)}$$

$$\frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(}\ \texttt{=}\ e_1\ e_2\ \texttt{)} \in \mathsf{WF}}\ \text{(eq)}$$

$$\frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF} \qquad e_3 \in \mathsf{WF}}{\texttt{(}\ \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)} \in \mathsf{WF}}\ \text{(cond)}$$

Figure 2.5: Inference systems for well-formed expressions in the language from Figure 2.4

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} \in \mathsf{WF}}\ \text{(one)} \quad \dfrac{}{\texttt{1} \in \mathsf{WF}}\ \text{(one)}}{\texttt{(+ 1 1)} \in \mathsf{WF}}\ \text{(add)} \quad \dfrac{\dfrac{}{\texttt{1} \in \mathsf{WF}}\ \text{(one)} \quad \dfrac{}{\texttt{1} \in \mathsf{WF}}\ \text{(one)}}{\texttt{(+ 1 1)} \in \mathsf{WF}}\ \text{(add)}}{\texttt{(* (+ 1 1) (+ 1 1))} \in \mathsf{WF}}\ \text{(mul)}$$

Figure 2.6: Derivation of "`(* (+ 1 1) (+ 1 1))` $\in$ WF" in the system from Figure 2.5.

Remark 2.3.1. Keep in mind that these expressions don't *mean* anything yet, even though we can guess that "`+`" will stand for addition and "`*`" will stand for multiplication. So, for example `(+ (= 1 1) (= 0 1))` is also a well-formed expression, even though this may not make sense if we interpret "`=`" as an equality operator which evalutes to a Boolean value.

Once we've defined the well-formed expressions using an inference system, we can prove things about well-formed expressions using induction on derivations (see Appendix A).

Exercise 2.3.1. Prove using induction on derivations that if $e \in \mathsf{WF}$, then the parentheses in e are balanced.

In addition to proving things via structural induction, we can also define properties on well-formed expressions by defining new inference systems. In particular, we can use the notion of a well-formed expression in other inference systems.

Suppose, for example, we want to reason about how many symbols are in a given well-formed expression. We can define an inference system to do this sort of reasoning (Figure 2.7). This system is defined over judgments of the form "$\#(e) = n$" where e ranges over well-formed expressions (i.e., we can derive "$e \in \mathsf{WF}$" in the previously defined inference system) and n ranges over natural numbers. The rules in this system express that `0` and `1` have 1 symbol, and every other expression has 3 plus [the number of symbols in its subexpressions] many symbols. We can derive within this system that `(* (+ 1 1) (+ 1 1))` has 13 symbols (Figure 2.8).

At this point, we'd like to start thinking about the *meaning* of these expressions. First, we must contend with the fact that not all well-formed expressions are meaningful. For example `(= (= 0 0) 1)` doesn't make much sense because it would require us to compare two values that aren't the "same kind

$$\frac{}{\#(\texttt{0}) = 1}\ \text{(zero)} \qquad \frac{}{\#(\texttt{1}) = 1}\ \text{(one)}$$

$$\frac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(}\ \texttt{+}\ e_1\ e_2\ \texttt{)}) = k}\ \text{(add)}$$

$$\frac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(}\ \texttt{*}\ e_1\ e_2\ \texttt{)}) = k}\ \text{(mul)}$$

$$\frac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(}\ \texttt{=}\ e_1\ e_2\ \texttt{)}) = k}\ \text{(eq)}$$

$$\frac{\#(e_1) = l \qquad \#(e_2) = m \qquad \#(e_3) = n \qquad k = l + m + n + 3}{\#(\texttt{(}\ \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)}) = k}\ \text{(add)}$$

Figure 2.7: Inference system for determining the number of symbols in a well-formed expression

$$\dfrac{\dfrac{\dfrac{}{\#(\texttt{1}) = 1}\ \text{(one)} \quad \dfrac{}{\#(\texttt{1}) = 1}\ \text{(one)}}{\#(\texttt{(+ 1 1)}) = 5}\ \text{(add)} \quad \dfrac{\dfrac{}{\#(\texttt{1}) = 1}\ \text{(one)} \quad \dfrac{}{\#(\texttt{1}) = 1}\ \text{(one)}}{\#(\texttt{(+ 1 1)}) = 5}\ \text{(add)}}{\#(\texttt{(* (+ 1 1) (+ 1 1))}) = 13}\ \text{(mul)}$$

Figure 2.8: Derivation of "$\#(\texttt{(* (+ 1 1) (+ 1 1))}) = 13$" in the system from Figure 2.7

$$\frac{}{\texttt{0} : \texttt{int}}\ \text{(zero)} \qquad \frac{}{\texttt{1} : \texttt{int}}\ \text{(one)}$$

$$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(}\ \texttt{+}\ e_1\ e_2\ \texttt{)} : \texttt{int}}\ \text{(add)} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(}\ \texttt{*}\ e_1\ e_2\ \texttt{)} : \texttt{int}}\ \text{(mul)}$$

$$\frac{e_1 : t_1 \qquad e_2 : t_2 \qquad t_1 = t_2}{\texttt{(}\ \texttt{=}\ e_1\ e_2\ \texttt{)} : \texttt{bool}}\ \text{(eq)} \qquad \frac{e_1 : \texttt{bool} \qquad e_2 : t_2 \qquad e_3 : t_3 \qquad t_2 = t_3}{\texttt{(}\ \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)} : t_2}\ \text{(cond)}$$

Figure 2.9: Inference system for determining the type of an expression

of thing." Formally, the "kind of thing" a value can be is called its **type**. There are two types of values that an expression can be in our toy calculator language: a number (`int`) or a Boolean value (`bool`).

Remark 2.3.2. It would be possible to represent Boolean values as numbers; this is done, for example, in C. Then there would be a single type of value in our toy lanugage. We'll avoid doing this, in part because it makes our example less interesting, but also because there are some serious problems that arise when we design programming languages this way.

We can define an inference system which determines the type of an expression, if it has one. This means that the system has two purposes: (1) it delineates which expressions have values (i.e., which we can evaluate), and (2) it determine the type of the value *before* we've evaluated the expression (Figure 2.9).

This system is defined over judgments of the form "$e : t$" where e is a well-formed expression and t is either `int` or `bool`. Let's take a brief moment to read what each of these rules says.

- ▷ (zero) says `0` is a `int`, no matter what.
- ▷ (one) says `1` is a `int`, no matter what.
- ▷ (add) says if e_1 is a `int` (i.e., evaluates to a number) and e_2 is a `int`, then `(` $+$ e_1 e_2 `)` is a `int`. In particular, we can only add numbers, not Boolean values.
- ▷ (mul) says if e_1 is a `int` (i.e., evaluates to a number) and e_2 is a `int`, then `(` $*$ e_1 e_2 `)` is a `int`. In particular, we can only multiply numbers, not Boolean values.
- ▷ (eq) says if e_1 and e_2 are the same type then we can compare them, and `(` $=$ e_1 e_2 `)` is a `bool`. In particular, we can't compare a number with a Boolean value.
- ▷ (cond) says if e_1 is a `bool`, and e_2 and e_3 are the same type, then we can condition on the value of e_1 to get either the value of e_2 or e_3, and the type of `(` $?$ e_1 e_2 e_3 `)` is the same as that of e_2 and e_3.

See Figure 2.10 for an example derivation in this system.

Remark 2.3.3. Notice that, ultimately, an inference rule, is a more compact way of expressing a statement in (mathematical) English. It takes some practice to read inference rules, but it is worthwhile to learn in the long run, and eventually becomes easier than reading the equivalent English.

Finally, we can define an inference system for determining the value of an expression in our toy calculator language (Figure 2.11). This system is defined over judgments of the form "$e \Downarrow v$", which we read as "e evaluates to v," where e ranges over well-formed expressions, and v ranges over numbers ($\mathbb{N}$) or Boolean values ($\{\top, \bot\}$). Let's again take a moment to read what each of these rules says.

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\,(\text{one}) \quad \dfrac{}{\texttt{1} : \texttt{int}}\,(\text{one})}{\texttt{(= 1 1)} : \texttt{bool}} \quad \dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\,(\text{one}) \quad \dfrac{}{\texttt{1} : \texttt{int}}\,(\text{one})}{\texttt{(+ 1 1)} : \texttt{int}}\,(\text{add}) \quad \dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\,(\text{one}) \quad \dfrac{}{\texttt{0} : \texttt{int}}\,(\text{zero})}{\texttt{(* 1 0)} : \texttt{int}}\,(\text{mul})}{\texttt{(? (= 1 1) (+ 1 1) (* 1 0))} : \texttt{int}}\,(\text{cond})$$

Figure 2.10: Derivation of "`(? (= 1 1) (+ 1 1) (* 1 0))` : `int`" in the system from Figure 2.9

$$\dfrac{}{\texttt{0} \Downarrow 0}\,(\text{zero}) \qquad \dfrac{}{\texttt{1} \Downarrow 1}\,(\text{one})$$

$$\dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{\texttt{(} + e_1\ e_2\ \texttt{)} \Downarrow v}\,(\text{add}) \qquad \dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 \times v_2}{\texttt{(} * e_1\ e_2\ \texttt{)} \Downarrow v}\,(\text{mul})$$

$$\dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 = v_2}{\texttt{(} = e_1\ e_2\ \texttt{)} \Downarrow \top}\,(\text{eqTrue}) \qquad \dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\texttt{(} = e_1\ e_2\ \texttt{)} \Downarrow \bot}\,(\text{eqFalse})$$

$$\dfrac{e_1 \Downarrow \top \qquad e_2 \Downarrow v}{\texttt{(} ?\ e_1\ e_2\ e_3\ \texttt{)} \Downarrow v}\,(\text{ifTrue}) \qquad \dfrac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v}{\texttt{(} ?\ e_1\ e_2\ e_3\ \texttt{)} \Downarrow v}\,(\text{ifFalse})$$

Figure 2.11: Inference system for determine the values of well-formed expressions

- ▷ (zero) says `0` evaluates to the number 0.
- ▷ (one) says `1` evaluates to the number 1.
- ▷ (add) says if e_1 evaluates to m and e_2 evaluates to n, then `(` e_1 e_2 `)` evaluates to $m + n$.
- ▷ (mul) says if e_1 evaluates to m and e_2 evaluates to n, then `(` e_1 e_2 `)` evaluates to $m \times n$.
- ▷ (eqTrue) says if e_1 evaluates to v_1 and e_2 evaluates to v_2, and v_1 and v_2 are the same then `(` e_1 e_2 `)` evaluates to true.
- ▷ (eqFalse) says if e_1 evaluates to v_1 and e_2 evaluates to v_2, and v_1 and v_2 are *not* the same then `(` e_1 e_2 `)` evaluates to false.
- ▷ (ifTrue) says if e_1 evaluates to true and e_2 evaluates to v, then `(` e_1 e_2 e_3 `)` evaluates to v. In particular, e_3 does not need to be evaluated.
- ▷ (ifFalse) says if e_1 evaluates to true and e_3 evaluates to v, then `(` e_1 e_2 e_3 `)` evaluates to v. In particular, e_2 does not need to be evaluated.

See Figure 2.12 for an example derivation in this system. Notice that the side conditions are where the "real" computation happens, e.g., the (add) rule draws a correspondence between the `+` operator in our language and normal addition "+" in the side condition.

There is quite a bit more we could say about this example, but we'll leave it for now. The purpose of this presentation is primarily to offer examples of inferences systems like the ones on which we'll focus for the remainder of the text. We defer more careful considerations to those later chapters.

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} \Downarrow 1}\,\text{(one)} \quad \dfrac{}{\texttt{1} \Downarrow 1}\,\text{(one)}}{\texttt{(+ 1 1)} \Downarrow 2}\,\text{(add)} \quad \dfrac{\dfrac{}{\texttt{1} \Downarrow 1}\,\text{(one)} \quad \dfrac{}{\texttt{1} \Downarrow 1}\,\text{(one)}}{\texttt{(+ 1 1)} \Downarrow 2}\,\text{(add)}}{\texttt{(* (+ 1 1) (+ 1 1))} \Downarrow 4}\,\text{(mul)}$$

Figure 2.12: Derivation of "`(* (+ 1 1) (+ 1 1))` $\Downarrow 4$" in the system from Figure 2.11

Appendix A

Trees

Trees—or, more generally inductively-defined structures—are core to the study of programming languages. This is, in part, why functional programming languages like OCaml are well-suited for *implementing* programming languages. As such, we have to spend some time on the humble notion of trees. This appendix covers a small slice of the topic, only what we need for the main part of the text.[1]

If you've taken a course in discrete mathematics, you've likely seen the graph-theoretic definition of trees.

Definition A.0.1. A **tree** is undirected graph which is connected and acyclic. A **directed tree** is a directed graph whose underlying graph is a tree. A directed tree is **rooted** if there is a unique vertex with in-degree 0.

We'll be primarily interested in *nonempty rooted directed* trees.[2] These are also sometimes called **rose trees**. To make this more explicit we'll work with the *inductive* definition of (rose) trees.

Definition A.0.2. A **tree** with values from V is defined inductively as follows:

- ▷ if v is a value from V and $T_1, \ldots, T_k$ are trees then so is $\mathsf{node}(v, T_1, \ldots, T_k)$.

Note, in particular, that $\mathsf{node}(v)$ a tree for any value v from V. We call this kind of tree a **leaf**. The **root** of a tree is defined as:

$$\mathsf{root}\,(\mathsf{node}(v, T_1, \ldots, T_k)) \triangleq v$$

and the **children** a tree are defined as:

$$\mathsf{children}\,(\mathsf{node}(v, T_1, \ldots, T_k)) \triangleq (T_1, \ldots, T_k)$$

[1] There are whole books dedicated to the study of mathematical induction and inductively defined structures.
[2] Moving forward "tree" we will always mean "nonempty rooted directed tree."

Example A.0.1. Decision trees represent boolean functions, and we can represent a decisions tree as a rose tree. One decision tree for the boolean function $\mathsf{OR}(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$ is:

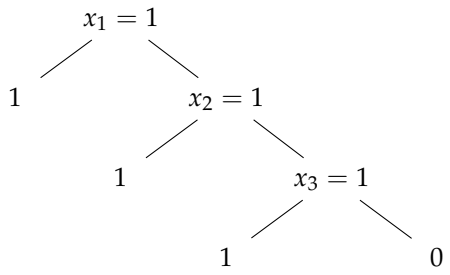
$$\mathsf{node}(x_1 = 1, \mathsf{node}(1), \mathsf{node}(x_2 = 1, \mathsf{node}(1), \mathsf{node}(x_3 = 1, \mathsf{node}(1), \mathsf{node}(0))))$$

Values of this tree are queries to the inputs of the function. Roughly speaking, this decision tree expresses that the value of $\mathsf{OR}(x_1, x_2, x_3)$ can be determined by searching from left to right for an input which is equal to 1.

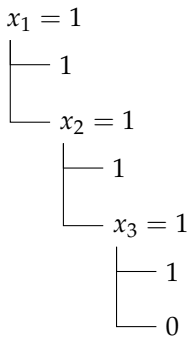
We can naturally define rose trees in OCaml, which means we can also define the usual functions on trees:

```
type 'a rose_tree = Node of 'a * 'a rose_tree list
let example = Node (1, [Node (2, []); Node (3, [])])

let rec depth (Node (_, ts)) =
  List.fold_left
    (fun acc n -> max acc (n + 1))
    0
    (List.map depth ts)

let rec size (Node (_, ts)) =
  List.(fold_left (+) 1 (map size ts))

let _ = assert (depth example = 1)
let _ = assert (size example = 3)
```

We visualize trees in the usual way. Here, for example, is a visualization of the tree from Example A.0.1:



We'll also occassionally use the following form of tree visualization, which we'll call **compact form**:



This form is based on visualizations of file trees.[3] It's particularly useful when a tree it too wide to draw otherwise, e.g., when the values in nodes are themselves very wide. There's one more form of tree visualization we'll use for drawing derivation trees; this will be covered in Chapter 2.

A.1 Structural Induction

Induction is a principle used to prove universal statements about inductively-defined structures (like trees). If you've taken a course in discrete mathematics, you've likely seen the principle of natural number induction:

> *Given a property P of natural numbers, if:*
>
> - ▷ *P holds of the number 0;*
> - ▷ *for every number k, if we assume P holds of k (an assumption often called the **induction hypothesis**), then we can demonstrate that P holds of $k + 1$;*
>
> *then P holds of every natural number.*

Exercise A.1.1. Prove that

$$2\sum_{i=1}^{n} i = n(n+1)$$

holds for all natural numbers n using natural number induction.

Inductive structures—by definition of being inductive—have analogous induction principles. Here's the principle of tree induction:

> *Given a property P of trees with values from V, if*
>
> - ▷ *for any value v from V, and trees $T_1, \ldots, T_k$, if we assume that P holds of each tree $T_1, \ldots, T_k$*[4] *then we can demonstrate that P holds of $\mathsf{node}(v, T_1, \ldots, T_k)$;*
>
> *then P holds of all trees with values from V.*

[3] If you're interested, look up the command line tool `tree`, or just type `tree .` in your terminal and see what you get.
[4] Again, this assumption is called the *induction hypothesis*.

We won't concern ourselves further with the general notion of trees.[5] We'll focus on **inductively-define subsets** of trees.

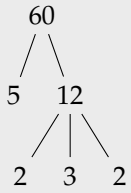
Notation A.1.1. For a set A, we write A^* for the set of finite sequences of elements of A and we write A^+ for the subset of nonempty sequences in A^*. For example, the sequence $(1, 2, 3, 4, 5)$ is an element of both $\mathbb{N}^*$ and $\mathbb{N}^+$.

Definition A.1.1. Let Q be a property on V^+. The **inductively-defined subset** of trees $\mathcal{Q}$ given by Q is defined (inductively) as follows:

- ▷ for any value v from V and trees $T_1, \ldots, T_k$ from $\mathcal{Q}$, if Q holds of $(v, \mathsf{root}(v_1), \ldots, \mathsf{root}(v_k))$ then $\mathsf{node}(v, T_1, \ldots, T_k) \in \mathcal{Q}$.

In other words, inductively-defined subsets of trees are defined by describing when we're allowed to build new trees from those we've already built.

Example A.1.1. Consider the inductively-defined set $\mathcal{Q}$ of trees with values from $\mathbb{N}$ given by the property Q which holds of $(v, v_1, \ldots, v_k)$ when v is prime or $v = v_1 \ldots v_k$. That is, the value at the root of a given tree is prime or the product of the roots of its children. The following is an example of such a tree.



We might recognize that the leaves of this tree define a prime factorization of the root, e.g., $60 = 5 * 2 * 3 * 2$.

Once we have inductively-defined subsets of trees, we can define their corresponding principle of induction. It's the same as the principle of tree induction but with a slightly stronger induction hypothesis:

> *Given a property P of trees with values from V and a property Q which inductively-defines a subset $\mathcal{Q}$ of trees, if*
>
> - ▷ *for any value v from V, and trees $T_1, \ldots, T_k$ from $\mathcal{Q}$, if we assume that P holds of each tree $T_1, \ldots, T_k$ and that Q holds of the sequence $(v, \mathsf{root}(T_1), \ldots, \mathsf{root}(T_k))$ then we can demonstrate that P holds of $\mathsf{node}(v, T_1, \ldots, T_k)$;*
>
> *then P holds of all trees in $\mathcal{Q}$.*

Let's see this in action. Consider the subset $\mathcal{Q}$ of trees defined in Example A.1.1. We'd like to prove that for any tree T from $\mathcal{Q}$, the number $\mathsf{root}(T)$ has a prime factorization.

[5] There isn't much more we can say without also formally defining *recursion* on trees, another interesting topic that we'll unceremoniously brush under the rug.

Proof. Let $\mathsf{node}(n, T_1, \ldots, T_k)$ be a tree in $\mathcal{Q}$ and suppose that $\mathsf{root}\,(T_i)$ has a prime factorization $p_{1,i} \ldots p_{l_i,i}$ for all indices i. Furthermore we assume, by definition of $\mathcal{Q}$, that n is prime or

$$n = \mathsf{root}\,(T_1) \ldots \mathsf{root}\,(T_k)$$

If n is prime, then n is also its prime factorization. Otherwise,

$$n = \prod_{i=1}^{k} \mathsf{root}\,(T_i) = \prod_{i=1}^{k} (p_{1,i} \ldots p_{l_i,i})$$

yielding a prime factorization of n. That is, we take the product of all the prime factorizations of the children of T to get a prime factorization of its root.[6] □

A.2 Induction on Derivations

What we've done so far is more general than necessary, and is ultimately in service of defining the *derivation induction principle*. First, observe that the set of derivations in a given inference system $\mathcal{I}$, as defined in Chapter 2, is an inductively-define subset of trees. The property is straightforward: it holds of the nonempty sequence of judgments $(J, J_1, \ldots, J_k)$ when

$$\frac{J_1 \qquad \cdots \qquad J_k}{J}$$

is (an instance of) an inference rule of $\mathcal{I}$. In other words, a derivation is a tree in which every node is either an axiom or follows from an inference rule. The upshot is that the derivation induction principle is a specialization of the induction principle for inductively-defined subsets of trees.

Our second observation: being able to prove that a property holds of all derivations is enough to be able to prove that a property holds of all *derivable judgments*. There's is an obvious correspondence between derivations and derivable judgments: if a judgment is derivable, then it has a derivation. And this is ultimately what we care about when we reason about things like type safety (**??**). All said, we can state the derivation induction principle as follows:

> *Given a property P of judgments of $\mathcal{I}$, if*
>
> - ▷ *for any judgment J and derivable judgments $J_1, \ldots, J_k$, if we assume that P holds of each judgment $J_1, \ldots, J_k$ where*
>
> $$\frac{J_1 \qquad \cdots \qquad J_k}{J}$$
>
> *then we can demonstrate that P holds of J;*
>
> *then P holds of all derivable judgments of $\mathcal{D}$.*

This principle tells us that we can look at the *last* inference rule applied and prove that the property holds assuming it holds of the *antecedents* of the applied rule.

It may be easier to grok this principle by example. Consider the following basic inference system over judgments of the form "n is even" where $n \in \mathbb{N}$.

[6] What we've done here is only *part of* the fundamental theorem of arithmetic. We would also need to prove that every natural number is the root of some tree in $\mathcal{Q}$, which we can prove by (strong) induction over natural numbers (we also say nothing of uniqueness...).

Example A.2.1. This system is taken directly from Chapter 2.

$$\frac{}{0 \text{ is even}}\ (\text{zero}) \qquad \frac{m \text{ is even} \qquad n = m + 2}{n \text{ is even}}\ (\text{addTwo})$$

We'd like to prove that this inference system is sound, i.e., that if "n is even" is derivable, then n is, in fact, even (in particular, "3 is even" is not derivable). We can prove this by induction on derivations.

Proof. Let J denote an arbitrary judgment "n is even." There are two cases to consider.

- ▷ If J follows from (zero), then J must be the judgment "0 is even" in which case the property holds.
- ▷ If J follows from (addTwo), then "$n - 2$ is even" must be derivable and we may assume that $n - 2$ is even. This implies that n itself is even.

□

In a sense, this example might be *too* simple. It's not immediately clear that we've done anything in this proof; the distinction between n being even and the judgment "n is even" being derivable is admittedly a subtle one. But, in order to avoid ballooning this appendix into a full-blown chapter, I'll relegate the presentation of more interesting examples to **??**.

Bibliography

[1] Gerhard Gentzen. Investigations into Logical Deduction. *American philosophical quarterly*, 1(4):288–306, 1964.