Concepts of Programming Languages

DRAFT

Boston University

January 22, 2026

Contents

Preface

This text is by its nature strange. It's the source text for the second part of `CAS CS 320`: *Concepts of Programming Languages* at Boston University, and the intended audience is ultimately students in this course. My hope is that it might find a wider audience, but describing a more inclusive notion of "intended audience" requires some context: `CS320` is a course in the computer science cirriculum at Boston University that is semi-required. And as much as I believe that everyone should know *something* about how programming languages work, there is a non-negligible portion of the student population that does *not* agree with this sentiment. So we're put in the difficult but perennial position: *how do we get students interested in a topic they're studying because they're told they have to?* The point: the intended audience of this text is undergraduate students who don't necessarily want to do research in programming languages, but "want" to know something about how programming languages work. There's a lot we don't cover, and although we try to maintain the rigor of some of the more serious texts in the field [6, 4, 3, 5] we're not gonna to lose sleep over the less-than-complete coverage we inevitably end up with.

The other operative part of the description above is "the second part of". We typically spend the first part of the course teaching OCaml from OCaml Programming: Correct + Efficient + Beautiful. This text should be viewed to some degree as an extension of that one, with the section on interpreters fleshed out. As such, we're greatly in debt to Michael R. Clarkson et al. who have put the time and energy into that amazing (and freely available) book.

This text is based on material created by several members of the *Principles of Programming and Verification (POPV)* group at Boston University, most notably Ankush Das, Robin Fu, Marco Gaboardi, Assaf Kfoury, and Hongwei Xi.

Chapter 1

Introduction

When you write a program in your favorite programming language, you fill a file with characters, typically by typing on a keyboard. This is one of the beauties of programming: looking past the bells and whistles provided by IDEs, a program is a stream of characters. At some point in the programming workflow, you need to verify that what you've typed up works as expected, so you *run* the program. In some cases this might mean pressing a "play" button, but in many cases it means opening a terminal and typing a few commands. In either case, you're running a *different* program—an **interpreter**—in order to run the program you yourself have written. Our goal is to understand: *What does an interpreter do? How does it do it, e.g., what sorts of data structures does it use?* Above all, *how do we get from a stream of characters representing a program to its output value?*

1.1 The Interpretation Pipeline

Let's get more concrete. An **interpreter** is a program that takes as input a stream of characters, along with additional inputs (e.g., user-provided command line arguments) and produces the output value of the program represented by the input stream of characters, if the stream of characters does in fact represent a valid program; the interpreter may choose to report an error otherwise. There several steps to this process which make up the *interpretation pipeline* visualized in Figure 1.1. As we will come to understand it, an interpreter does four things:

1. It attempts to convert the input stream of characters into a stream of *tokens* by grouping together related characters. We think of tokens as the *units* or *atoms* of our programming langauge This part interpretation is called **lexical analysis**, and its primary purpose is to simplify the process of analyzing the input program. It's useful, for example, to know whether the character `1` that appears in our program is part of a number (e.g., `210`) or a variable name (e.g., `case1`). Lexical analysis handles all these low-level syntactic concerns up front. In analogy with natural language, this is akin to when your brain combines sequences of sounds or letters into whole words.

2. The interpreter then attempts to convert the stream of tokens into an *abstract syntax tree (AST)*, which is a representation of our program as *hierarchical data*. This part of interpretation is called **syntactic analysis** or **parsing**. Hierarchical data is easier to analyze and evaluate, e.g., it's useful to know whether a variable `x` that appear in our program is part of an arithmetic expression (e.g., `x + 1`) or part of a new variable declaration (e.g., `let x = 1`). In analogy with natural language, this is akin to when your brain combines words into sub-phrases like prepositional phrases and verb phrases.

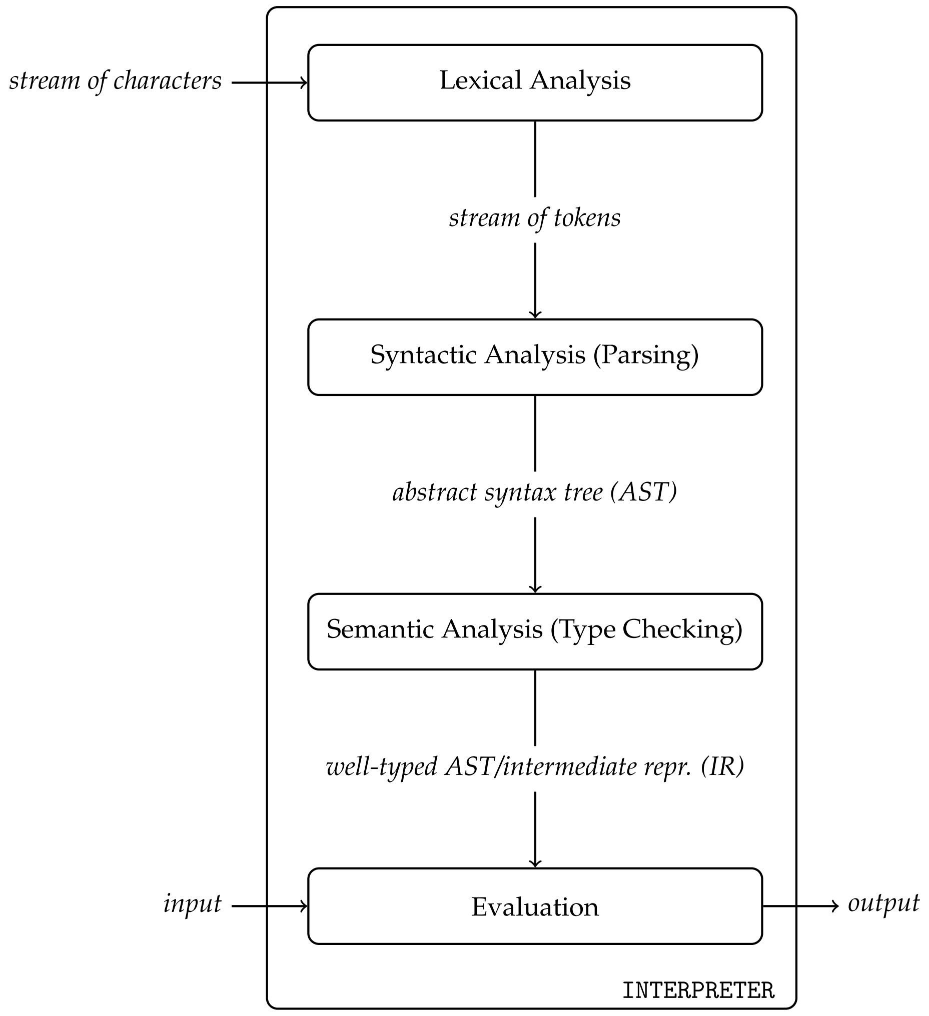

stream of characters
Lexical Analysis
stream of tokens
Syntactic Analysis (Parsing)
abstract syntax tree (AST)
Semantic Analysis (Type Checking)
well-typed AST/intermediate repr. (IR)
input
Evaluation
output
INTERPRETER

Figure 1.1: Visualization of the interpretation pipeline

3. Not all programs we can write make sense. It doesn't make sense, for example, to add a number to a string.[1] The next part of interpretation consists of analyzing an AST to verify that everything in it "looks reasonable"; this is generally called **static (semantic) analysis**. There are many forms of static analysis, but the one we'll focus on is **type checking**. Types help us describe what kind of things we're working with *before we run our program*. They help us determine *before we run our program* if we're using data correctly. In analogy with natural language, this is akin to the uneasy feeling you get when you hear or see a sentence like "the happiness is cold". The word "happiness" isn't the right kind of word for its location in the sentence, despite the fact that the 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 interpretion 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.

The about outline should be understood as our roadmap; we will consider each part in turn, and build several interpreters along the way. 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**. By the end we hop you'll have gained an appreciation for the considerations that go into designing "good" programming languages.

Remark 1.1.1. It should be noted at this point that we will *not* consider **compilation** in this text. In rough terms, compilation is the process of translating a program in one programming language into an *equivalent* program in a different programming language. The target language of the translation is often one that deals with low-level concerns like memory management (e.g., assembly language or LLVM). We'll only consider high-level semantics for our programming languages, and when we implement interpreters we'll work in high-level abstractions using OCaml. If you're interested in compilers, there are many great sources, perhaps mostly notably the "dragon book" [1].

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

Chapter 2

Inference Systems

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

We begin with a picture. As we said in Chapter 1, when we program we type a bunch of characters into a file. Of the sequence of characters we can type, only a handful of them are valid *sequences of tokens* in our programming language. 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.[2] Furthermore, OCaml requires that 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.

Now, 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*.

The above observations can be visualized as nested sets (Figure 2.1). Note that things get 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 Chapter 7.

Our goal is to transform a sequence of characters representing a program into the output *value* of the program. This is done by successive transformations, each of which corresponds roughly to one of these nested sets:

▷ determining valid sequences of tokens in part of *lexical analysis*;

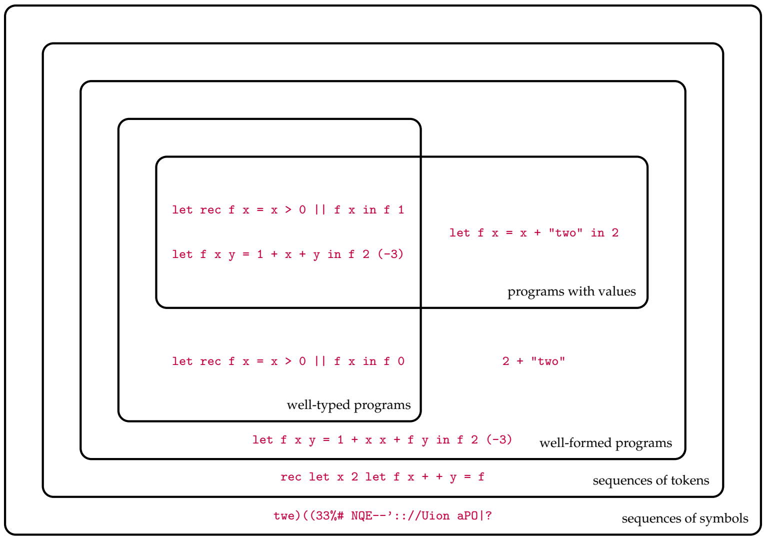
▷ determining well-formed programs is part of *parsing* or *syntactic analysis*;

▷ 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.

[2] We'll take of the question of what makes a valid token more formally in **??**.



```
let rec f x = x > 0 || f x in f 1
let f x y = 1 + x + y in f 2 (-3)
let f x = x + "two" in 2
let rec f x = x > 0 || f x in f 0
2 + "two"
let f x y = 1 + x x + f y in f 2 (-3)
rec let x 2 let f x + + y = f
twe)((33%# NQE--'::://Uion aPO|?
```

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 each transformation has a corresponding inference system that *formally* describes which inputs are "good" and which are "junk".

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 in an inference system;
- ▷ use **derivations** to demonstrate that a judgment is derivable in an inference system;
- ▷ present an extended example which consists of a collection of inference systems that define the syntax, typing behavior, and evaluation behavior of a simple programming language for a 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.

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

This inference rule expresses that we can infer a number n is even if we already know that $n = m + 2$ and m is even (i.e., if $n - 2$ 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 judgments as the things we can say while inferring something. Mathematically speaking, judgments can be anything we want, i.e., 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 our set of judgments $\mathcal{J}$ 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 $\mathcal{J}$. Note that statements in $\mathcal{J}$ are parametrized by the set of natural numbers.

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 be a bit cavalier about the 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 additional statements that need to hold in order for a given rule to be applied; these are called **side conditions**.[3] 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 $S_1, \dots, S_l$ depending on the parameters used in those judgments. An inference rule is denoted by

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

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

$$\frac{J'_1 \quad \dots \quad J'_k}{J'_{k+1}}\ \textsc{ruleName}$$

and say that this is an **instance** of the rule RULENAME. We'll also call this an **inference**.

We call the judgments above the line **antecedences** and the judgment below called the **consequent**. An inference rule may have no side-condition or no antecedents. An inference rule with no antecedents is called an **axiom**.

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

Example 2.1.2. The inference

$$\dfrac{10 \text{ is even}}{12 \text{ is even}}\ \textsc{addTwo}$$

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". Also note that

$$\dfrac{11 \text{ is even}}{13 \text{ is even}}\ \textsc{addTwo}$$

is an instance of ADDTWO. It expresses that, hypothetically, if 11 were even then 13 would be even.

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).

$$\dfrac{}{0 \text{ is even}}\ \textsc{zero} \qquad\qquad \dfrac{m \text{ is even} \qquad n = m + 2}{n \text{ is even}}\ \textsc{addTwo}$$

What we're interested in with regard to inference systems is the collection of judgments that are *possible* to infer. We make this concrete with the notion of a **derivation**, which is used to demonstrate that it's possible to infer a given judgment in an inference system by invocations of its inference rules (see Appendix A for a refresher on trees).

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 $\mathcal{J}$;
- ▷ root (T) is J;
- ▷ For every node of T of the form $\texttt{node}(K, T_1, \dots, T_l)$

$$\dfrac{\text{root }(T_1) \qquad \dots \qquad \text{root }(T_l)}{K}\ \textsc{rule}$$

  is an instance of RULE, where RULE appears 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 axioms are important: 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 fiat.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{0 \text{ is even}}\,\textsc{Zero}}{2 \text{ is even}}\,\textsc{AddTwo}}{4 \text{ is even}}\,\textsc{AddTwo}}{6 \text{ is even}}\,\textsc{AddTwo}}{8 \text{ is even}}\,\textsc{AddTwo}$$

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

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{1 \text{ is odd}}\,\textsc{One} \quad \cfrac{}{1 \text{ is odd}}\,\textsc{One}}{2 \text{ is even}}\,\textsc{Even}}{3 \text{ is odd}}\,\textsc{Odd} \quad \cfrac{}{1 \text{ is odd}}\,\textsc{One}}{4 \text{ is even}}\,\textsc{Even}}{5 \text{ is odd}}\,\textsc{Odd} \quad \cfrac{\cfrac{\cfrac{}{1 \text{ is odd}}\,\textsc{One} \quad \cfrac{}{1 \text{ is odd}}\,\textsc{One}}{2 \text{ is even}}\,\textsc{Even}}{3 \text{ is odd}}\,\textsc{Odd}}{8 \text{ is even}}\,\textsc{Even}$$

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

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 derivation more naturally, but it’s also historical: Gerhard Gentzen established much of the notation we use here (and in the mathematical field called *proof theory*) in the 1930s [2].

Example 2.2.2. Let’s consider a slightly more interesting inference system. We can define an alternative system for parity reasoning which takes advantage of what we know about odd numbers.

$$\frac{}{0 \text{ is even}}\,\textsc{Zero} \qquad \frac{}{1 \text{ is odd}}\,\textsc{One} \qquad \frac{m \text{ is even} \qquad n = m + 1}{n \text{ is odd}}\,\textsc{Odd}$$

$$\frac{m \text{ is odd} \qquad n \text{ is odd} \qquad k = m + n}{k \text{ is even}}\,\textsc{Even}$$

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.

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

Figure 2.4: Lisp-like syntax for a simple calculator

2.3 Extended Example: Calculator

We now consider several inference systems defined over judgments about expressions for a calculator with lisp-like syntax. This syntax will only allow for 8 possible tokens: `(`, `)`, `+`, `*`, `=`, `?`, `0`, and `1`. Anticipating the next chapter, the formal grammar for such expressions is give in Figure 2.4. You're not expected to 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 tokens constitute well-formed expressions (Figure 2.5).[4] This system is defined over judgments of the form "$e \in \mathsf{WF}$" where e is a sequence of tokens. It captures in its inference rules that operators appear before their arguments, and that all invocations of an operator are surrounded in parentheses, e.g.,

```
( * ( + 1 1 ) ( + 1 1 ) )
```

is a well-formed expression, and we can derive this judgment formally (Figure 2.6). Let's take a brief moment to read what each of these rules says.

- ▷ ZEROS says that `0` is a well-formed expression, no matter what.
- ▷ ONES says that `1` is a well-formed expression, no matter what.
- ▷ ADDS says that that if e_1 and e_2 are well-formed expressions, then so is `(` `+` e_1 e_2 `)`.
- ▷ MULS says that that if e_1 and e_2 are well-formed expressions, then so is `(` `*` e_1 e_2 `)`.
- ▷ EQS says that that if e_1 and e_2 are well-formed expressions, then so is `(` `=` e_1 e_2 `)`.
- ▷ CONDS says that that if e_1, e_2, and e_3 are well-formed expressions, then so is `(` `?` e_1 e_2 e_3 `)`.

We cannot stress this enough: *inference rules are formal shorthand for expressing rules which can also be expressed in natural language.* The natural language descriptions above may at first seem a lot easier to read. But over time, as you learn to read inference rules, you'll find that they are *much* more convenient, especially when the inference system has 50 (or even 500) rules.

Exercise 2.3.1. Show that `( = ( = 0 0 ) 1 )` $\in \mathsf{WF}$ by deriving it in the inference system from Figure 2.5.

[4]The astute reader might notice that we're skipping lexical analysis. This is partly for simplicity, but it's also because all our tokens are exactly one character long, and so, barring issues of whitespace for now, *any* sequence of valid characters constitutes a valid sequence of tokens.

$$\frac{}{\texttt{0} \in \mathsf{WF}}\ \textsc{zeroS} \qquad \frac{}{\texttt{1} \in \mathsf{WF}}\ \textsc{oneS} \qquad \frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(} \texttt{+} \ e_1 \ e_2 \ \texttt{)} \in \mathsf{WF}}\ \textsc{addS} \qquad \frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(} \ \texttt{*} \ e_1 \ e_2 \ \texttt{)} \in \mathsf{WF}}\ \textsc{mulS}$$

$$\frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF}}{\texttt{(} \ \texttt{=} \ e_1 \ e_2 \ \texttt{)} \in \mathsf{WF}}\ \textsc{eqS} \qquad \frac{e_1 \in \mathsf{WF} \qquad e_2 \in \mathsf{WF} \qquad e_3 \in \mathsf{WF}}{\texttt{(} \ \texttt{?} \ e_1 \ e_2 \ e_3 \ \texttt{)} \in \mathsf{WF}}\ \textsc{condS}$$

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

$$\frac{\dfrac{\dfrac{}{\texttt{1} \in \mathsf{WF}}\ \textsc{oneS} \qquad \dfrac{}{\texttt{1} \in \mathsf{WF}}\ \textsc{oneS}}{\texttt{(+ 1 1)} \in \mathsf{WF}}\ \textsc{addS} \qquad \dfrac{\dfrac{}{\texttt{1} \in \mathsf{WF}}\ \textsc{oneS} \qquad \dfrac{}{\texttt{1} \in \mathsf{WF}}\ \textsc{oneS}}{\texttt{(+ 1 1)} \in \mathsf{WF}}\ \textsc{addS}}{\texttt{(* (+ 1 1) (+ 1 1))} \in \mathsf{WF}}\ \textsc{mulS}$$

Figure 2.6: Derivation of "`( * ( + 1 1 ) ( + 1 1 ) )` $\in \mathsf{WF}$" 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. The expression

```
( + ( = 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 2.3.2. Prove using induction on derivations that if $e \in \mathsf{WF}$, then there are the same number of left parentheses as right parentheses in e.

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 formally derive that the expression

```
( * ( + 1 1 ) ( + 1 1 ) )
```

has 13 symbols (Figure 2.8).

$$\dfrac{}{\#(\texttt{0}) = 1}\ \textsc{zeroC} \qquad \dfrac{}{\#(\texttt{1}) = 1}\ \textsc{oneC} \qquad \dfrac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(} \texttt{ + } e_1\ e_2 \texttt{)}) = k}\ \textsc{addC}$$

$$\dfrac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(} \texttt{ * } e_1\ e_2 \texttt{)}) = k}\ \textsc{mulC} \qquad \dfrac{\#(e_1) = m \qquad \#(e_2) = n \qquad k = m + n + 3}{\#(\texttt{(} \texttt{ = } e_1\ e_2 \texttt{)}) = k}\ \textsc{eqC}$$

$$\dfrac{\#(e_1) = l \qquad \#(e_2) = m \qquad \#(e_3) = n \qquad k = l + m + n + 3}{\#(\texttt{(} \texttt{ ? } e_1\ e_2\ e_3 \texttt{)}) = k}\ \textsc{condC}$$

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

$$\dfrac{\dfrac{\dfrac{}{\#(\texttt{1}) = 1}\ \textsc{oneC} \qquad \dfrac{}{\#(\texttt{1}) = 1}\ \textsc{oneC}}{\#(\texttt{(+ 1 1)}) = 5}\ \textsc{addC} \qquad \dfrac{\dfrac{}{\#(\texttt{1}) = 1}\ \textsc{oneC} \qquad \dfrac{}{\#(\texttt{1}) = 1}\ \textsc{oneC}}{\#(\texttt{(+ 1 1)}) = 5}\ \textsc{addC}}{\#(\texttt{(* (+ 1 1) (+ 1 1))}) = 13}\ \textsc{mulC}$$

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

Exercise 2.3.3. Show that $\#(\texttt{(= (= 0 0) 1)}) = 9$ by deriving of it in the inference system from Figure 2.7.

We'd like to now start thinking about the *meaning* of these expressions. First, we have to contend with the fact that not all well-formed expressions are meaningful. The expression `( = ( = 0 0 ) 1 )` cannot be evaluated successfully because it would require us to compare two values that aren't the "same kind 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 evaluate to in our 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 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 a set of expressions that have values (i.e., which we can evaluate), and (2) it determines the type of that 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`. We read a judgment of this form as "e is of type t" or "e is a t". Let's again take a brief moment to read what each of these rules says.

- ▷ ZEROT says `0` is a `int`, no matter what.
- ▷ ONET says `1` is a `int`, no matter what.
- ▷ ADDT 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.

$$\frac{}{\texttt{0} : \texttt{int}}\ \textsc{ZeroT} \qquad \frac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(}\ \texttt{+}\ e_1\ e_2\ \texttt{)} : \texttt{int}}\ \textsc{AddT} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(}\ \texttt{*}\ e_1\ e_2\ \texttt{)} : \texttt{int}}\ \textsc{MulT}$$

$$\frac{e_1 : t_1 \qquad e_2 : t_2 \qquad t_1 = t_2}{\texttt{(}\ \texttt{=}\ e_1\ e_2\ \texttt{)} : \texttt{bool}}\ \textsc{EqT} \qquad \frac{e_1 : \texttt{bool} \qquad e_2 : t_2 \qquad e_3 : t_3 \qquad t_2 = t_3}{\texttt{(}\ \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)} : t_2}\ \textsc{CondT}$$

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

$$\frac{\dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT} \qquad \dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT}}{\texttt{(= 1 1)} : \texttt{bool}}\ \textsc{EqT} \qquad \dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT} \qquad \dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT}}{\texttt{(+ 1 1)} : \texttt{int}}\ \textsc{AddT} \qquad \dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{OneT} \qquad \dfrac{}{\texttt{0} : \texttt{int}}\ \textsc{ZeroT}}{\texttt{(* 1 0)} : \texttt{int}}\ \textsc{MulT}}{\texttt{(? (= 1 1) (+ 1 1) (* 1 0))} : \texttt{int}}\ \textsc{CondT}$$

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

- ▷ MULT 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.
- ▷ EQT 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.
- ▷ CONDT 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.

Exercise 2.3.4. Show that `( = ( + 0 0 ) 1 ) : bool` by deriving it in the inference system from Figure 2.9.

Finally, we can define an inference system for determining the value of an expression in our 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 ($\{\mathsf{true}, \mathsf{false}\}$). Let's on last time take a moment to read what each of these rules says.

- ▷ ZEROE says `0` evaluates to the number 0.
- ▷ ONEE says `1` evaluates to the number 1.
- ▷ ADDE says if e_1 evaluates to m and e_2 evaluates to n, then `( +` e_1 e_2 `)` evaluates to $m + n$.
- ▷ MULE says if e_1 evaluates to m and e_2 evaluates to n, then `( *` e_1 e_2 `)` evaluates to $m \times n$.
- ▷ EQTRUEE 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.
- ▷ EQFALSEE 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.

$$\frac{}{\texttt{0} \Downarrow 0}\ \textsc{ZeroE} \qquad \frac{}{\texttt{1} \Downarrow 1}\ \textsc{OneE} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{\texttt{(} \texttt{+}\ e_1\ e_2\ \texttt{)} \Downarrow v}\ \textsc{AddE}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 \times v_2}{\texttt{(} \texttt{*}\ e_1\ e_2\ \texttt{)} \Downarrow v}\ \textsc{MulE} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 = v_2}{\texttt{(} \texttt{=}\ e_1\ e_2\ \texttt{)} \Downarrow \mathsf{true}}\ \textsc{EqTrue}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\texttt{(} \texttt{=}\ e_1\ e_2\ \texttt{)} \Downarrow \mathsf{false}}\ \textsc{EqFalse} \qquad \frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow v}{\texttt{(} \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)} \Downarrow v}\ \textsc{IfTrue}$$

$$\frac{e_1 \Downarrow \mathsf{false} \qquad 3_2 \Downarrow v}{\texttt{(} \texttt{?}\ e_1\ e_2\ e_3\ \texttt{)} \Downarrow v}\ \textsc{IfTrue}$$

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

$$\frac{\dfrac{\dfrac{}{\texttt{1} \Downarrow 1}\ \textsc{OneE} \qquad \dfrac{}{\texttt{1} \Downarrow 1}\ \textsc{OneE}}{\texttt{(+ 1 1)} \Downarrow 2}\ \textsc{AddE} \qquad \dfrac{\dfrac{}{\texttt{1} \Downarrow 1}\ \textsc{OneE} \qquad \dfrac{}{\texttt{1} \Downarrow 1}\ \textsc{OneE}}{\texttt{(+ 1 1)} \Downarrow 2}\ \textsc{AddE}}{\texttt{(* (+ 1 1) (+ 1 1))} \Downarrow 4}\ \textsc{MulE}$$

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

- ▷ IFTRUEE 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.
- ▷ IFFALSEE 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.

Exercise 2.3.5. Determine the value v such that `(= (+ 0 1) 1)` $\Downarrow v$ and derive this judgment in the system from Figure 2.11.

There is quite a bit more we could say here. For example, this system exhibits **type safety**: for any expression e, if there is a type t such that $e : t$, then there is a value v such that $e \Downarrow v$ and v is of type t. We'll come back to this later. 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.

2.4 Additional Exercises

Chapter 3

Formal Grammar

Most people are familiar with grammar in the context of natural language. In English class, for example, we might learn that we shouldn't end a sentence with a preposition. This is a *grammatical rule* of English. In broad strokes, grammar refers to the rules that govern what constitutes a well-formed sentence in a languange. It's the concern of grammar to determine that "I taught the car in the refrigerator" is well-formed, whereas "I car teach refrigerator in there" is not. It's *not* the concern of grammar to determine that the first sentence, despite being grammatically correct, has no reasonable interpretation in English.

The grammar of a programming language determines what constitutes a well-formed *program* in that language. Due to the precision of programming languages, these tend to be called **formal grammars**. In OCaml, the program:

```
let f x = x + 1
```

is well-formed, but the program:

```
f let x = x 1 +
```

is not. As with natural language, the grammar of a programming language is not concerned with the *meaning* of programs, just their well-formedness, e.g., the program:

```
let omega x = x x
```

is well-formed, but isn't well-typed.

A good programming language has a clear specification of its grammar. The grammar of OCaml, for example, is given in its entirety in The OCaml Manual. After going though this chapter, you should be able to read the specification given there.[1]

3.1 A Thought Experiment

How do we know that a sentence is grammatically correct? Let's try to break down the cognitive process of this determination. Consider the following English sentence.

`the cow jumped over the moon`

[1]In order to not be entirely biased, I'll note that Python also has a complete specification of its grammar in The Python Manual.

We might first recognize that each word is a particular part of speech (noun, verb, preposition, etc.) and that, with regards to grammar, *only the part of speech counts*; that is, it doesn't matter *what* noun we use, just that we use a noun in the correct position in our sentence. We represent this step of the cognitive process by replacing each word with an abstract symbol that stands in for its part of speech.

<article> <noun> <verb> <prep> <article> <noun>

After this, there are some familiar patterns: <article> <noun> is the determination or quantification of an object, e.g., "the house" or "a car." In other words, this pattern represents a *noun phrase* or *nominal phrase*. We mentally group these forms, and this group can be represented by a new symbol.

<noun-phrase> <verb> <prep> <noun-phrase>

Another pattern we might recognize: a proposition followed by a noun phrase is a single unit, e.g., "over the moon", "through the woods", or "behind the wall." These are called *prepositional phrases* and can, again, be represented by a new symbol.

<noun-phrase> <verb> <prep-phrase>

Prepositional phrases modify verbs, creating a *verb phrase* (e.g., "ran to the car") leaving us with something like:

<noun-phrase> <verb-phrase>

which we should finally recognized as the canonical structure of a well-formed sentence: *thing does*.

The point of this thought experiment is to emphasize the following: there are general structures that are allowed by a grammar, and these structures can be *instantiated* at particular words in the language. The rules that define these structures are often heirarchical; a sentence is made of parts (e.g, a noun phrase and a verb phrase) and those constituent parts are made of smaller parts (e.g., an article and a noun) and so on until we reach units of our language, the particular words. Demonstrating that a sentence is grammatically correct means demonstrating that it fits into this heirarchy of allowed structures.

Next, an observation: if we reverse the order of steps we took above, and break up some of the steps we did in parallel, we get a *proof* that our sentence is grammatically correct (Figure 3.1). We call this a **derivation**. Each line corresponds to the application of a grammatical rule. For example, the fourth line follows from the third by way of applying the grammatical rule that a prepositional phrase can be made up of a preposition followed by a verb phrase. But more importantly, if we squint, we can start to see the tree-like hierarchical structure embodied in this derivation, a structure we call a **parse tree** (Figure 3.2).

3.2 Backus-Naur Form

A formal grammar is defined over a fixed collection of symbols. This collection is divided into two disjoint groups: the **terminal symbols** and the **non-terminal** symbols. We denote nonterminal symbols using notation of the form <non-terminal>. We denote terminal symbols in red typewriter font, e.g., `terminal`.

Remark 3.2.1. We almost never state outright what our underlying symbols are. We can determine the terminal and non-terminal symbols of a grammar by looking at its specification, assuming that every symbol must appear at least once in the specification.

<sentence>

<noun-phrase> <verb-phrase>

<noun-phrase> <verb> <prep-phrase>

<noun-phrase> <verb> <prep> <noun-phrase>

<noun-phrase> <verb> <prep> <article> <noun>

<article> <noun> <verb> <prep> <article> <noun>

`the` <noun> <verb> <prep> <article> <noun>

`the cow` <verb> <prep> <article> <noun>

`the cow jumped` <prep> <article> <noun>

`the cow jumped over` <article> <noun>

`the cow jumped over the` <noun>

`the cow jumped over the moon`

Figure 3.1: Derivation of `the cow jumped over the moon` in the grammar in Figure 3.3

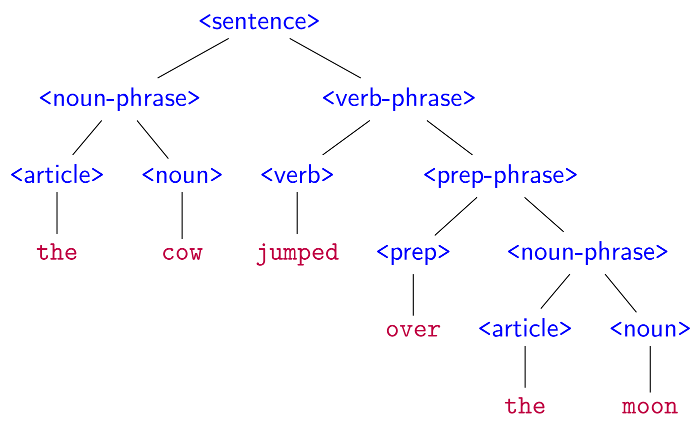


Figure 3.2: Parse tree for `the cow jumped over the moon` in the grammar in Figure 3.3

In the derivation in Figure 3.1, we built a sequence of not-quite sentences, until the very last one, which was the sentence we wanted to prove grammatically correct. These possibly-not-quite sentences are called **sentential forms**.

Definition 3.2.1. A **sentential form** is a sequence of symbols (terminal or non-terminal). A **sentence** is a sequence of terminal symbols. We will use the symbol ε to refer to the empty sentential form.

Remark 3.2.2. We denote sentential forms using whitespace as a delimiter. But it's important to recognize that *this is just notation.* We're not interested in low-level concerns like whitespace when studying formal grammar. It may be useful to think of a sentential form as a *list* of symbols.

Example 3.2.1. The following are sentential forms that appear above:

```
<sentence>
<noun-phrase> <verb> <prep> <article> <noun>
the cow jumped over the moon
```

but only the last one is a sentence.

In the process of constructing sentential forms, we replaced a non-terminal symbol with a sentential form, e.g., we replaced `<noun-phrase>` with `<article> <noun>`. These replacements are specified using **production rules**.

Definition 3.2.2. A **production rule** is a nonterminal symbol together with a sentential form. Production rules are denoted using equations of the form

```
<non-terminal> ::= SENTENTIAL-FORM
```

where the left-hand side is the nonterminal symbol and the right-hand side is the sentential form.

We read a production rule as: "the non-terminal symbol on the left-hand side can be replaced with the sentential form on the right hand side."

Example 3.2.2. We can express that a prepositional phrase can be made up of a preposition followed by a noun-phrase using the following production rule:

```
<prep-phrase> ::= <prep> <noun-phrase>
```

Backus-Naur Form (BNF) specifications are used to describe **context-free grammars**, which form a class of formal grammars that are expressive enough to capture the syntax of programming languages. A BNF grammar is determined by what replacements we're allowed to perform and what kind of things we're trying to derive.

Definition 3.2.3. A **Backus-Naur Form (BNF) specification** is a collection of production rules, together with a designated symbol called the **starting symbol**. If the start symbols is not explicitely given, it is taken to be the left-hand side of the *first* rule appearing in the specification.

```
   <sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase>
<verb-phrase> ::= <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
    <article> ::= the
       <noun> ::= cow
       <noun> ::= moon
       <verb> ::= jumped
       <prep> ::= over
```

Figure 3.3: BNF specification for a simple grammar based on English

Example 3.2.3. Figure 3.3 has a BNF specification for the underlying grammar of the previous section. The nonterminal symbols of this grammar are: <sentence>, <noun-phrase>, <verb-phrase>, <verb>, <prep>, <prep-phrase>, <article>, and <noun>. The terminal symbols of this grammar are: `the`, `cow`, `moon`,`jumped`, and `over`. The nonterminal symbol <sentence> is our starting symbol (because it appears as the left-hand side of the first rule). This is, of course, not a complete or accurate model of English grammar (such a model would be incredibly complicated).

Remark 3.2.3. John Backus and Peter Naur are computer scientists who formalized this meta-syntax in the process of developing and implementing the programming language ALGOL 60.

As Figure 3.3 indicates, it's possible for a nonterminal symbol to have multiple associated production rules. This is very common so it has it's own syntax.

Definition 3.2.4. We write

$$\text{<non-terminal>} ::= \mathsf{SENTENTIAL\text{-}FORM}_1 \mid \dots \mid \mathsf{SENTENTIAL\text{-}FORM}_k$$

as shorthand for

$$\begin{aligned}\text{<non-terminal>} &::= \mathsf{SENTENTIAL\text{-}FORM}_1 \\ &\vdots \\ \text{<non-terminal>} &::= \mathsf{SENTENTIAL\text{-}FORM}_k\end{aligned}$$

The multiple forms of a given nonterminal symbol are often called **alternatives**, and this shorthand is often called **alternative syntax**.

It's not difficult to rewrite the grammar in Figure 3.3 using alternative syntax (Figure 3.4). The last piece of our thought experiment that we need to formalize is the *proof* of grammatical correctness.

```
   <sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase> | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
    <article> ::= the
       <noun> ::= cow | moon
       <verb> ::= jumped
       <prep> ::= over
```

Figure 3.4: The grammar in Figure 3.3 rewritten with alternative syntax

Definition 3.2.5. A **derivation** of a sentence S in a BNF grammar $\mathcal{G}$ is a sequence of sentential forms $S_1, \ldots, S_k$ with the following properties:

- ▷ S_1 is the starting symbol of $\mathcal{G}$;
- ▷ $S_k = S$;
- ▷ for every index i satisfying $i > 1$, the sentential form S_i is the result of replacing a single nonterminal symbol in S_{i-1} with a sentential form, according to a production rule of $\mathcal{G}$.

We say that a grammar **recognizes** a sentence S if there is a derivation of S in the grammar.

Example 3.2.4. The derivation in Figure 3.1 is a derivation is the sense of Definition 3.2.5. We can also derive the sentence `the moon jumped` as follows:

```
<sentence>
<noun-phrase> <verb-phrase>
<article> <noun> <verb-phrase>
the <noun> <verb-phrase>
the moon <verb-phrase>
the moon <verb>
the moon jumped
```

Exercise 3.2.1. For the derivation in Example 3.2.4, determine which production rule is applied in each line of the derivation.

A sentence is not guaranteed to have a *unique* derivation, but there a form of derivation we'll single out for reasons which will become more clear at the end of this section.

Definition 3.2.6. A **leftmost derivation** is one in which the leftmost nonterminal symbol is expanded in each step.

The derivation given Example 3.2.4 is a leftmost derivation, whereas the derivation in Figure 3.1 is not.

Exercise 3.2.2. Write a leftmost derivation of the sentence

`the cow jumped over the moon`

in the grammar in Figure 3.3.

The theme of this section has been that grammars imbue sentences with hierarchical structure. We represent this hierarchical structure using **parse trees**.

Definition 3.2.7. A **parse tree** for a sentence S recognized by a grammar $\mathcal{G}$ is a tree T with the following properties:

- ▷ The values of T are symbols of $\mathcal{G}$;
- ▷ $\textsf{root}\,(T)$ is the starting symbol of $\mathcal{G}$;
- ▷ For every node of T of the form $\texttt{node}(s, T_1, \dots, T_k)$

  $$s \ ::= \ \textsf{root}\,(T_1) \ \dots \ \textsf{root}\,(T_k)$$

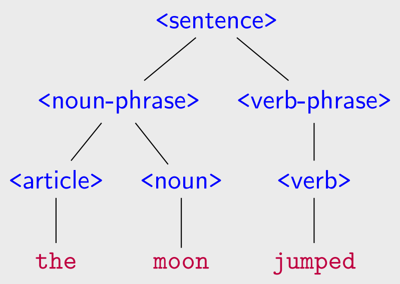
  is a production rule of $\mathcal{G}$;
- ▷ The terminal symbols at the leaves of T, in order from left to right form the sentence S.

Note that the value at a leaf l may be a nonterminal symbol in the case that

$$\textsf{root}\,(l) \ ::= \ \varepsilon$$

is a production rule of $\mathcal{G}$.

Example 3.2.5. The tree in Figure 3.2 is a parse tree in the sense of . The following is a parse tree for the sentence `the moon jumped` in the same grammar:



If a sentence S has a derivation in a grammar $\mathcal{G}$, then it also has a parse tree, and vice versa. Multiple derivations may correspond to the same parse tree, but to every parse tree there is a unique corresponding leftmost derivation; a leftmost derivation corresponds to a pre-order traversal of a parse tree.

Theorem 3.2.1. There is a one-to-one correspondence between parse trees and leftmost derivations. In particular, a sentence S has exactly one parse tree in $\mathcal{G}$ if and only if it has a exactly one leftmost derivation in $\mathcal{G}$.

```
<program> ::= <stmts>
  <stmts> ::= ε | <stmt> ; <stmts>
   <stmt> ::= <var> = <expr>
   <expr> ::= <term> | <term> + <term> | <term> - <term>
   <term> ::= <var> | <num>
    <num> ::= 1 | 2 | 3
    <var> ::= a | b | c | d
```

Figure 3.5: BNF specification for a toy imperative programming language

Exercise 3.2.3. (Challenge) Prove Theorem 3.2.1.

Remark 3.2.4. In addition to context-free grammars, there are also context-*sensitive* grammars. For context-sensitive grammars, production rules can transform *sequences of* symbols, i.e., the left-hand side of a production rule is not required to be a single nonterminal symbol.

3.3 Examples

Despite all the abstractions and formalisms, formal grammars are suppose to be fairly straigtfoward. The easiest way to grok them is by example. In this section, we'll look at a couple more complicated examples based on programming languages.

An Imperative Language

Figure 3.5 has a BNF specification for a toy imperative programming language which, in essense, consists of what are called *straight-line programs*. In English, we would read this specification as:

▷ a *program* is made up of *statements*;

▷ a collection of *statements* is empty or is a single *statement*, followed a semicolon, followed by a collection of *statements*;

▷ a statement is a *variable*, followed by an equals sign, followed by a *expression*;

▷ and so on...

The second rule highlights an interesting feature of BNF specifications: production rules are allowed to be *recursive*. The production rule for <stmts> allows us to expand <stmts> to a sentential form that *contains* the non-terminal symbol <stmts>. This means the above grammar recognizes infinitely many sentences.

Consider the following program in this language.

```
a = 1;
a = a + 2;
b = a;
```

```
<program>
<stmts>
<stmt> ; <stmts>
<var> = <expr> ; <stmts>
a = <expr> ; <stmts>
a = <term> ; <stmts>
a = <num> ; <stmts>
a = 1 ; <stmts>
a = 1 ; <stmt> ; <stmts>
a = 1 ; <var> = <expr> ; <stmts>
a = 1 ; a = <expr> ; <stmts>
a = 1 ; a = <term> + <term> ; <stmts>
a = 1 ; a = <var> + <term> ; <stmts>
a = 1 ; a = a + <term> ; <stmts>
a = 1 ; a = a + <num> ; <stmts>
a = 1 ; a = a + 2 ; <stmts>
a = 1 ; a = a + 2 ; <stmt> ; <stmts>
a = 1 ; a = a + 2 ; <var> = <expr> ; <stmts>
a = 1 ; a = a + 2 ; b = <expr> ; <stmts>
a = 1 ; a = a + 2 ; b = <term> ; <stmts>
a = 1 ; a = a + 2 ; b = <var> ; <stmts>
a = 1 ; a = a + 2 ; b = a ; <stmts>
a = 1 ; a = a + 2 ; b = a ;
```

Figure 3.6: Example derivation in the grammar in Figure 3.5

To verify that this program is recognized by the above grammar, we can write a (leftmost) derivation of it (Figure 3.6).

Remark 3.3.1. As a reminder, we're not interested in low-level syntactic concerns like whitespace when we consider whether or not a sentence is recognized by a grammar. The choice to present the program above in three lines was for readability, and the choice to present it in a single line in the derivation was for convenience.

Alternatively, we could construct a parse for the given program (Figure 3.7).

Exercise 3.3.1. Verify that

```
a = a + a;
b = b;
```

is recognized by the above grammar by giving both a derivation and a parse tree.

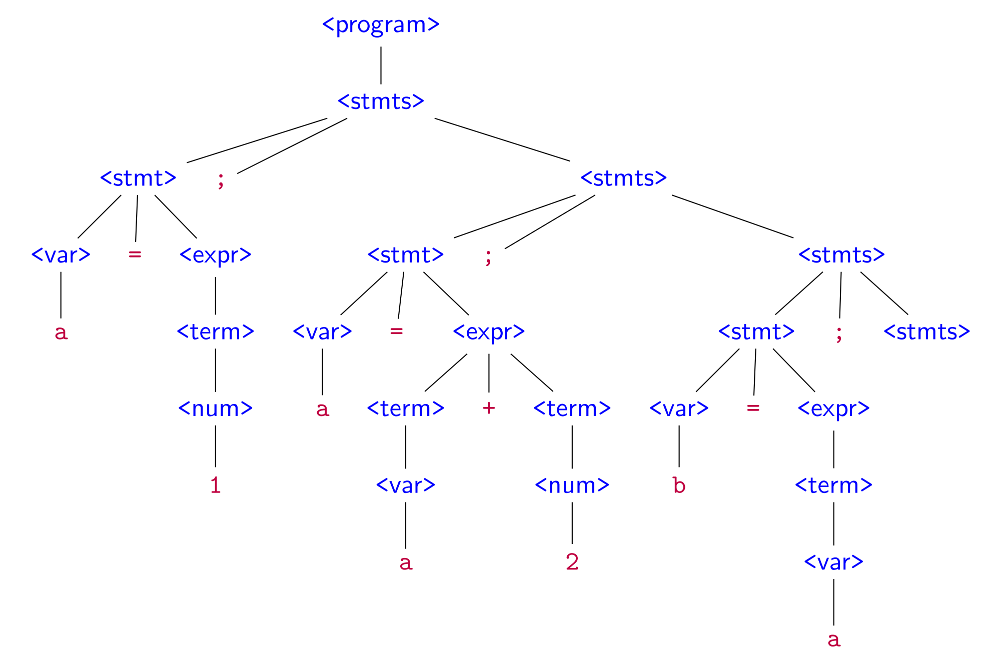


Figure 3.7: Example parse tree for the grammar in Figure 3.5.

$$
\begin{aligned}
\texttt{<expr>} &::= \texttt{<num>} \mid \texttt{<expr>}\ \texttt{<op>}\ \texttt{<expr>} \mid \texttt{(}\ \texttt{<expr>}\ \texttt{)} \\
\texttt{<op>} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\texttt{<num>} &::= \texttt{1} \mid \texttt{2} \mid \texttt{3}
\end{aligned}
$$

Figure 3.8: First attempt at a BNF specification for arithmetic expressions

Arithmetic Expressions

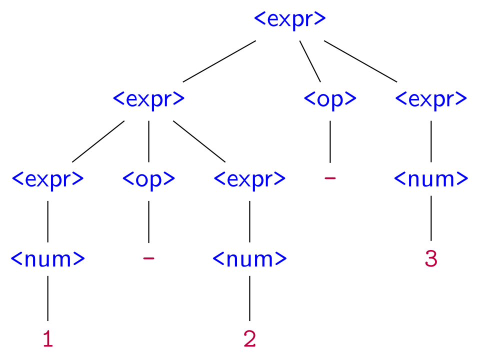
Figure 3.8 has a first attempt at a BNF specification for arithmetic expressions.

Exercise 3.3.2. Give a derivation and parse tree for the following sentence:

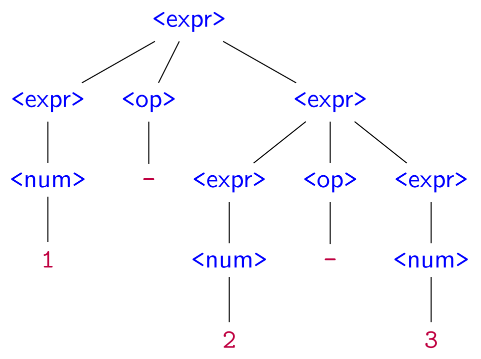
(1 + 2) * 3

using the grammar in Figure 3.8.

This grammar seems, ignoring obvious issues like having only three possible numbers, to be a reasonable enough specification: an expression is either a number or a pair of expressions with an operator between them. Note that the recursive nature of the first production rule means that this grammar recognizes infinitely many sentences. But this grammar has a curious feature: *there are sentences which have multiple parse trees.* Consider the sentence `1 - 2 - 3`. One the one hand, we might thing that this expression says "subtract 2 from 1, and then subtract 3." This reading is witnessed by the following parse tree:



But there's nothing in the structure of the expression itself which keeps us from intepreting it as "subtract from 1 the result of subtracting 3 from 2" which is witnessed by this alternative parse tree:



This ambiguity matters for the design of programming languages because it can change the *meaning* of the program itself; the first reading gives us the value -4 whereas the latter gives us 2. If we want to allow the programmer to write down an expression like `1 - 2 - 3`, then we have to be clear about what the intended ouput should be. More generally, we'd like to ensure that our grammar is *unambiguous* so there's no question as to how to read what the programmer has written. This will be our focus for the next section.

Remark 3.3.2. Before moving forward, I'd like to point out that BNF specifications are formal systems, as introduced in the previous chapter, written in an alternative form. From this perspective, judgments are of the form "$S \in$ <non-terminal>", where S is a sentence. We read a judgment as "the sentence S is a <non-terminal>." Non-terminal symbols thus define categories of sentences. Production rules define inference rules, which inform how sentences can be combined to produce new sentences of a given category. We won't make this formal, but will point to Figure 3.10 for the grammar in Figure 3.8 written as a formal system. Note also in the derivation in Figure 3.10 is, up to minor differences is labels, the same as a parse tree drawn (upside down) in derivation tree form.

3.4 Grammatical Ambiguity

As participants of language, we are no strangers to grammatical ambiguity. Take, for instance, the following sentence:[2]

```
John saw the man on the mountain with the telescope
```

[2]This example is taken from the Wikipedia page on Syntactic ambiguity.

$$\frac{}{\texttt{1} \in \texttt{<num>}}\ (\text{one}) \qquad \frac{}{\texttt{2} \in \texttt{<num>}}\ (\text{two}) \qquad \frac{}{\texttt{3} \in \texttt{<num>}}\ (\text{three})$$

$$\frac{}{\texttt{+} \in \texttt{<op>}}\ (\text{add}) \qquad \frac{}{\texttt{-} \in \texttt{<op>}}\ (\text{sub})$$

$$\frac{}{\texttt{*} \in \texttt{<op>}}\ (\text{mul}) \qquad \frac{}{\texttt{/} \in \texttt{<op>}}\ (\text{div})$$

$$\frac{S \in \texttt{<num>}}{S \in \texttt{<expr>}}\ (\text{num}) \qquad \frac{S \in \texttt{<expr>}}{S \in \texttt{(<expr>)}}\ (\text{paren})$$

$$\frac{S_1 \in \texttt{<expr>} \qquad S_2 \in \texttt{<op>} \qquad S_3 \in \texttt{<expr>}}{S_1\ S_2\ S_3 \in \texttt{<expr>}}\ (\text{op})$$

Figure 3.9: Formal system for the grammar in Figure 3.8

$$\frac{\dfrac{\dfrac{\dfrac{}{\texttt{1} \in \texttt{<num>}}}{\texttt{1} \in \texttt{<expr>}} \quad \dfrac{}{\texttt{-} \in \texttt{<op>}} \quad \dfrac{\dfrac{}{\texttt{2} \in \texttt{<num>}}}{\texttt{2} \in \texttt{<expr>}}}{\texttt{1 - 2} \in \texttt{<expr>}} \quad \dfrac{}{\texttt{-} \in \texttt{<op>}} \quad \dfrac{\dfrac{}{\texttt{3} \in \texttt{<num>}}}{\texttt{3} \in \texttt{<expr>}}}{\texttt{1 - 2 - 3} \in \texttt{<expr>}}$$

Figure 3.10: A derivation tree proving the recognition of `1 - 2 - 3` using the inference rules in

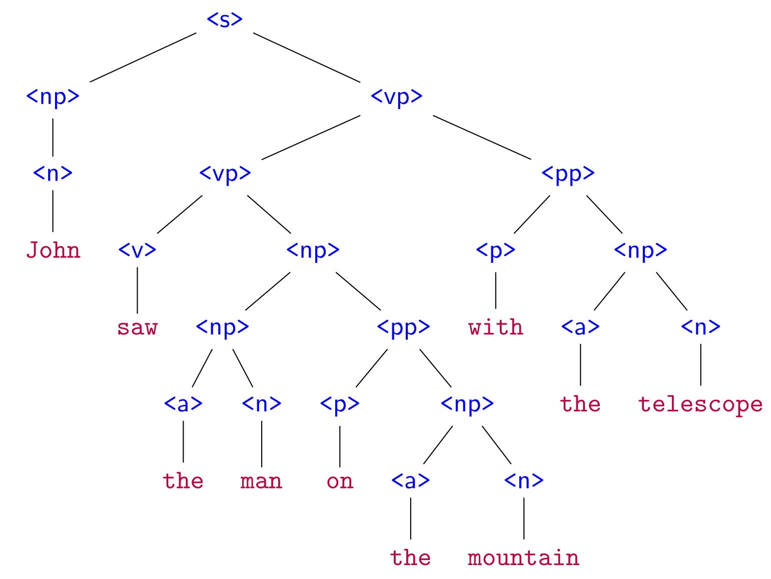


Figure 3.11: Parse tree of an ambiguous sentence for the grammar in Figure 3.13

Was John using the telescope to see the man on the mountain? Was the man carrying the telescope? Are there multiple mountains, one of which has a telescope on it? The ambiguity comes from it not being clear *which* hierarchical structure should scaffold the sentence. Figure 3.11 has one possible parse tree for this sentence in grammar based on English (Figure 3.13). The prepositional phrase "with the telescope" is grouped with the verb phrase starting with "saw", indicating that John was *using* the telescope. Figure 3.12 has an alternative parse tree for the same sentence. For this parse tree, the prepositional phrase "with the telescope" is grouped with the noun phrase "the mountain" indicating that there is a telescope on the mountain itself.

Remark 3.4.1. We experience language in a linear fashion, either by reading it or hearing it. Ifn our interlocutor could *display* the parse tree of their statement (floating eerily in space before our eyes) there would be nothing to say of (grammatical) ambiguity.

In another light, grammatical ambiguity is related to parentheses. If it were standard practice to use parentheses in natural language, we could avoid some grammatical ambiguity. For example, the parse tree in Figure 3.11 corresponds to:

```
John saw ( the man on the mountain ) with the telescope
```

whereas the parse tree in Figure 3.12 corresponds to:

```
John saw the man on ( the mountain with the telescope )
```

Ambiguity in natural language is a complex topic, but restricted to formal grammars, ambiguity is a well-defined notion.

Definition 3.4.1. A grammar is **ambiguous** if it recognizes a sentence with (at least) two distinct parse trees. Equivalently, it is ambiguous if it recognizes a sentence with (at least) two distinct leftmost derivations.

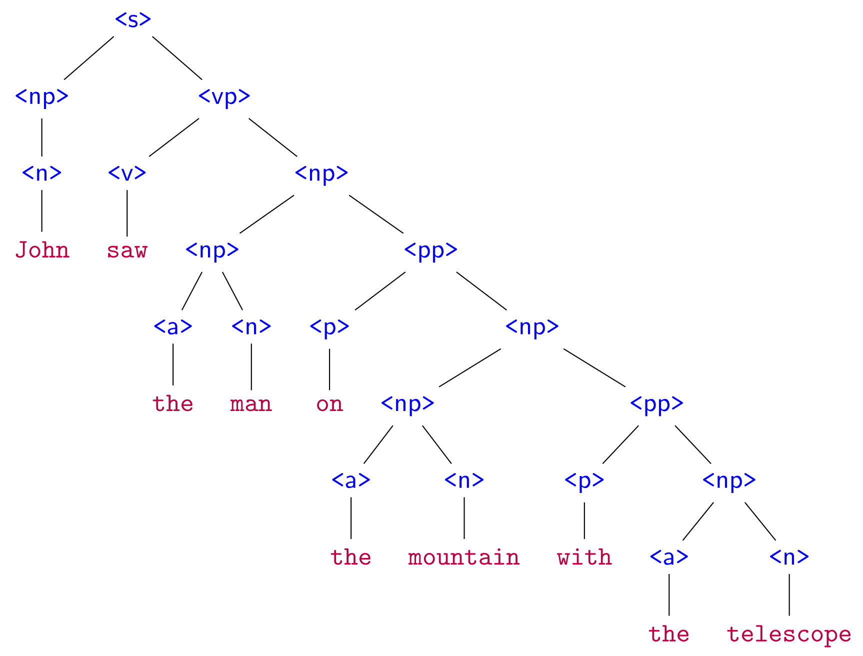


Figure 3.12: Another parse tree of an ambiguous sentence for the grammar in Figure 3.13.

Example 3.4.1. The grammar in Figure 3.13 is ambiguous. That is, in this grammar, our ambiguous sentence has multiple parse trees. Equivalently, by Theorem 3.2.1, it has multiple leftmost derivations. The grammar in Figure 3.8 is also ambiguous by virtue of the examples given in the previous section.

Exercise 3.4.1. Give a third distinct parse tree for the "John" sentence.

Exercise 3.4.2. Give two (or three) leftmost derivations of the "John" sentence.

Remark 3.4.2. In the previous example, along with many of the examples we'll see, it'll be fairly obvious that the grammar is ambiguous. As computer scientists, we might think that we could write program to determine if a grammar is ambiguous. Unfortunately, this is impossible (not just very difficult, but *impossible*). Formally speaking, this is to say that determining if a context-free grammar is ambiguous is *undecidable* [**?**].

Our next task it to determine how to *avoid* grammatical ambiguity when possible. But first, *why should we care?* Natural language is ambiguous and we get along perfectly fine, so why should we go through the trouble of making sure grammars we design for programming languages are unambiguous? The way I see it, it's a promise that to a programmer: *we never make unspoken assumptions about what you meant when we read your program.* We try to do this with natural language too, but in communication, if a statement is ambiguous, we can usually just ask our interlocutor what they meant. We can't do this for a program, so instead we make it *impossible* for a sentence to have multiple readings.[3]

[3]We see a similar phenomena in legal language, which tends to be grammatically sterile, and no fun to read.

```
 <s> ::= <np> <vp>
<vp> ::= <v> | <v> <np> | <vp> <pp>
<np> ::= <n> | <a> <n> | <np> <pp>
 <n> ::= John | man | mountain | telescope
 <v> ::= saw
 <a> ::= the
 <p> ::= on | with
```

Figure 3.13: BNF grammar that recognizes the ambiguous sentence in this section.

```
<expr> ::= <num> | <op> <expr> <expr>
  <op> ::= + | - | * | /
 <num> ::= 1 | 2 | 3
```

Figure 3.14: BNF specification for arithmetic expressions in Polish Notation

3.4.1 Fixity

If our only concern is avoiding ambiguity, we can do so by making sure operators appear *before* all their arguments. Figure 3.14 has a such a grammar for arithmetic expressions. This is called **Polish notation**, and always yields an unambiguous grammar, and *without any need for parentheses.*

It's not difficult to then guess what reverse polish notation is: operators appear *after* their arguments. This is how early calculators like the HP 9100A Desktop Calculator were designed. If you wanted to calculate something like `(2 + 3) * (4 - 5)`, you would push the operands onto a *stack* and then apply operators to the top elements of the stack:

$$\varnothing \xrightarrow{2} [2] \xrightarrow{3} [2,3] \xrightarrow{+} [5] \xrightarrow{4} [5,4] \xrightarrow{5} [5,4,5] \xrightarrow{-} [5,-1] \xrightarrow{*} [-5]$$

so that the expression you typed was in reverse Polish notation, in this case:

`2 3 + 4 5 - *`.

Exercise 3.4.3. Derive the sentence `+ * 1 * 2 - 3 2 1` using the grammar in Figure 3.14.

The benefit of Polish notation is that it is extremely simple, and easy to parse. The issue with Polish notation is that it's difficult to read. Imagine having to write code with a prefix form of if-then-else. OCaml code of the form:

```
if n < 10
then -1
else if n > 10
```

```
<expr> ::= <num> | ( <expr> <op> <expr>)
  <op> ::= + | - | * | /
 <num> ::= 1 | 2 | 3
```

Figure 3.15: BNF specification for arithmetic expressions using many parentheses

```
then 1
else 0
```

becomes something like:

```
if_then_else < n 10
  -1
  if_then_else > n 10
    1
    0
```

which isn't *so* bad, but is clearly less readable.[4] This is all to say that ambiguity in expressions is in part due to issues of operator **fixity**.

Definition 3.4.2. The **fixity** of an operator refers to where the syntactic components of an operator are placed relative to its arguments. There are four kinds of operator fixity.

- A **prefix** operator appears before its arguments. In the expression `-5` the negation operator `-` appears before its argument `5`.
- A **postfix** operator appears after its arguments. In the expression `10!` the factorial operator `!` appears after its argument `10`.
- A **infix** operator appears between its arguments. Infix operators are required to be binary. In the expression `4 + 3` the binary addition operator `+` appears between its arguments `4` and `3`.
- A **mixfix** operator has multiple syntactic components which appear in some combination of before, after, and in between its arguments. In the expression `if x < 0 then -x else x` the syntactic components `if`, `then`, and `else` appear in various positions with respect to the arguments of the if-then-else operator.

The above discussion indicates that if we're willing to accept *only* prefix operators or *only* postfix operators, then it's easy to design unambiguous grammars. But if we want to contend with multiple kinds of operator fixity, then we'll have to confront the issues of grammatical ambiguity.

3.4.2 Parentheses

Another solution to the ambiguity problem is to surround *every* application of an operator with parentheses. Figure 3.15 has such a grammar for arithmetic expressions.

[4]There are cases in which the simplicity trumps readability, e.g., for dialects of Forth.

Exercise 3.4.4. Derive the sentence `((1 * (2 * (3 - 2))) + 1)` using the grammar in Figure 3.15.

The issue with this approach is that it means our expressions has a lot of parentheses. This can be frustrating, but in reality it's not so bad. Note that, if we parenthesize everything, the fixity of operators becomes irrelevant; the parentheses play entirely the role of disambiguating multiple applications of operators. Furthermore, if we design our grammar so that all operators are prefix, then we get **S-expressions**. This might feel like piling one hard-to-read syntactic paradigm onto another, but there are quite a few programming languages whose surface-level syntax is made up of S-expressions.[5] And the benefits are vast: S-expressions are dead-simple to parse, which makes for quick programming language development. They also allow for some fancier features we won't go into, but make for convenient macro systems in languages like Lisp and Scheme.[6]

Remark 3.4.3. One common conceptual roadblock is in recognizing that *parentheses are part of syntax*. We're trained to think of parentheses as meta-syntax, included only to make things more clear, but never explicitely considered (e.g., when you learned calculus, you likely didn't have a lesson on "the use of parentheses in analytic expressions"). But we're on "the other side" so to speak, meaning we're the ones who have to specify exactly (i.e., formally) how parentheses work. In other words, we don't get parentheses "for free," we have to implement them as part of the syntax of the language.

All told, our basic question becomes: *how do we avoid grammatical ambiguity while being able to mix operator fixity and not use so many parentheses?* And this question has a simple answer in theory: *make explicit assumptions about how operator arguments should be grouped.* This will mean contending with two things: **associativity** and **precedence**.

3.4.3 Associativity

Associativity refers to how arguments are grouped when we're given a sequence of applications of an infix operator in the absence of parentheses. For example, the expression:

```
1 + 2 + 3 + 4
```

can be understood as any one of the following:

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

In the case of addition, the point is somewhat moot. The order in which we group arguments doesn't affect the *value* of a sequence of additions. That is, addition is **associative**.

Definition 3.4.3. An operation $\circ : X \to X \to X$ is **associative** if

$$(a \circ b) \circ c = a \circ (b \circ c)$$

for all a, b, and c in X.

[5] I understand this might not be a compelling argument to the parentheses-intolerant among us (of which there are far too many) but it at least indicates that the multitude of parentheses is an aesthetic *inclination*, and not just a quirk of the bizarre and aged.

[6] If you're interested, look up the term *homoiconicity*.

But not all operators are associative, e.g., subtraction. We need to *decide* how to implicitly parenthesize an expression like `1 - 2 - 3 - 4`. For binary operators, we typically choose one of the following two ways of grouping terms.

Definition 3.4.4. An operator $\circ : X \to X \to X$ is said to be **left-associative** if sequences of applications of the operator are understood as grouping arguments from left to right, i.e.,

$$a \circ b \circ c \circ d = (((a \circ b) \circ c) \circ d)$$

for any a, b, c, and d, in X. It's said to be **right-associative** if arguments are grouped from right to left, i.e.,

$$a \circ b \circ c \circ d = (a \circ (b \circ (c \circ d)))$$

Bringing this back to grammatical ambiguity, giving an implicit parenthesization of a sequence of operators means specifying a "shape" for the corresponding parse tree. Taking subtraction to be left-associative means that one of the two parse trees for the sentence `1 - 2 - 3` in the previous section is correct (the first one) and the other is not (the second one).

Exercise 3.4.5. The following is a grammar for function types over the base types `int` and `bool`.

```
 <fun-type>  ::=  <base-type> | <fun-type> -> <fun-type> | ( <fun-type> )
<base-type>  ::=  int | bool
```

Give two leftmost derivation of `int -> bool -> int`.

All of this rests on the following mathematical fact about the formal grammars.

Definition 3.4.5. A symbol X is **useful** in a BNF grammar $\mathcal{G}$ if there is a derivation of a sentence S in $\mathcal{G}$ which uses the symbol X.

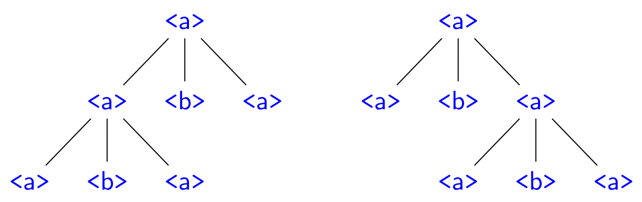
Proposition 3.4.1. Let $\mathcal{G}$ be a BNF grammar and let `<a>` and `<b>` be arbitrary nonterminal symbols which are useful in $\mathcal{G}$. Further suppose that

```
<a>  ::=  <a> <b> <a>
```

is a production rule of $\mathcal{G}$. Then $\mathcal{G}$ is ambiguous.

Exercise 3.4.6. (Challenge) Prove Proposition 3.4.1.

Generally speaking we should be wary of production rules of the form given in Proposition 3.4.1, because it's possible to construct subtrees along the lines of the ones given for subtraction above:



```
       <expr> ::= <expr> <op> <expr-no-op> | <expr-no-op>
<expr-no-op> ::= <num> | ( <expr> )
         <op> ::= + | - | * | /
        <num> ::= 1 | 2 | 3
```

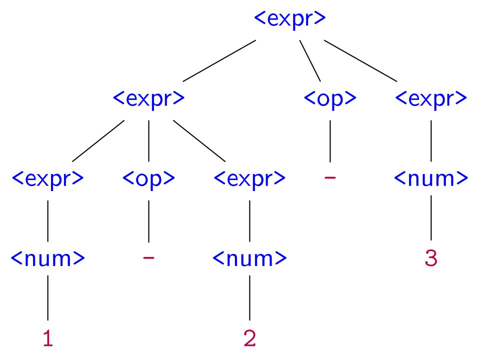
Figure 3.16: BNF specification for arithmetic with left-associative operators

Proposition 3.4.1 implies that a grammar for arithmetic will be ambiguous as long as it contains the rule

```
<expr> ::= <expr> <op> <expr>
```

as the grammar in Figure 3.8 does. The question remains: *what do we do about this?* According to the discussion above, this is in part an issue of associativity. The problem arises from the fact that we can expand *both* the left and the right argument of an operator to be another application of a binary operator.

The first step is to specify an associativity assumption for binary operators. It's typical to take addition, subtraction, multiplication and division to be *left associative*, e.g., this is the "correct" parse tree for `1 - 2 - 3`:



Another way of expressing left-associativity is that the *right* argument of an operator can't be another application of the operator. In order to enforce left-associativity of operators, we need to "factor out" the application of a binary operator. Said another way, we need to create a *new* nonterminal symbol `<expr-no-op>` which is the same as `<expr>` but which can't be expanded to an application of a binary operator. This is done for the grammar in Figure 3.16.

Exercise 3.4.7. Draw parse trees for the following sentences using the grammar in Figure 3.16.

```
1 - 2 - 3
1 * (2 - 3)
1 + 2 * 3
```

Proposition 3.4.2. The grammar in Figure 3.16 is unambiguous.

Exercise 3.4.8. (Challenge) Prove Proposition 3.4.2.

Exercise 3.4.9. Rewrite the grammar in Exercise 3.4.5 so that it is unambiguous and the function type arrow is taken to be *right* associative.

3.4.4 Precedence

We're getting closer to a grammar that captures our intuitions about arithmetic expressions. We've succeeded in designing a grammar that is not ambiguous, but as is demonstrated in Exercise 3.4.7, we've failed to enforce an aspect of arithmetic expressions that we learn in grade school: `1 + 2 * 3` should be equivalent to `1 + (2 * 3)` and not `(1 + 2) * 3`. This is an issue of **precedence** or **order of operations**, something that you're likely already familiar with. And like associativity, incorrectly dealing with precedence can affect the output value of a program.

If you went through the American public school system then you probably learned the abbreviation PEMDAS (Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction) along with an accompanying mnemonic, something like "please excuse my dear aunt sally." Focusing on just the last four letters, this rule tells us that we should group multiplications and divisions first, and then group additions and subtractions. That is to say, multiplication and division have greater **precedence** than addition and subtraction.

Definition 3.4.6. The **precedence** of an operator, relative to another operator, determines which operator binds more tightly, in the presents of ambiguity.

Like associativity, the relative precedence of a collection of operators determines the shape of the parse tree. To say that multiplication has higher precedence is to say that when we build the parse tree for `1 * 2 + 3`, the operator "`+`" should be the top-level operation, i.e., at the root of the tree.

Remark 3.4.4. One thing that was probably glossed over if/when you learned PEMDAS: *what do you do with something like `1 + 2 - 3 + 4`?* Do you group additions and then subtractions? Or vice versa? The issue here is that addition and subtraction have the *same* precedence (something which is not made clear by PEMDAS). In this case, we'll use the associativity of the operators to determine how to parenthesize, i.e., given a sequences of operators of the same precedence, we use their associativity to group them.

Since addition and subtraction are both left-associative, the above expression should be equivalent to `((1 + 2) - 3) + 4`. Things get complicated if we have two operators with the same precedence, but *different* associativity. We'll ignore this possibility, but this matters in languages like Haskell, where users can define their own operators with specified precedence and associativity.

How do we deal with precedence? In the same way that as we dealt with associativity: by "factoring out" the operators of higher precedence. Another way of expressing that "`*`" has higher precedence than "`+`" is to say that the arguments of "`*`" can't be applications of "`+`" (of course we can still put additions in parentheses). Said another way, we need to create a new nonterminal symbol `<expr-no-pm>` which is the same as `<expr>` but which can't be expanded to an addition or subtraction. This is done for the grammar in Figure 3.17. Note that, in this grammar, we also maintain the asymmetry introduced to deal with associativity.

Figure 3.18 has a parse tree for the sentence `1 + 2 * 3` in this grammar. Naturally the tree is "noisier" in that there's more intermediate nonterminal symbols, but hopefully the point is clear: we have to expect with addition first because otherwise, we'd get stuck. That is, if we expand the multiplication first, the grammar disallows us expanding the left argument to `1 + 2`.

Exercise 3.4.10. Give a derivation of `(1 + 2) * 3` using the grammar in Figure 3.17

```
       <expr>  ::=  <expr> <pm> <expr-no-pm> | <expr-no-pm>
 <expr-no-pm>  ::=  <expr-no-pm> <md> <expr-no-op> | <expr-no-op>
 <expr-no-op>  ::=  <num> | ( <expr> )
         <pm>  ::=  + | -
         <md>  ::=  * | /
        <num>  ::=  1 | 2 | 3
```

Figure 3.17: BNF specification for arithmetic with left-associative operators

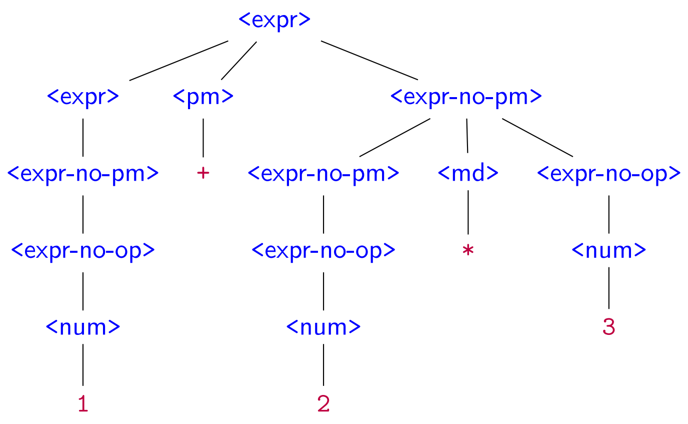


Figure 3.18: Parse tree of `1 + 2 * 3` using the grammar in Figure 3.17

3.4.5 Operator Tables

In the next chapter, we'll see how to implement parsers for a given grammar. One way is to use a **parser generator**, which is a program that generates the code for a parser based on a specification. More on that later, the point: most parser generators don't require us to use the techniques above to deal with associativity and precedence. Rather, they take additional about associativity and precendence directly, often in the form of an **operator table**. For arithmetic, we might use the following table, which presents operators in order of increasing precedence.

Operator	Associativity
`+`, `-`	left
`*`, `/`	right

Note that, in light of Remark 3.4.4, it's reasonable to assume that all operators of the same precedence (i.e., appearing in the same row) have the same associativity. The OCaml Manual has an analogous table for its operators.

Remark 3.4.5. We can think of this as an *a posteriori* approach to disambiguation; rather than designing a grammar in which every sentence has a unique parse tree, we design an ambiguous grammar and then specify which of the parse trees we consider acceptable. As long as the only source of ambiguity is due to associativity and precedence of binary operators, operator tables allow use to associate with every sentence a unqiue parse tree.

A final word of warning for this section: as much as we've tried to present grammars as a rigorously as possible, they tend to be a bit handwavy in practice. For example, the following is a standard presentation of arithmetic expressions in a more mathematical setting:

$$e ::= e + e \mid e - e \mid e * e \mid e/e \mid \mathbb{Z}$$

where $\mathbb{Z}$ refers to any integer. Operator associativity and precedence are understood from context and the production rule:

$$e ::= (e)$$

is implicitly assumed, contrary to Remark 3.4.3! In other settings, (e.g., the full grammar specification of Python) the notation used is a mixture of multiple syntaxes for grammar specifications. This is just to say, there is a difference between grammars as they are studied in formal language theory, and as they are used to specify programming languages or used to specify syntaxes in mathematical settings like logic and type theory. In the later cases, you'll likely have to read between the lines, so to speak.

3.5 Extended BNF

There are several extensions of the BNF meta-syntax which make it more usable. We'll consider a small collection of extensions which capture common patterns of programming language syntax. We'll call BNF plus these extensions **extended BNF**, or **EBNF** for short. In particular, we'll use BNF if we intend to exclude the following extensions to the meta-syntax.

Options

Definition 3.5.1. We use the notation [SENT-FORM] for parts of a production rule which are optional. That is, we write

$$\texttt{<nt>} ::= \text{SENT-FORM}_1\ [\text{SENT-FORM}_2]\ \text{SENT-FORM}_3$$

as shorthand for

$$\begin{aligned} \texttt{<nt>} ::= {} & \text{SENT-FORM}_1\ \text{SENT-FORM}_3 \\ \mid {} & \text{SENT-FORM}_1\ \text{SENT-FORM}_2\ \text{SENT-FORM}_3 \end{aligned}$$

Example 3.5.1. We can use the EBNF production rule

$$\texttt{<if-expr>} ::= \texttt{if}\ \texttt{<expr>}\ \texttt{then}\ \texttt{<expr>}\ [\texttt{else}\ \texttt{<expr>}]$$

to write a grammar whose expressions allow for both if-then statements and if-then-else statements.

Exercise 3.5.1. Rewrite the following EBNF production rule as a collection of BNF production rules.

$$\texttt{<a>} ::= \texttt{a}\ [\texttt{<b>}]\ [\texttt{a}]$$

Remark 3.5.1. One issue with extending BNF syntax is that it's more difficult to express grammars which include symbosl used for the extensions (e.g, "[" and "]"). In practice this is not a huge problem, we'll try to be explicit (e.g., by using color) in distiguishing between symbols in the meta-language and terminal symbols in the grammar.

Alternatives

Definition 3.5.2. We use the notation $(\text{SENT-FORM}_1 \mid \ldots \mid \text{SENT-FORM}_k)$ for parts of a production rule which can be one of several. This is like alternative syntax, but it allows us to embed the alternatives within the production rule. We write

$$\texttt{<nt>} ::= \text{ST-FM}_0\ (\text{ST-FM}_1 \mid \ldots \mid \text{ST-FM}_k)\ \text{ST-FM}_{k+1}$$

as shorthand for

$$\begin{aligned} \texttt{<nt>} ::= {} & \text{ST-FM}_0\ \text{ST-FM}_1\ \text{ST-FM}_{k+1} \\ & \vdots \\ \mid {} & \text{ST-FM}_0\ \text{ST-FM}_k\ \text{ST-FM}_{k+1} \end{aligned}$$

Example 3.5.2. We can simplify the grammar in Figure 3.8 by removing the nonterminal symbol `<op>` and putting the alternative binary operators within a single rule:

$$\texttt{<expr>} ::= \texttt{<expr>}\ (\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/})\ \texttt{<expr>}$$

$$
\begin{aligned}
\texttt{<e>} &::= \texttt{<e-no-pm>}\ \{(\texttt{+} \mid \texttt{-})\ \texttt{<e-no-pm>}\} \\
\texttt{<e-no-pm>} &::= \texttt{<e-no-op>}\ \{(\texttt{*} \mid \texttt{/})\ \texttt{<e-no-op>}\} \\
\texttt{<e-no-op>} &::= \texttt{<num>} \mid \texttt{(}\ \texttt{<e>}\ \texttt{)} \\
\texttt{<num>} &::= \texttt{1} \mid \texttt{2} \mid \texttt{3}
\end{aligned}
$$

Figure 3.19: EBNF specification for arithmetic

Exercise 3.5.2. TODO

Repetition

Definition 3.5.3. We use the notation {SENT-FORM} for part of a production rule which can be repeated as many times as we want. That is, we write

$$\texttt{<nt>} ::= \text{ST-FM}_0\ \{\text{ST-FM}_1\}\ \text{ST-FM}_2$$

as shorthand for

$$
\begin{aligned}
\texttt{<nt>} &::= \text{ST-FM}_0\ \text{ST-FM}_2 \\
&\mid \text{ST-FM}_0\ \text{ST-FM}_1\ \text{ST-FM}_2 \\
&\mid \text{ST-FM}_0\ \text{ST-FM}_1\ \text{ST-FM}_1\ \text{ST-FM}_2 \\
&\qquad\vdots
\end{aligned}
$$

The previous definition is an abuse of notation. Even though we didn't strictly enforce it, we tend to think of grammars as having a finite number of production rules. We can't give a simple translation into BNF production rules as we did for the previous extensions because such a translation would require use to make a choice: *do we repeat to the left or the right?* In other words, is

$$\texttt{<nt>} ::= \text{ST-FM}_0\ \{\text{ST-FM}_1\}\ \text{ST-FM}_2$$

shorthand for

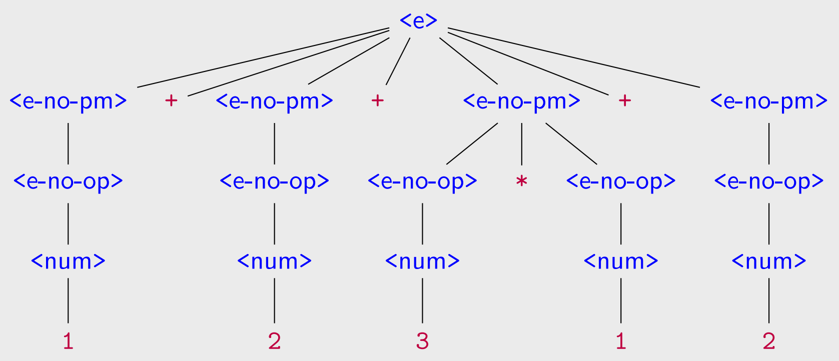
$$
\begin{aligned}
\texttt{<nt>} &::= \text{ST-FM}_0\ \texttt{<reps>}\ \text{ST-FM}_2 \\
\texttt{<reps>} &::= \varepsilon \mid \text{ST-FM}_1\ \texttt{<reps>}
\end{aligned}
$$

or

$$
\begin{aligned}
\texttt{<nt>} &::= \text{ST-FM}_0\ \texttt{<reps>}\ \text{ST-FM}_2 \\
\texttt{<reps>} &::= \varepsilon \mid \texttt{<reps>}\ \text{ST-FM}_1
\end{aligned}
$$

Rather than fixing a choice, we treat a repetition as being expandable to any number of its corresponding sentential form.

Example 3.5.3. Figure 3.19 has a grammar which takes advantage of these extensions to BNF in the case of arithmetic expressions. Note that we lose the ability to express that the operators are left-associative because the repetitions are expanded in a single step. Take, for example the following derivation of `1 + 2 + 3 * 1 + 2`:



Because the operator applications are collapsed to a single level the left associativity of the operators needed to be dealt with in a different way. We'll see how this is done in the next chapter.

Remark 3.5.2. These extensions of BNF syntax are strictly *meta*-syntactic. The notion of a derivation or a parse tree does not change. In particular, as in Example 3.5.3, *EBNF syntax should never appear in a derivation or a parse tree.*

Exercise 3.5.3. TODO

3.6 Regular Expressions

The last topic of formal language which we'll take on is that of **regularity**. As we'll see, regular grammars play a key role **lexical analysis** the first step of our interpretation pipeline.

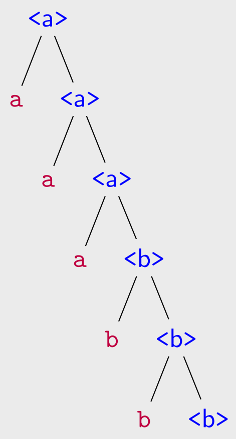
Definition 3.6.1. A **(right) regular grammar** is one in which every production rule is of the following form:

1. `<n>` ::= `t <m>`
2. `<n>` ::= `t`
3. `<n>` ::= ε

Example 3.6.1. The following is a regular grammar.

```
<a> ::= a <a>
<a> ::= a <b>
<b> ::= b <b>
<b> ::= ε
```

This grammar recognizes the sentence `aaabb` and has the following parse tree.



Regular grammars are incredibly simple. The parse trees for the sentences they recognize are always skewed, as in Example 3.6.1. This simplicity makes it easy to write a naive algorithm for determining if a sentence is recognized by a regular grammar. We can interpret the skewedness as visually demonstrating that every rule in a grammar "makes forward progress" towards recognising a sentence. This means that a basic backtracking approach suffices for the recognition algorithm. We begin by setting up some types.

```
type nonterminal = NT of char
type terminal = T of char

type rule_rhs =
  | Empty
  | Term of terminal
  | Continue of terminal * nonterminal

type rule = nonterminal * rule_rhs
type sentence = terminal list
```

Then, for a given sentence, we find a rule which matches its first character on the right-hand-side, and recurse on the remainder of the sentence as necessary.

```
let recognizes
    (start : nonterminal)
    (rules : rule list)
    (sent : sentence) : bool =
  let rec go start = function
    | [] -> List.exists ((=) (start, Empty)) rules
    | t :: ts ->
      let matches (nterm, rule) =
        match rule with
        | Empty -> false
        | Term term -> term = t && List.is_empty ts
        | Continue (term, next)
            when nterm = start
                 && term = t ->
            go next ts
      in
      List.exists matches rules
  in
  go start sent
```

Regular grammars are surprisingly expressive. They can recognize a fairly wide range of useful languages. But in a sense, regular grammars are *too* simple. Specifying the regular grammars for "useful" lanuages can be complicated. **Regular expressions** are an alternative to BNF specifications for regular grammars with a compact and natural syntax. And they maintain the feature that its possible to write simple recognition algorithsm for regular expressions.

Definition 3.6.2. The set of regular expressions over the terminal symbols $\mathcal{T}$ are defined inductively as follows:

- ▷ (empty) ε is a regular expression;
- ▷ (terminal) if `t` $\in \mathcal{T}$ then `t` is a regular expression;
- ▷ (repetition) if e is a regular expression, then e^* is a regular expression;
- ▷ (alternatives) if $e_1, \dots, e_k$ are regular expressions, then $(e_1 \mid \dots \mid e_k)$ is a regular expression;
- ▷ (concatenation) if $e_1, \dots, e_k$ are regular expressions, then $e_1 \dots e_k$ is a regular expression;

Definition 3.6.3. ▷ $e^?$ is shorthand for $(\varepsilon \mid e)$

- ▷ e^+ is shorthand for $e\ e^*$

Definition 3.6.4. TODO

$$\frac{}{\varepsilon \lhd \varepsilon}\ \text{(empty)}$$

$$\frac{\boxed{\mathtt{t} \in \mathcal{T}}}{\mathtt{t} \lhd \mathtt{t}}\ \text{(terminal)}$$

$$\frac{e \lhd S_1 \qquad \ldots \qquad e \lhd S_k}{e^* \lhd S_1 \ldots S_k}\ (\text{repetition}_k)$$

$$\frac{e_i \lhd S_i}{(e_1 \mid \ldots \mid e_k) \lhd S_i}\ (\text{alternatives}_i)$$

$$\frac{e_1 \lhd S_1 \qquad \ldots \qquad e_k \lhd S_k}{e_1 \ldots e_k \lhd S_1 \ldots S_k}\ (\text{concatenation}_k)$$

Example 3.6.2. The regular expression $\mathtt{a}^+\mathtt{b}^*$ recognizes `aaabb`.

$$\dfrac{\dfrac{\mathtt{a} \lhd \mathtt{a} \qquad \dfrac{\mathtt{a} \lhd \mathtt{a} \qquad \mathtt{a} \lhd \mathtt{a}}{\mathtt{a}^* \lhd \mathtt{aa}}\ \text{(rep)}}{\mathtt{aa}^* \lhd \mathtt{aaa}}\ \text{(concat)} \qquad \dfrac{\mathtt{b} \lhd \mathtt{b} \qquad \mathtt{b} \lhd \mathtt{b}}{\mathtt{b}^* \lhd \mathtt{bb}}\ \text{(rep)}}{\mathtt{aa}^*\mathtt{b}^* \lhd \mathtt{aaabb}}\ \text{(concat)}$$

Theorem 3.6.1. Let $\mathcal{G}$ be a regular grammar. There is a regular expression e such that for any sentence S, the grammar $\mathcal{G}$ recognizes S if and only if $e \lhd S$.

Theorem 3.6.2. Let e be a regular expression. There is a regular grammar $\mathcal{G}$ such that for any sentence S, the grammar $\mathcal{G}$ recognizes S if and only if $e \lhd S$.

3.7 Exercises

Chapter 4

Parsing

4.1 General Parsing

4.2 Lexical Analysis

4.3 Recursive-Descent

4.4 Parser Combinators

4.5 Parser Generators

Chapter 5

Operational Semantics

Write section of formal semantics

Chapter 6

Evaluation

Write section of evaluation.

Chapter 7

Type Systems

Write section on type systems.

Chapter 8

Type Checking

write section on type checking

Chapter 9

Type Inference

Write section on type inference.

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 $T_1, \dots, T_k$ are trees then so is $\mathsf{node}(v, T_1, \dots, T_k)$.

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:

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

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

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

[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:

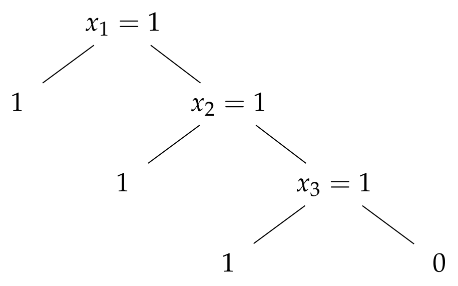
$$\mathsf{node}(x_1 = 1, \mathsf{node}(1), \mathsf{node}(x_2 = 1, \mathsf{node}(1), \mathsf{node}(x_3 = 1, \mathsf{node}(1), \mathsf{node}(0))))$$

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.

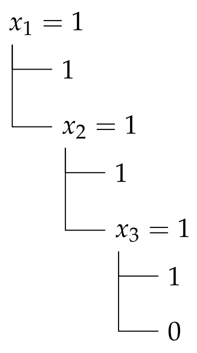
We can naturally define rose trees in OCaml, which means we can also define the usual functions on trees:

```
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 $T_1, \dots, T_k$, if we assume that P holds of each tree $T_1, \dots, T_k$[4] then we can demonstrate that P holds of* $\mathsf{node}(v, T_1, \dots, T_k)$;
>
> *then P holds of all trees with values from V.*

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 $T_1, \dots, T_k$ from $\mathcal{Q}$, if Q holds of $(v, \mathsf{root}\,(v_1), \dots, \mathsf{root}\,(v_k))$ then $\mathsf{node}(v, T_1, \dots, T_k) \in \mathcal{Q}$.

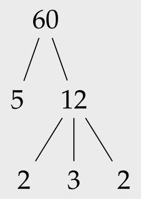
[3] If you're interested, look up the command line tool `tree`, or just type `tree .` in your terminal and see what you get.

[4] Again, this assumption is called the *induction hypothesis*.

[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.

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.

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

[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...).

Chapter 2, is an inductively-define subset of trees. The property is straightforward: it holds of the nonempty sequence of judgments $(J, J_1, \dots, J_k)$ when

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

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 $J_1, \dots, J_k$, if we assume that P holds of each judgment $J_1, \dots, J_k$ where*

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

  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 $n \in \mathbb{N}$.

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 **??**.

Appendix B

320Caml Specification

Include 320Caml spec.

Bibliography

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers: Principles, Techniques & Tools*. Pearson Education, 2007.

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

[3] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[4] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[5] Norman Ramsey. *Programming Languages: Build, Prove, and Compare*. Cambridge University Press, 2022.

[6] Michael Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.