Chapter 1

Introduction

1.1 Model Programming Languages

The mathematical analysis of programming languages begins with the formulation of "model" programming languages. For example, if we want to analyze procedure call mechanisms, we might begin by identifying a simple programming language whose main constructs are procedure declarations and calls. We may then analyze this simplified programming language without worrying about irrelevant details of a larger programming language. This process is not only useful in analyzing an existing programming language, but also in designing a new one. Since practical programming languages are typically large and very complex, programming language design involves careful and separate consideration of various sublanguages. Of course, it is important to keep in mind that a small language with only a few constructs may give false impressions. We might conclude that a programming language is much simpler than it really is, or we might rely on properties that are immediately destroyed when important features are added. Therefore, good taste and careful judgement are required. In developing a programming language theory, or applying theoretical analysis to practical situations, we must always keep in mind the nature of our simplifying assumptions and how they may affect the conclusions we reach.

In this book, we will will study programming language concepts using the framework of typed lambda calculus. The idea that lambda calculus is the basic mechanism underlying function definition and naming conventions in programming languages is an old one. It is described in an early paper of Landin [Lan66], for example, which suggests that whatever the next 700 programming languages turn out to be, they will surely be variants and extensions of lambda calculus. By adding to basic typed lambda calculus in various ways, we may devise model languages with a variety of historical, contemporary, or forward-looking features. An advantage of sticking to typed lambda calculus is that there is a certain degree of "modularity" to our theory. Although there are exceptions, many of the extensions to typed lambda calculus may be combined without unexpected synergistic effects. For example, after exploring polymorphism and records separately, we can easily formulate a language with polymorphism and records and identify many of its properties.

The main goals in our investigation of programming language concepts and features are to see beyond the surface syntax and understand the meanings of program phrases (expressions, commands, declarations, etc.) at an appropriate level of detail. Since it is impossible to select a single "appropriate level of detail" that is appropriate for all forms of programming and program analysis, we concentrate on general techniques that may be used in a variety of ways to abstract away from some properties of programs and focus on others.

The first chapter beyond this introduction is concerned with the syntax, operational semantics and programming capabilities of a simple, illustrative language for *Programming Computable Functions,* called PCF. This language is based on typed lambda calculus, with natural numbers and booleans (truth values) as basic types. PCF allows us to form pairs of values, and define recursive functions. Later in the book, we study the pure typed lambda calculus separately, as well as additional data types for PCF such as stacks and trees. Since the type system of PCF is not as flexible as we might want, we will consider systems with polymorphic functions and data type declarations. This brings us to languages that resemble and extend ML [GMW79, Mil85a] and Miranda [Tur85]. In a later chapter, we consider more flexible type systems based on subtyping, concluding with a chapter on type inference. Throughout, we rely on the basic mechanisms of lambda abstraction (for defining functions) and function application.

Organization of this book

Each chapter of this book begins with a brief introductory section. Each introductory section contains a list of the main topics of that chapter. To give an example, and to warn the reader of what is ahead, here is a list of the main topics covered in this chapter:

- Overview of lambda notation and the system of lambda calculus.
- Brief discussion of types and type systems.
- List of standard mathematical notation used in this book.
- Background discussion on set theory.
- Background discussion on syntax and semantics, including a brief overview of grammars and parsing.
- Background discussion on induction, covering induction on the natural numbers, structural induction on expressions, structural induction on proofs, and well-founded induction. In discussing induction on proofs, we also review basic terminology and properties of formal proof systems.

Each of the background sections are intended to be used as reference for later chapters of the book.

Like most lists in later chapters, the topics listed here correspond closely to the subsections of the chapter that are listed in the Table of Contents. Therefore, if you wish to find the reference section on standard mathematical notation, for example, you may turn to the Table of Contents and look for the subsection entitled, "Notation and Mathematical Conventions." (You will find that this is Section 1.5)

1.2 Lambda Notation

Lambda calculus has proven useful in describing, analyzing and implementing programming languages. With a little practice, the reader may become familiar enough with the notation to see C, Pascal or Ada program phrases as syntactic variants of lambda expressions. This will make the theory described in this book more useful, and also make it much easier to understand a variety of programming languages. The language PCF will use lambda notation directly, as do the programming languages Lisp [Ste84] and Scheme [AS85, SS75]. However, the reader familiar with Lisp should be warned that PCF does not have the complete "look and feel" of a Lisp-like language.

There are no lists or atoms, there is a relatively rigid compile-time typing discipline, and the order of evaluation is different.

The primary features of lambda notation are *lambda abstraction,* which we use to write function expressions, and *application,* which allows us to make use of the functions we define. Expressions written in lambda notation are called *lambda expressions* or *lambda terms.* Lambda notation is used in both typed and untyped systems of lambda calculus. A comprehensive treatment of untyped lambda calculus may be found in [Bar84]. Some comments on the relationship between typed and untyped languages appear in Section 1.4.

In *typed* lambda calculus, the domain of a function is specified by giving a type to the formal parameter. If M is some expression that is well formed under the assumption that the variable x has type σ, then $\lambda x{:}\, \sigma.\, M$ defines the function mapping any x in σ to the value given by M. A simple example is the lambda expression

$$\lambda x{:}\, nat.\, x$$

for the identity function on natural numbers. The notation "$x{:}\, nat$" specifies that the domain of this function is nat, the type of natural numbers. Since the function value at $x{:}\, nat$ is x, the range is also nat. For each form of typed lambda calculus, there are precise rules saying which expressions are well formed under assumptions about the types of variables. These rules also tell us how to determine the range of $\lambda x{:}\, \sigma.\, M$ from the form of the function body M. Intuitively, in writing $\lambda x{:}\, \sigma.\, M$, we cannot apply any operations to x inside M that do not make sense for all values of type σ. For example, the expression $\lambda x{:}\, nat.\, x + true$ is not well formed, since it does not make sense to add $true$ to a natural number.

One way to understand lambda terms is by comparison with alternative notations. Another way to describe the function taking any $x{:}\, nat$ to itself is by writing $x{:}\, nat \mapsto x$. Perhaps a more familiar way of defining the identity function in programming languages is by writing something like

$$Id(x{:}\, nat) = x.$$

However, this notation forces us to make up a name for every function we write, while lambda notation gives us a succinct way of defining functions directly. Some other examples of lambda expressions are

$$\lambda x{:}\, nat.\, x + 1$$

which defines the successor function on natural numbers, and the constant function

$$\lambda x{:}\, nat.\, 5$$

which returns the natural number 5, regardless of the value of its argument.

An important aspect of lambda abstraction is that λ is a binding operator. This means that in a lambda term $\lambda x{:}\, \sigma.\, M$, the variable x serves only as a "placeholder," like the variable x in the integral

$$\int f(x)\, dx.$$

Just as $\int f(x)\, dx$ and $\int f(y)\, dy$ are different ways of writing the same integral, $\lambda x{:}\, \sigma.\, x$ and $\lambda y{:}\, \sigma.\, y$ are different ways of writing the same function. Therefore, we can rename λ-bound variables without changing the meaning of any expression, as long as any new name we choose does not conflict with other variables already in use (see Exercise 1.2.3). Terms that differ only in the names of bound variables are called α-equivalent; we sometimes write $M =_\alpha N$ if M and N are α-equivalent. When a variable x occurs in an expression M, we say an occurrence of x is *bound* if it is inside a subexpression of the form $\lambda x\colon \sigma.\, N$, and *free* otherwise. We write $FV(M)$ for the set of free variables of M. An expression is *closed* if it has no free variables.

In lambda notation, we write function application just by putting a function expression next to one or more arguments. This lets us use parentheses to specify association of operations. For example, we apply the natural-number identity function to the number 3 by writing

$$(\lambda x\colon nat.\, x)\, 3.$$

The value of this application is the identity function applied to the number 3, which is just 3, of course. This gives us the equation

$$(\lambda x\colon nat.\, x)\, 3 = 3.$$

There are several methods for calculating the values of lambda expressions, or proving equations between lambda expressions, as we shall see in the next section.

Some notational conventions make it easier to write and read lambda expressions. Unfortunately, the conventions of lambda calculus seem slightly awkward at first, and take a little practice to get used to. The first convention is that application associates to the left, so that MNP should be read as $(MN)P$. In words, we can read MNP as the expression, "apply the function M to argument N, and then apply the resulting function to argument P." For this to make sense, M must be a function which, given one argument, returns a function. The second notational convention is that the scope of each lambda is interpreted as being as large as possible. In other words, an expression $\lambda x\colon \sigma. \ldots$ may be parenthesized by inserting a left parenthesis after the type σ, and the matching right parenthesis as far to the right as will produce a syntactically well-formed expression. For example, we read $\lambda x\colon \sigma.\, MN$ as $\lambda x\colon \sigma.(MN)$, rather than $(\lambda x\colon \sigma.\, M)N$. Similarly, $\lambda x\colon \sigma.\, \lambda y\colon \tau.\, MN$ is short-hand for the parenthesized expression $\lambda x\colon \sigma.(\lambda y\colon \tau.(MN))$. The two conventions work well together. For example, a multi-argument function application may be written

$$(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, \lambda z\colon \rho.\, M)\, N\, P\, Q.$$

According to the two conventions, this expression is fully parenthesized as

$$(((\lambda x\colon \sigma.(\lambda y\colon \tau.(\lambda z\colon \rho.\, M)))\, N)\, P)\, Q$$

so that the i-th argument corresponds to the i-th formal parameter.

Exercise 1.2.1 Using the symbol $*$ for natural-number multiplication, write a lambda expression for the function mapping $x\colon nat \mapsto x^2$.

Exercise 1.2.2 Insert parentheses into the expression $(\lambda x\colon nat.\, \lambda y\colon nat.\, x + y)\, 3\, 2$ so that it is fully parenthesized. What would you expect the value of this expression to be? Given the type information about x and y, could you meaningfully insert parentheses in some other way?

Exercise 1.2.3 We may rename the bound variable x in $\lambda x\colon nat.\, \lambda y\colon nat.\, x + y$ to z without changing the function defined by this term. Explain why renaming x to y changes the meaning of the term and is therefore not a legal renaming of a bound variable. Can we rename x to y in $\lambda x\colon nat.\, x + y$?

1.3 Equations, Reduction, and Semantics

Historically, lambda notation developed as part of what is called *lambda calculus,* a system for reasoning about lambda expressions. In addition to syntax, there are three main parts of the formal system. In contemporary programming language terminology, these are called *axiomatic semantics, operational semantics,* and *denotational semantics.* A logician might call the first two "proof systems" and the third a "model theory." The axiomatic semantics is a formal system for deriving equations between expressions. The operational semantics is based on a directed form of equational reasoning called *reduction.* In computer science terminology, reduction may be regarded as a form of symbolic evaluation. The denotational semantics, or model theory, is similar in spirit to the model theories of other logical systems, such as equational logic or first-order logic: a model consists of a family of sets, one for each type, with the property that each well-typed expression may be interpreted as a specific element of the appropriate set.

1.3.1 Axiomatic semantics

In the equational axiom system, we have an axiom for renaming bound variables, and an axiom relating function application to *substitution.* To write these axioms, we will use the notation $[N/x]M$ for the result of substituting expression N for variable x in M. One subtle aspect of substitution is that free variables should not become bound. The simplest way to substitute N for x in M systematically is to first rename all bound variables in M so that they are different from all free variables in N. Then we can replace all occurrences of x with N. (A more detailed definition is given in Section 2.2.3.) Using substitution, the axiom for renaming bound variables is written

$$(\alpha) \qquad \lambda x{:}\sigma.\, M = \lambda y{:}\sigma.[y/x]M, \quad y \text{ not free in } M$$

For example, we have $\lambda x{:}\sigma.\, x = \lambda y{:}\sigma.\, y$.

Since the lambda term $\lambda x{:}\sigma.\, M$ defines the function with value M on argument x, we can compute the value at argument N by substituting N for x. For example, the result of applying function $\lambda x{:}\, nat.\, x + 5$ to argument 3 is

$$(\lambda x{:}\, nat.\, x + 5)\, 3 = [3/x](x + 5) = 3 + 5.$$

More generally, we have the equational axiom

$$(\beta)_{eq} \qquad (\lambda x{:}\sigma.\, M)\, N = [N/x]M,$$

called β-equivalence. Essentially, β-equivalence says that we can evaluate a function application by substituting the actual argument for the formal parameter within the function body. In addition to these axioms, and a few others, the equational proof system includes rules like symmetry, transitivity, and a congruence rule that says, "equal functions applied to equal arguments yield equal results." The latter rule may be written out formally as

$$\frac{M_1 = M_2,\ N_1 = N_2}{M_1 N_1 = M_2 N_2}\ ,$$

although to be completely precise, we must also specify types to make sure everything makes sense. As with other logical proof systems, the equational proof rules of typed lambda calculus allow us to derive logical consequences of any set of equational hypotheses.

1.3.2 Operational semantics

The reduction rules for lambda expressions give us a directed form of equational reasoning. Intuitively, the basic reduction rules describe single computation steps that can be repeated to evaluate an expression symbolically (or compute indefinitely if the expression does not have a simplest form). This symbolic "evaluation procedure," or "interpreter," is what gives lambda calculus its computational character. Although β-equivalence is an equation, we often tend to read it as a simplification rule, from left to right. For example, the equation $(\lambda x{:}\, nat.\, x + 5)\, 3 = 3 + 5$ suggests that we can simplify the function application to $3 + 5$. Using properties of addition, we may then simplify this to 8. Since reduction is asymmetric, an arrow $\rightarrow$ is commonly used for one-step reduction, and a double-headed arrow $\twoheadrightarrow$ for zero or more reduction steps.

The central reduction rule is a directed version of $(\beta)_{eq}$ called *β-reduction,* written

$$(\beta)_{red} \qquad (\lambda x{:}\sigma.\, M)\, N \xrightarrow{\beta} [N/x]M.$$

Since we may need to rename bound variables when performing a substitution, reduction is only defined up to α-equivalence. More precisely, the definition of substitution, $[N/x]M$, allows bound variables to be renamed, in order to avoid conflicts. As a result, the term obtained by β-reduction may depend on arbitrary choices of new bound variables; it is not uniquely determined. However, any two terms we might produce will differ only in the names of bound variables, and therefore be α-equivalent. An important point is that β-reduction often produces an expression that is much longer than the one we begin with, even though it might seem "simpler" in some intuitive sense. The reason is that when x occurs several times within M, the expression $[N/x]M$ may be much longer than $(\lambda x{:}\sigma.\, M)\, N$. Both α-conversion and renaming bound variables in substitution are part of providing *static scope,* as illustrated in Exercise 2.2.13.

The entire reduction system combines $(\beta)_{red}$ and other basic one-step reductions with rules that allow us to evaluate parts of a term, and repeat reduction steps. More precisely, we write $M \twoheadrightarrow N$ if we can produce N from M by repeatedly applying single-step reductions to subexpressions. For example,

$$(\lambda x{:}\sigma.\, M)((\lambda y{:}\tau.\, N)\, P) \twoheadrightarrow (\lambda x{:}\sigma.\, M)([P/y]N) \twoheadrightarrow [([P/y]N)/x]M.$$

Since there may be several ways to begin reducing an expression, $\twoheadrightarrow$ might be viewed as a "non-deterministic" model of execution. Another way to reduce the term above is to begin with the left-most lambda:

$$(\lambda x{:}\sigma.\, M)((\lambda y{:}\tau.\, N)\, P) \twoheadrightarrow [((\lambda y{:}\tau.\, N)\, P)/x]M \twoheadrightarrow [([P/y]N)/x]M.$$

However, choosing a different place to begin does not keep us from reaching the same final result. As we shall see, this is true in general about many forms of lambda calculus.

Two notable properties of reduction on pure typed lambda terms are the *Church-Rosser property,* and the *strong normalization property.* These are also called *confluence* and *termination,* respectively. The Church-Rosser property is that if $M \twoheadrightarrow N$ and $M \twoheadrightarrow N'$, then there exists an expression P such that $N \twoheadrightarrow P$ and $N' \twoheadrightarrow P$. It follows from confluence that an equation between lambda expressions is provable (using the equational proof system) iff both expressions may be reduced to a common form. Confluence may also be used to show that certain terms cannot be proved equal, establishing the consistency of the equational proof system. The strong normalization property states that no matter how we try to reduce a typed lambda expression, we cannot go on applying single-step reductions indefinitely. We eventually produce a *normal form,* an expression that cannot be simplified further. In PCF, we will add recursion to typed lambda calculus. This makes it possible to write nonterminating algorithms, and destroys strong normalization. However, PCF reduction remains confluent.

1.3.3 Denotational semantics

In the denotational semantics of typed lambda calculus, each type expression is associated with a set, called the set of values of this type. A term of type σ is interpreted as an element of the set of values of type σ, according to a definition by induction on the structure of terms. The set of values of type $\sigma \to \tau$ is a set of functions (or isomorphic to this), so that a typed lambda term $\lambda x\colon \sigma.\, M$ is interpreted as a mathematical function. While the semantics of pure typed lambda calculus is relatively simple, the semantics of its extensions may be considerably more complex. Some features that provide a challenge are recursive function definitions, which are computationally important but difficult to accommodate in classical set theory, recursive definition of types, and polymorphic functions, which are functions that may be applied to arguments of many types.

For the reader familiar with untyped lambda calculus, it is worth mentioning that untyped lambda calculus may be derived from typed lambda calculus in a meaningful way. In fact, one of the most natural ways to think about the semantics of untyped lambda calculus begins with the semantics of typed lambda calculus. For this reason, we consider typed lambda calculus the more basic system, and a more appropriate place to begin our study.

Exercise 1.3.1 Use $(\beta)_{red}$ twice to simplify the lambda expression $(\lambda f\colon nat \to nat.\, f5)(\lambda x\colon nat.\, x+x)$ to the sum of two natural numbers.

Exercise 1.3.2 For the purpose of this exercise and the next, the *length* of a typed lambda term is the number of symbols we use to write the term down, counting parentheses but not counting ":" or the type expressions inside the term. For example, the length of $\lambda x\colon a.\, \lambda y\colon b.\, y$ is 7. Find a pure typed lambda expression (without extra functions such as $+$) of the form

$$(\lambda f\colon (a \to a) \to (a \to a).\, \lambda x\colon a \to a.\, M)(\lambda f\colon a \to a.\, \lambda x\colon a.\, f(fx))$$

whose length at least doubles as the result of one or more β-reductions.

Exercise 1.3.3 Generalize the construction of Exercise 1.3.2 to show that as a result of performing one or more β-reductions, a pure typed lambda term of length $O(n)$ may produce a term whose length is greater than $f(n) = 2^{2^{\cdots 2^n}}$, where $f(n)$ is computed by exponentiating n times.

1.4 Types and Type Systems

In the general literature on computer science and mathematical logic, the word "type" is used in a variety of contexts and with a variety of meanings. In this book, *type* is a basic term whose meaning is determined by the precise definition of a *type system.* In any type system, types provide a division or classification of some universe of possible values: a type is a collection of values that share some property. Therefore, it always makes sense to ask what the elements of a type are. However, the kind of values that may have types, and the kinds of distinctions we make in typing, may vary from system to system.

Throughout this book, we will distinguish between *types,* which are collections, and *values,* which are members of types. In some systems, there may be types with types as members. Types with types as members are usually called something else, such as *universes, orders* or *kinds,* to avoid the impression of circularity. An exception is a language with a "type of all types," which is circular in the sense that the type of all types is a member of itself. However, since a type of all types introduces several anomalies (discussed in Chapter 9), none of the systems we consider in depth will have a type of all types.

In most programming languages, types are "checked" in some way, either during program compilation, or during execution. In compile-time type checking, we attempt to guarantee that each program phrase (expressions, declarations, etc.) defines an element of a type that is either specified explicitly as part of the program text, or determined implicitly by the way the program phrase is used. Programs that fail to type check are rejected by the compiler. Some advantages of compile-time type checking are

- *Early detection of errors.* An error such as adding an integer to a string that might occur only for certain input to the program can be detected by a compile-time type checker before the program is executed.

- *Documentation.* When programmers specify the types of identifiers and expressions, this tells something about the expected value at run-time. This can be valuable information when the reader is trying to figure out how the program works.

- *Guarantee validity of optimizations.* A simple example might be accessing a data structure such as an array whose layout (and therefore indexing function) depends on the type of data it contains. If the type and therefore size of the data is known at compile-time, a more efficient access function can be compiled. This arises in the compilation of Pascal records, C structs or C objects, where the offset of each field or member is determined statically using the type of the record or object.

Run-time type checking similarly tries to prevent execution of erroneous operations, but postpones the test until the program is executed. For example, although Lisp programs are not type-checked prior to compilation, run-time tests are used to make sure that list operations are only applied to lists; anyone who has tried to debug a nontrivial Lisp program has seen the error stating that you cannot apply *car* to an atom. Run-time type checking is more "accurate," in the sense that since the exact values of operands are known at run-time, only errors that actually occur in execution are detected. A simple example illustrating the difference is the expression

$$\texttt{if}\ B\ \texttt{then}\ 3\ \texttt{else}\ 4 + \text{``polyglot''}$$

which does not contain a run-time error if the expression B has value *true*. However, most compile-time type checkers would reject this expression. Since the value of an arbitrary expression cannot be determined at compile-time, compile-time type checking is "conservative." Most compile-time tests are based on simple algorithms that reject all programs with run-time type errors, but also reject some programs that would not contain run-time errors. This is an inevitable consequence of basic recursion theory: predicting run-time type errors is an undecidable property of programs.

We will focus on languages that may be called *statically-typed.* These are languages in which each program phrase must have a type and, except for extreme cases, the type can be determined efficiently from the syntactic form of the expression. Beginning with the precise presentation of syntax in Chapter 4, the languages we will discuss are defined by typing rules. Each well-formed phrase will have a type that may be determined by making a single pass over the phrase, reading the types of identifiers that are not declared locally from some kind of symbol table. One reason to focus on statically-typed languages is that we do not have to consider "run-time" error tests or the problems associated with certain kinds of programming errors. By only considering well-typed programs, we simplify our theory and focus on correct rather than incorrect programs. Another reason to consider typed languages is that types generally allow us to reason about the values of expressions in a way that would not be possible in an "untyped" (or run-time-typed) language. This is illustrated by the method of logical relations, presented in Chapter 8.

An important aspect of adopting a typed theory of programming languages is that untyped languages arise naturally as a special case. The main idea behind studying untyped languages in a typed framework is that we could use a type *untyped* for all the untyped expressions of a language. A specific example can be given using the syntax of the programming language ML [Mil85b, MTH90, MT91, Ull94]. If we have an untyped language with integers, booleans pairs and functions, we can represent the associated "untyped value space" using the following ML datatype:

```
datatype untyped = N of int | B of bool | P of untyped*untyped
                 | F of untyped → untyped
```

Intuitively, the type `untyped` is the union of the integers, booleans and all pairs and functions definable over these types. This ML `datatype` is an example of a recursively-defined type. Recursive types are considered in Sections 2.6.3 and 7.4. Example 4.4.6 discusses a related approach to untyped lambda calculus in a typed lambda calculus framework.

A final reason for including types in our theory of programming languages is the role of types in many approaches to modular software design. Since pragmatic issues in software design are not specifically considered in this book, the reader may wish to consult [Boo91, LG86, Mey88], for example, for a broader perspective. Since types appear in a variety of programming languages, for all of the reasons mentioned above, it is important to have a theoretical framework for typed languages.

1.5 Notation and Mathematical Conventions

The remainder of this chapter presents background information that may be useful for various parts of the book. While some may want to work through these sections before proceeding, many readers will prefer to skim the material to see what is here and use these sections later for reference, as needed.

In this brief section, we list some of the notational conventions associated with mathematical topics that are not covered explicitly in the book. Other symbols are defined as they are introduced. The point of definition of each symbol may be found in the index.

Equality relations

Several forms of equality and equality-like relations are used. We use the standard equality symbol $=$ to assert that two expressions have equal value. This is consistent with standard mathematical usage, such as $3+4 = 5+2$. An exception is that we also use $=$ in the declaration form $\texttt{let}\ x = M\ \texttt{in}\ N$, which is an expression of some of the languages studied in the book. However, it should be easy to distinguish these two uses by context. When symbols such as M and N stand for expressions of some language, we use $M \equiv N$ to mean that they are syntactically identical. In other words, the symbols "M" and "N" stand for the same expression. The notions of free and bound variable are discussed in Section 1.2. We consider two expressions that differ only in the names of bound variables syntactically equal, *e.g.*, $\lambda x\colon int.\, \lambda y\colon int.\, x \equiv \lambda y\colon int.\, \lambda x\colon int.\, y$. However, the names of free variables do matter, so $\lambda x\colon int.\, fx \not\equiv \lambda x\colon int.\, gx$.

The main equality symbols and their meaning are listed below.

$=$ Two expressions have the same value.

$\equiv$ Two expressions are syntactically equal, except possibly for the names of bound variables, as discussed above.

$\stackrel{def}{=}$ We write $M \stackrel{def}{=} N$ to indicate that a symbol or expression M is defined to be equal to N.

$::=$ Symbol used in grammar to indicate possible forms of expressions.

$\cong$ Isomorphism of structures (sets, algebras, etc.).

Logical symbols

$\forall$ Universal quantifier. The formula $\forall x.\, \phi$ may be read, " for all x, ϕ is true."

$\exists$ Existential quantifier. The formula $\exists x.\, \phi$ may be read, " there exists x such that ϕ is true."

$\wedge$ Conjunction. The formula $\phi \wedge \psi$ may be read, "ϕ and ψ."

$\wedge$ Disjunction. The formula $\phi \vee \psi$ may be read, "ϕ or ψ."

$\neg$ Negation. The formula $\neg\phi$ may be read, "not ϕ."

$\supset$ Implication. The formula $\phi \supset \psi$ may be read, "ϕ implies ψ." (We do not use $\rightarrow$ for implication since $\rightarrow$ is used in type expressions and for reduction (evaluation) of expressions.)

iff If and only if.

Like λ, discussed in Section 1.2, the quantifiers $\forall$ and $\exists$ are binding operators. Since a bound variable is just a place-holder, formulas $\forall x.\,\phi$ and $\forall y.\,[y/x]\phi$ are considered syntactically equal, provided y does not already occur free in ϕ and the substitution $[y/x]\phi$ is carried out with renaming of bound variables in ϕ to avoid capture, as summarized in Section 1.3 and explained in more detail in Section 2.2.3. Similarly, we have $\exists x.\,\phi \equiv \exists y.\,[y/x]\phi$.

Set operations

Although some aspects of set theory are summarized in Section 1.6, the primary symbols are listed here for reference.

$\in$ Element of.

$\cup$ Union.

$\cap$ Intersection.

$\subseteq$ Subset.

$\times$ Cartesian product.

1.6 Set-theoretic Background

1.6.1 Fundamentals

In some ways, set theory is the "machine language" of mathematics. Most of the time, we work with higher-level notions that are definable in set theory, without worrying about exactly how these concepts would be expressed in pure set theory. This subsection illustrates some of the main ideas of "elementary" set theory, without going into the foundational issues in depth. The main emphasis is on the spirit of the system and how it is used; this is not a comprehensive presentation of any particular axiomatic set theory. A good general reference on set theory, at an accessible level of detail but covering far more than is needed for this book, is [Hal60].

Intuitively, a *set* is a collection of elements, possibly empty. This is formalized in set theory by giving specific methods for defining and reasoning about sets. The main ideas are largely straightforward, except that we must be careful about the way we define sets. For example, we might think that for any property P, there is a set S with

$$x \in S \quad \text{iff} \quad P(x).$$

However, this cannot be true for "$x \notin S$," since this would give us a set S with

$$x \in S \quad \text{iff} \quad x \notin S.$$

This "definition" implies that any x is in S iff x is not in S, which cannot be satisfied by any reasonable collection. This example, which is called *Russell's paradox* after the philosopher and mathematician Bertrand Russell [Rus03], shows that we cannot make arbitrary set definitions and hope to have a reasonable mathematical theory. (There are other paradoxes, but Russell's paradox is the easiest to describe.) After the discovery of this paradox in the early 1900's, several satisfactory set theories were developed. The main idea in modern set theory is to begin with some simple, non-problematic sets and give operations for constructing additional ones.

The most basic principle about sets is that two sets are equal iff they have the same elements. This may be written more precisely as

$$A = B \quad \text{iff} \quad \forall x\ (x \in A \text{ iff } x \in B),$$

which may be read, "sets A and B are equal iff, for all x, x is in A iff x is in B." This statement is called the axiom of *extension.* Since this is an obvious property of collections, the point of this axiom about sets is just to make explicit that sets are collections, determined by their elements.

A useful operation on sets is to form their *cartesian product.* If A and B are sets, then $A \times B$ is the set of *ordered pairs*

$$\langle a, b\rangle \in A \times B \quad \text{iff} \quad a \in A \text{ and } b \in B.$$

More precisely, $x \in A \times B$ iff x has the form of an ordered pair, $x = \langle a, b\rangle$, with $a \in A$ and $b \in B$. In the remainder of this book, we will consider forming a pair $\langle a, b\rangle$ from elements a and b a basic operation. However, to give some feel for the way many operations are defined in set theory, we will show how cartesian products may be defined using other set constructions.

A basic axiom of set theory is that there exists an *empty set,* $\emptyset$, with no elements. This may be stated formally by writing

$$\forall x.\ x \notin \emptyset.$$

In words, "for every x, x is not an element of $\emptyset$." In most versions of set theory, we build all other sets out of the empty set. This results in a set-theoretic universe where everything is a set. However, it is also possible to develop set theory with what are traditionally called *urelements.* These are basic values that may be elements of sets, but are not sets themselves. The difference between set theory with urelements, and set theory without, will have no bearing on the topics covered in this book.

A simple operation is to form, for any mathematical object x, the *singleton set* $\{x\}$ with

$$y \in \{x\} \quad \text{iff} \quad y = x.$$

In other words, x is the only element of $\{x\}$. Another basic operation is *set union,* $A \cup B$, with

$$x \in A \cup B \quad \text{iff} \quad x \in A \text{ or } x \in B$$

for any sets A and B. Using singletons and union, we can form a set with any two elements by writing

$$\{a, b\} \quad \stackrel{\mathit{def}}{=} \quad \{a\} \cup \{b\}.$$

It is not hard to see, using properties of singleton and union given above, that this definition has the property

$$x \in \{a, b\} \quad \text{iff} \quad x = a \text{ or } x = b.$$

Although the general definition form $\{\, x \mid P(x)\,\}$ can be problematic, as we saw above with Russell's paradox, we can always write

$$\{\, x \in A \mid P(x)\,\}$$

for the set of all elements of A that satisfy the property P. A special case is the *intersection* of two sets,

$$A \cap B \quad \stackrel{\mathit{def}}{=} \quad \{\, x \in A \mid x \in B\,\}$$

Returning to ordered pairs, we can define the ordered pair $\langle a, b \rangle$ by

$$\langle a, b \rangle \stackrel{def}{=} \{\{a\}, \{a, b\}\}.$$

Before proceeding, it is worthwhile to pause and consider the sense of making this definition. If we were to define pairing as a basic notion, without reducing it to some construction on sets, two operations would be important. The first is that we need a way of forming the ordered pair $\langle a, b \rangle$ from a and b, and the other is that we need to be able to extract the components a and b from the pair $\langle a, b \rangle$. We show that our representation of ordered pairs is reasonable by showing that these essential operations are definable as set operations. If we define $\langle a, b \rangle$ as above, then we can form the ordered pair from a and b by taking the unions of singleton sets:

$$\{\{a\}, \{a, b\}\} = \{\{a\}\} \cup \{\{a\} \cup \{b\}\}.$$

Conversely, we can determine the first and second components of the pair uniquely. Specifically, if we write $\mathit{fst}\, p$ for the first component, we can characterize fst by

$$\mathit{fst}\, p = a \quad \text{iff} \quad \{a\} \in p.$$

Similarly, we can characterize the second component, $\mathit{snd}\, p$, by

$$\mathit{snd}\, p = b \quad \text{iff} \quad \{(\mathit{fst}\, p), b\} \in p.$$

Now that we have seen that our representation of ordered pairs by sets is reasonable, we may try to define cartesian product by

$$A \times B \stackrel{?}{=} \{\, \langle a, b \rangle \mid a \in A \text{ and } b \in B \,\}.$$

However, this uses the set definition form that led to Russell's paradox. For this reason, we still do not have a correct set-theoretic definition. To use the definition form $\{\, x \in C \mid P(x) \,\}$, we must find a set C that contains $A \times B$ as a subset. For this, we need the "powerset" constructor.

An important construction of set theory is called the *powerset.* The powerset, $\mathcal{P}(A)$, of a set A is the set of all sets drawn from the elements of A. Another way of saying this is to define the *subset* relation by

$$A \subseteq B \quad \text{iff} \quad \forall x.\, (x \in A \text{ implies } x \in B)$$

and write

$$A \in \mathcal{P}(B) \quad \text{iff} \quad A \subseteq B.$$

The powerset operation may be used to build bigger and bigger sets. For example, beginning with only the empty set, we may define the sequence of sets

$$\emptyset,\ \mathcal{P}(\emptyset),\ \mathcal{P}(\mathcal{P}(\emptyset)),\ \mathcal{P}(\mathcal{P}(\mathcal{P}(\emptyset))),\ \ldots$$

The powerset $\mathcal{P}(\emptyset)$ of the empty set is the singleton $\{\emptyset\}$ whose only element is the empty set. The kth element in this sequence may be written $\mathcal{P}^k(\emptyset)$, where the superscript k indicates that $\mathcal{P}$ is applied a total of k times. If $k > 0$, then $\mathcal{P}^k(\emptyset)$ has 2^{k-1} elements (see Exercise 1.6.1).

Returning to cartesian products, let A and B be sets with $a \in A$ and $b \in B$. The singleton $\{a\}$ and the set $\{a, b\}$ with two elements are both in the powerset $\mathcal{P}(A \cup B)$. Therefore, the ordered pair $\{\{a\},\{a,b\}\}$ is in $\mathcal{P}(\mathcal{P}(A \cup B))$, the powerset of the powerset. This finally gives us a way of defining the cartesian product using only set-theoretic operations that do not lead to paradoxes:

$$A \times B \stackrel{def}{=} \{\, \langle a,b \rangle \in \mathcal{P}(\mathcal{P}(A \cup B)) \mid a \in A \text{ and } b \in B \,\}.$$

In addition to ordered pairs, it is also possible to define ordered triples, or k-tuples for any positive integer k. If $A_1, \ldots, A_k$ are sets, then the cartesian product $A_1 \times \ldots \times A_k$ is the collection of all *k-tuples* $\langle a_1, \ldots, a_k \rangle$, with $a_i \in A_i$ for $1 \le i \le k$. The k-tuple $\langle a_1, \ldots, a_k \rangle$ may be defined as the set

$$\langle a_1, \ldots, a_k \rangle \stackrel{def}{=} \{\{a_1\}, \{a_1, a_2\} \ldots, \{a_1, \ldots, a_k\}\}$$

with basic operations on tuples as defined for ordered pairs. Exercise 1.6.2 shows that k-tuples can also be represented by nested pairs.

The final axiom of set theory that we will consider here is that there exists an infinite set. Without this assumption (*i.e.*, using only the empty set and the operations described so far), we would only be able to define and reason about finite sets. The "infinity" axiom allows us to define the set of natural numbers (non-negative integers), for example. In defining the natural numbers in pure set theory, we represent natural numbers by sets in much the same way as we represented ordered pairs by sets. Specifically the natural number 0 is represented by the empty set and the natural number $n+1$ by the set of all natural numbers $\le n$. However, it will not be necessary to go into this construction in any detail.

Exercise 1.6.1 This exercise asks about the size of the powerset of a finite set.

(a) Show that if a set A has n elements, we can associate a sequence of n bits with each subset of A. Use this to show that if A has n elements, then $\mathcal{P}(A)$ has 2^n elements.

(b) Consider the sequence of sets $\mathcal{P}^0(\emptyset)$, $\mathcal{P}^1(\emptyset)$, $\mathcal{P}^2(\emptyset)$, $\ldots$, where the superscript k in $\mathcal{P}^k(\emptyset)$ indicates that $\mathcal{P}$ is applied a total of k times. Show by induction on k that if $k > 0$, then $\mathcal{P}^k(\emptyset)$ has 2^{k-1} elements.

Exercise 1.6.2 In addition to defining k-ary cartesian products directly, as above, we can also represent k-ary cartesian products as repeated binary products. Specifically, if $A_1, \ldots, A_k$ are sets, then we can use $A_1 \times (A_2 \times \ldots \times (A_{k-1} \times A_k) \ldots)$ as the representation of the k-ary cartesian product. Show how this works by showing that $A_1 \times (A_2 \times \ldots \times (A_{k-1} \times A_k) \ldots)$ is *isomorphic* to $A_1 \times A_2 \times \ldots \times A_{k-1} \times A_k$. More specifically, show that there are functions

$$\begin{array}{lcl} f & : & A_1 \times (A_2 \times \ldots \times (A_{k-1} \times A_k) \ldots) \to A_1 \times A_2 \times \ldots \times A_{k-1} \times A_k \\ g & : & A_1 \times A_2 \times \ldots \times A_{k-1} \times A_k \to A_1 \times (A_2 \times \ldots \times (A_{k-1} \times A_k) \ldots) \end{array}$$

so that both function compositions $f \circ g$ and $g \circ f$ are the identity. (The *composition* $f \circ g$ is the function h with $h(x) = f(g(x))$. This is discussed more generally for relations on Section 1.6.2 and in Exercise 1.6.6.)

1.6.2 Relations and Functions

Intuitively, a relation between elements of some set A and elements of a set B is a "binary property" R such that $R(a,b)$ is either true or false for each $a \in A$ and $b \in B$. The common representation of relations as sets uses subsets of the cartesian product. Formally, a *relation* R between sets A and B is a subset $R \subseteq A \times B$ of their cartesian product. If an ordered pair $\langle a,b \rangle$ is in the subset

R, then we consider the relation true of a and b. If $\langle a, b\rangle$ is not in the subset R, then we consider the relation false of a and b. It is common to write $R(a, b)$ instead of $\langle a, b\rangle \in R$.

In addition to binary relations, it is possible to define k-ary relations, for any positive integer k. If $A_1, \ldots, A_k$ are sets, then a relation over $A_1, \ldots, A_k$ is a subset of the cartesian product $A_1 \times \ldots \times A_k$. In the special case $k = 1$, we have unary relations, or subsets, which are also called *predicates.*

Some important kinds of relations are equivalence relations and various kinds of orderings. A relation $R \subseteq A \times A$ is

Reflexive if $R(a, a)$, for all $a \in A$,

Symmetric if $R(a, b)$ implies $R(b, a)$, for all $a, b \in A$,

Transitive if $R(a, b)$ and $R(b, c)$ imply $R(a, c)$, for all $a, b, c \in A$.

An *equivalence relation* is a relation that is reflexive, symmetric and transitive. One example of an equivalence relation is equality, *i.e.*, the relation R with $R(a, b)$ iff $a = b$. Another example, on the natural numbers, is the relation $R \subseteq \mathcal{N} \times \mathcal{N}$ with $R(a, b)$ iff a and b are both odd or both even. A relation is

Antisymmetric if $R(a, b)$ and $R(b, a)$ then $a = b$, for all $a, b \in A$.

A *partial order* is a relation $R \subseteq A \times A$ that is reflexive, antisymmetric and transitive. An example on the natural numbers is the usual ordering relation $R \subseteq \mathcal{N} \times \mathcal{N}$ with $R(a, b)$ iff $a \leq b$. Some other partial orders are discussed in Section 5.2.2. The usual $\leq$ relation on numbers is called a *total order* since it also satisfies the property

Total order: For all $a, b \in A$, either $R(a, b)$ or $R(b, a)$.

This is not generally true for partial orders. For example, the prefix order on strings described in Exercise 5.2.4 is not a total order.

The set-theoretic representation of a function uses a special kind of relation. Before defining this, let us recall the intuitive notion of function. Informally, a function from set A to B is some way of associating an element of b with each element of a. This could take the form of an algorithm for computing b from a, or the function might be given by some condition that could not be carried out by a computer.

In set theory, a function is identified with its *graph,* the set of pairs $\langle a, b\rangle$ such that the value of the function on a is b. The graph of function is a relation that associates exactly one element of B with each element of A. More precisely, a *function* $f : A \to B$ from set A to B is a relation $f \subseteq A \times B$ satisfying the following properties:

$$\begin{array}{ll} (i) & \forall a \in A.\, \exists b \in B.\, \langle a, b\rangle \in f \\ (ii) & \forall a \in A.\, \forall b, b' \in B.\ \text{if } \langle a, b\rangle \in f \text{ and } \langle a, b'\rangle \in f \text{ then } b = b' \end{array}$$

Condition (i) says that the function is *defined* on every $a \in A$: for every a in A, there exists some b in B that is the value of f on a. Condition (ii) says that the function value is unique for each $a \in A$. The standard notation is to write $f(a)$ for the unique $b \in B$ with $\langle a, b\rangle \in f$. If $f : A \to B$, then we say A is the *domain* of f and B is the *range* or *codomain.*

Partial functions are important in computation since an algorithm for computing an element of set B, for each $a \in A$, might turn out not to halt for some $a \in A$. We think of such a rule as defining a *partial function,* which is like a function, but not necessarily defined on all $a \in A$. More precisely, a *partial function* $f : A \rightharpoonup B$ from set A to B is a relation $f \subseteq A \times B$ satisfying the property (ii) above, but not necessarily (i). Trivially, every total function from A to B is also a partial one. Exercises 1.6.4 and 1.6.5 show how to regard partial functions as total functions by changing either the domain or range. Note that we use an arrow $\rightharpoonup$ with a "partial" arrowhead to indicate that a function may be partial. If $f : A \rightharpoonup B$ is a partial function and $a \in A$, we write $f(a)$ for the unique element of b with $\langle a, b\rangle \in f$, if there is one. If not, then we consider $f(a)$ *undefined,* and say that the partial function f is undefined on a, or that a is outside the *domain of definition* of f.

Example 1.6.3 The recursive function expression

$$f(x\colon int) \;=\; \texttt{if } x = 0 \texttt{ then } 0 \texttt{ else } x + f(x-2)$$

defines a *partial* function on the integers. If we think of executing a function defined in this way on any number n, the result will be the sum of the even numbers up to n if n is even and not negative. Otherwise, the computation should continue indefinitely in principle, although most computers will terminate when the run-time stack is full or the smallest representable negative integer is reached. If we ignore this problem with "overflow," then the partial function defined by this expression is the following set of ordered pairs

$$f \;=\; \{\, \langle x, y\rangle \in int \times int \mid x = 2n \geq 0 \text{ and } y = \sum_{0 \leq i \leq n} 2i \,\}$$

It is easy to check that this relation satisfies condition (ii) above, but not condition (i) for odd numbers. ∎

Although we represent functions as sets of ordered pairs, we generally do not apply set-theoretic operations like union or intersection to functions. An exception is the use of the subset ordering on partial functions in Section 5.2.1.

A useful operation on relations and functions is *composition.* Since functions and partial functions are special cases of relations, it is simplest to define this operation on relations first, and then see that the composition of two functions is always a function, and similarly for partial functions. If we have two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, then we define their composition $S \circ R \subseteq A \times C$ by

$$S \circ R \;\stackrel{def}{=}\; \{\, \langle a, c\rangle \in A \times C \mid \exists b \in B.\, \langle a, b\rangle \in R \text{ and } \langle b, c\rangle \in S \,\}$$

In the case that these relations are functions $f : A \to B$ and $S : B \to C$, this definition gives us

$$g \circ f \;=\; \{\, \langle a, c\rangle \in A \times C \mid c = g(f(a)) \,\}$$

and similarly for partial functions as verified in Exercise 1.6.6. Some standard properties of functions are that a partial or total function $f : A \rightharpoonup B$ is *one-to-one,* sometimes written *1-1,* if

$$f(x) = f(y) \;\text{ implies }\; x = y$$

and *onto* if

$$\forall b \in B.\, \exists a \in A.\, f(a) = b$$

In words, a function is one-to-one if it maps distinct elements of its domain to distinct elements of its codomain and onto if every element in its codomain is the value of the function on some argument. These concepts generalize to relations in the obvious way, as noted in Exercise 1.6.6. Sometimes *injective* is used as a synonym for one-to-one and *surjective* as a synonym for onto. A function that is both injective and surjective is called *bijective.*

Exercise 1.6.4 Show that a relation $f \subseteq A \times B$ is a partial function $f : A \rightharpoonup B$ iff f is a total function $f : A' \to B$ on some subset of $A' \subseteq A$.

Exercise 1.6.5 Let A and B be sets and assume ∞ is not an element of B. Show that there is a one-to-one correspondence between partial functions $A \rightharpoonup B$ and total functions $A \to (B \cup \{\infty\})$.

Exercise 1.6.6 Prove the following facts about composition of relations $R \subseteq A \times B$ and $S \subseteq B \times C$:

(a) If R and S are partial functions, then $S \circ R$ is a partial function. Similarly for total functions.

(b) If R and S are one-to-one relations, then $S \circ R$ is one-to-one. Similarly for onto. (Begin by generalizing the definitions of one-to-one and onto from functions to relations. The idea remains that a relation is one-to-one if it relates distinct elements of its "domain" to distinct elements of its "codomain" and onto if every element in its "codomain" is related to some element of the "domain.")

(c) For any additional relation $T \subseteq C \times D$, we have $T \circ (S \circ R) = (T \circ S) \circ R$. This is called *associativity of composition* for relations (or functions).

1.7 Syntax and Semantics

1.7.1 Object language and meta-language

One fundamental idea in this book is the mathematical interpretation of syntactic expressions. This is actually a familiar concept, from all of our basic math courses. When we write an expression $3 + 5 - 7$, we use symbols 3,5,7,+ and $-$ to denote a mathematical entity, the number 1. What makes it confusing to talk (or write!) about the interpretation of syntax is that everything we write is actually syntactic. When we study a programming language, we need to distinguish the language we study from the language we use to describe this language and its meaning. The language we study is traditionally called the *object language,* since this is the object of our attention, while the second language is called the *meta-language,* because it transcends the object language in some way. With this distinction in mind, we can describe the relation between the expression $3 + 5 - 7$ and the number it identifies more precisely. Specifically, in our meta-language for discussing arithmetic expressions, let us use an underlined number, such as $\underline{1}$, to mean "the mathematical entity called the natural number 1". Then we can say that the meaning of the object language expression $3+5-7$ is the natural number $\underline{1}$. In this sentence, the symbol $\underline{1}$ is a symbol of the meta-language, while the expression $3 + 5 - 7$ is written using symbols of the object language.

1.7.2 Grammars

Grammars provide a convenient method for defining infinite sets of expressions. In addition, the structure imposed by a grammer gives us a a systematic way of defining properties of expressions, such as their semantic interpretation. This section summarizes basic properties of grammars, with parsing discussed briefly in the next section. In simple terms, the main point of these sections is to show how we can use ambiguous grammars, without getting bogged down in lots of details about parsing and ambiguity. The method we use, often referred to by the phrase *abstract syntax,* takes parse trees, rather than strings, as the true expressions of a language. We illustrate this use of grammars in Section 1.7.4 by giving the semantics of a simple expression language.

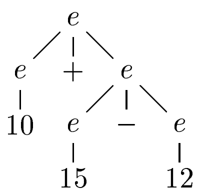
Consider the simple language of numeric expressions given by the grammar

$$\begin{array}{rcl} e & ::= & n \mid e+e \mid e-e \\ n & ::= & d \mid nd \\ d & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid 9 \end{array}$$

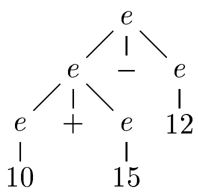
where e is what is called the *start symbol*, the symbol we begin with if we want to derive a well-formed expression from the rules of a grammar. The way a start symbol is used is that we begin with e, and continue to replace a symbol that occurs on the left of a ::= with one of the strings between vertical bars on the right until none of the *nonterminals* e, n or d are left. (The symbols that appear in expressions are called *terminals.*) For example, here are two derivations of well-formed expressions:

$$\begin{array}{lllllllll} e & \to n & \to nd & \to dd & \to 2d & \to 25 \\ e & \to e+e & \to e+e-e & \to \ldots & \to n+n-n & \to \ldots & \to 10+15-12 \end{array}$$

It is often convenient to represent a derivation by a tree. This tree, called the *parse tree* of a derivation, or *derivation tree,* is constructed using the start symbol as the root of the tree. If a step in the derivation is to replace s by $x_1 \ldots x_n$, then the children of s in the tree will be nodes labeled $x_1, \ldots, x_n$. The parse tree for $10 + 15 - 12$ has some useful structure. Specifically, since the first step yields $e + e$, the parse tree has the form



where we have contracted the subtrees for each two-digit number to a single node. What is useful to note about this tree is that it is different from



which is another parse tree for the same expression. An important fact about parse trees is that each corresponds to a unique parenthesization of the expression. Specifically, the first tree above corresponds to $10+(15-12)$ while the second corresponds to $(10+15)-12$. It is an accident that these expressions have the same numeric value. In general, the value of an expression may depend on how it is parsed or parenthesized, as illustrated by $1 - 1 - 1$.

A grammar is said to be *ambiguous* if there is some expression with more than one parse tree. If every expression has exactly one parse tree, the grammar is *unambiguous.*

Exercise 1.7.1 Draw the parse tree for the derivation of 25 given in this section. Is there another derivation for 25? Is there another parse tree?

1.7.3 Lexical analysis and parsing

Most compilers are structured into a series of distinct phases. In a standard compiler, the first two phases are lexical analysis and parsing:

$$\langle\text{program text}\rangle \longrightarrow \boxed{\text{lexical analyzer}} \longrightarrow \boxed{\text{parser}} \longrightarrow \quad \ldots\text{rest of compiler}\ldots$$

Lexical analysis typically separates input characters into tokens and identifies the keywords of the language (also called "reserved words"). Parsing determines whether the program satisfies the conditions imposed by the context-free grammar of the language. In the process, a syntactically correct program is converted from a linear sequence of symbols to a tree representation. Although it is possible for the resulting tree to be the actual parse tree of the input program, this is not necessary. What we generally want is a parse tree for some related program, in a grammar that may be simpler. For example, if the input program contains parentheses, these need not occur in the parser output, since parenthesization is determined by the tree structure. Therefore, for the purposes of this book, we will consider a parser to be an algorithm that takes a string generated by one grammar as input and produces, as output, a parse tree for a possibly different grammar.

In programming language theory, we prefer to work with parsed expressions, instead of text strings. The reason is that parsing resolves ambiguities that are routine and syntactic and have nothing to do with more fundamental properties of programs. Just as compilers are structured so that program analysis and code generation phases take parsed programs as input, our mathematical treatment of programs works best if we assume programs are already parsed.

The traditional terminology of the field is that programs are written according to a *concrete syntax,* which specifies how expressions may (or must) be parenthesized, the spelling of keywords, and so on. The output of parsing may be a parse tree for an *abstract syntax,* which may be a grammar that does not include such things as parenthesization, since this has been resolved by the parser. The parse tree for an expression, written in the abstract syntax, is called an *abstract syntax tree.* In programming language theory, we write expressions in the usual way, but read these strings of symbols as if they are shorthand for some abstract syntax tree.

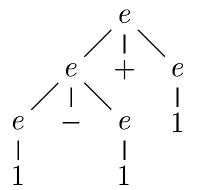
Example 1.7.2 There are several reasonable ways to write arithmetic expressions. One common variation is the position of the operand in an expression. In prefix expressions, the operand comes first, while postfix expressions have the operand last, and infix in the middle. These three choices are illustrated in the following three grammars.

$$\begin{array}{lcll}
p & ::= & 0 \mid 1 \mid +\,p\,p \mid -\,p\,p & \qquad\qquad \text{(prefix expressions)} \\
s & ::= & 0 \mid 1 \mid s\,s\,+ \mid s\,s\,- & \qquad\qquad \text{(postfix expressions)} \\
i & ::= & 0 \mid 1 \mid (i+i) \mid (i-i) & \qquad\qquad \text{(infix expressions)}
\end{array}$$

Regardless of which we choose as concrete syntax, we can parse expressions to the abstract syntax

$$e \quad ::= \quad 0 \mid 1 \mid e+e \mid e-e$$

More precisely, given any prefix, postfix or infix expression, we can write a unique parse tree in the fourth grammar that faithfully captures the syntactic structure and meaning of the expression. For example, the expressions $+{-}111$, $111{-}{+}$ and $(1-1)+1$ are all faithfully represented by the parse tree



This would be the abstract syntax tree for each of these expressions.

Note that in prefix and postfix notation, parentheses are unnecessary. For example, the prefix expression $+-+0111$ can only be parenthesized as $+(-(+01)1)1$ and the postfix expression $010++$ can only be parenthesized as $0(10+)+$. However, parentheses are needed in the concrete syntax of infix expressions, since $1-1+1$ can be parenthesized as $(1-1)+1$ or $1-(1+1)$. ∎

The useful syntactic conventions of *precedence* and *right-* or *left-associativity* are illustrated briefly by example in the following exercise. For more information, the reader may consult a compiler text such as [ASU86].

Exercise 1.7.3 A programming language designer might decide that expressions should include addition, subtraction, and multiplication and write the following *abstract syntax:*

$$e \ ::= \ 0 \mid 1 \mid e+e \mid e-e \mid e*e$$

(a) Explain why this is a perfectly reasonable abstract syntax for an expression language but not (by itself) a good concrete syntax.

We can make parsing "unambiguous" by adopting parsing conventions. Specifically, a plausible concrete syntax might be

$$e \ ::= \ 0 \mid 1 \mid (e+e) \mid (e-e) \mid (e*e) \mid e+e \mid e-e \mid e*e$$

combined with two parsing conventions that take effect when an expression is not fully parenthesized. The first convention is that $*$ has a higher *precedence* than $+$ and $-$, which have equal precedence. An unparenthesized expression $e \ op_1 \ e \ op_2 \ e$ is parsed as if parentheses are inserted around the operator of higher precedence. The second convention is that when identical operators, or operators of equal precedence, appear contiguously, the operations are associated to the left. This parses $e \ op \ e \ op \ e$ as if it were $(e \ op \ e) \ op \ e$. Write abstract syntax trees for the following expressions.

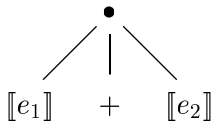
(b) $1-1*1$.

(c) $1-1+1$.

(d) $1-1+1-1+1$ if we give $+$ higher precedence than $-$.

1.7.4 Example mathematical interpretation

We may interpret the expressions given in Section 1.7.2 as natural numbers using induction on the structure of parse trees. More specifically, we define a function $\mathcal{E}$ from parse trees to natural numbers, defining the function on a compound expression by referring to its value on simpler expressions. An historical convention is to write $[\![e]\!]$ for any parse tree of the expression e. When we write $[\![e_1+e_2]\!]$, for example, we mean a parse tree of the form



with $\llbracket e_1 \rrbracket$ and $\llbracket e_2 \rrbracket$ as immediate subtrees.

Using this notation, we may define the meaning $\mathcal{E}\llbracket e \rrbracket$ of an expression e, according to its parse tree $\llbracket e \rrbracket$, as follows:

$$
\begin{array}{lcl}
\mathcal{E}\llbracket 0 \rrbracket & = & \underline{0} \\
\mathcal{E}\llbracket 1 \rrbracket & = & \underline{1} \\
\ldots & = & \ldots \\
\mathcal{E}\llbracket 9 \rrbracket & = & \underline{9} \\
\mathcal{E}\llbracket nd \rrbracket & = & \mathcal{E}\llbracket n \rrbracket * \underline{10} + \mathcal{E}\llbracket d \rrbracket \\
\mathcal{E}\llbracket e_1 + e_2 \rrbracket & = & \mathcal{E}\llbracket e_1 \rrbracket + \mathcal{E}\llbracket e_2 \rrbracket \\
\mathcal{E}\llbracket e_1 - e_2 \rrbracket & = & \mathcal{E}\llbracket e_1 \rrbracket - \mathcal{E}\llbracket e_2 \rrbracket
\end{array}
$$

In words, the value associated with a parse tree of the form $\llbracket e_1 + e_2 \rrbracket$, for example, is the sum of the values given to the subtrees $\llbracket e_1 \rrbracket$ and $\llbracket e_2 \rrbracket$. Since these parse trees are shorter than the parse tree $\llbracket e_1 + e_2 \rrbracket$ we may assume inductively that the function $\mathcal{E}$ is already defined on $\llbracket e_1 \rrbracket$ and $\llbracket e_2 \rrbracket$. This is the sense in which the definition of $\mathcal{E}$ is "by induction on the structure of (parse trees of) expressions." For those uncomfortable or unfamiliar with this form of induction, a summary of various forms of induction is given in Section 1.8. On the right of the equal signs, numbers and arithmetic operations $*$, $+$ and $-$ are meant to mean the actual natural numbers $\underline{0}, \ldots, \underline{10}$ and the standard integer operations of multiplication, addition and subtraction. In contrast, the symbols $+$ and $-$ in double square brackets on the left of the equal signs are symbols of the object language, the language of arithmetic expressions.

The main property to observe about this definition is that the meaning of a term depends on how it is parsed. If we take the example $6 - 3 - 2$, then we have two possibilities for $\llbracket 6 - 3 - 2 \rrbracket$. Both have the form $\llbracket e_1 - e_2 \rrbracket$, but in one we have $e_1 \equiv 6$ and in the other $e_1 \equiv 6 - 3$. Using the first tree, we may work out the meaning of the term as

$$
\begin{array}{lcll}
\mathcal{E}\llbracket 6 - 3 - 2 \rrbracket & = & \mathcal{E}\llbracket 6 \rrbracket - \mathcal{E}\llbracket 3 - 2 \rrbracket & \\
& = & \underline{6} - (\mathcal{E}\llbracket 3 \rrbracket - \mathcal{E}\llbracket 2 \rrbracket) & \\
& = & \underline{6} - \underline{1} & = \quad \underline{5}
\end{array}
$$

In general, we will define the syntax of terms using ambiguous grammars, but use parentheses or syntactic conventions such as precedence or association to the left to identify the intended parsing of each expression we write. In this view, using our example grammar without parentheses, the parentheses in $(1 - 1) + 1$ are not symbols of the object language, but additional symbols that indicate a specific abstract syntax tree. This abstract syntax tree represents the parenthesization of this expression by its tree structure.

1.8 Induction

This book contains many proofs by induction. The most common forms are induction on the structure of expressions and induction on the length or structure of proofs. This section explains these forms of induction and puts them in perspective by presenting some other forms of induction, beginning with induction on the natural numbers. In the process of covering induction on proofs, we also review some basic terminology and properties of formal proof systems. While some readers may choose to review induction on natural numbers and expressions immediately, the later topics in this section are more likely to be useful as reference for later parts of the book.

There are many books with additional discussion of induction at approximately this level, such as [AU92, Win93, MW90]. More thorough mathematical treatments of inductive proofs and inductive definitions may be found in [Acz77, Mos74].

1.8.1 Induction on the natural numbers

A simple and intuitive way to think of induction on the natural numbers is that it is a method for writing down an infinite proof in a finite way. Suppose we have a property P that we would like to prove for every natural number. For example, $P(n)$ might be the simple property "n is either odd or even." One way to prove $P(n)$ for every n, if we had an infinite amount of time and an infinite sheet of paper to write on, would be to write out the proof of $P(0)$, then write out the proof of $P(1)$, then write out the proof of $P(2)$, and so on. This is not really feasible, but if it were, the result should certainly be considered a proof. The value of induction is that it provides a simple way of demonstrating that if we had infinite time and space, we could write down a proof of $P(n)$ for every natural number n.

The most common form of induction on the natural numbers is this:

> **Natural Number Induction, Form 1:** To prove that $P(n)$ is true for every natural number n, it is sufficient to prove $P(0)$ and to prove that for any natural number m, if $P(m)$ is true then $P(m+1)$ must also be true.

The proof of $P(0)$ is called the *base case* and the proof that $P(m)$ implies $P(m+1)$ is called the *induction step.* When we assume $P(m)$ in order to prove $P(m+1)$ in the induction step, this assumption is called the *induction hypothesis.* Sometimes natural-number induction is presented using a template like this:

> *Goal:* Prove $P(n)$ for every natural number n.
> *Base case:* Prove $P(0)$.
> *Induction step:* Prove that for any natural number m, if $P(m)$ then $P(m+1)$.

It is easy to argue that if we can prove the base case and the induction step, then we could in principle write out the proof of $P(n)$ for every natural number n. We would begin by writing the base case at the top of our infinite sheet of paper. Then, since $P(0)$ implies $P(1)$, we can use the fact that we have already proved $P(0)$ to prove $P(1)$. Now we have a proof of $P(0)$ and a proof of $P(1)$. Repeating this idea, we can use the proof of $P(1)$ to write out a proof of $P(2)$, then the proof of $P(2)$ to write out a proof of $P(3)$, and so on indefinitely, eventually proving $P(n)$ for each n. Since the base case and the induction step are all that we needed to construct this "infinite proof," the base case and induction step for P certainly must imply $P(n)$ for every natural number n. In other words, induction is an intuitively sound (correct) proof method for establishing facts about the natural numbers. It is not clear that every "infinite proof" can be captured by an inductive argument, but the reader should be ready to believe that when we have a proof by induction, the conclusion must be true. We give a very simple example to illustrate the method.

Example 1.8.1 There is a story claiming that as a young child, the mathematician Gauss derived the formula

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

for the sum of the first n nonnegative integers. Gauss apparently figured this out when his teacher gave him the tedious chore of adding up the first hundred natural numbers, presumably in an attempt to keep him quiet. If we let $sum(n) = \sum_{i=1}^{n} i$ be the sum of the first n integers, then we can easily prove by induction that for every natural number n, $sum(n) = n(n+1)/2$. For clarity, we use the template above.

Goal: Prove $sum(n) = n(n+1)/2$ for every natural number n.

Base case: We must prove $sum(0) = 0(0+1)/2$. This is an easy calculation.

Induction step: We must show that for any natural number n, if $sum(n) = n(n+1)/2$ then $sum(n+1) = (n+1)(n+2)/2$. We therefore assume that $sum(n) = n(n+1)/2$ for some (arbitrary) natural number, n, and show that this holds for the next natural number, $n+1$. Since the sum of the first $n+1$ numbers is just the sum of the first n, plus $n+1$, we have $sum(n+1) = sum(n) + (n+1)$. We may now apply the induction hypothesis, namely, $sum(n) = n(n+1)/2$. This gives us

$$sum(n+1) = sum(n) + (n+1) = n(n+1)/2 + (n+1),$$

which completes the proof since it is an easy calculation to show that $n(n+1)/2 + (n+1) = (n+1)(n+2)/2$. ■

There is an equivalent form of natural number induction that has a stronger-looking induction hypothesis. This may also be understood by thinking about infinite proofs. If we have written proofs of $P(0), P(1), \ldots, P(n)$ and our next task is to write out the proof for $n+1$, then we should be able to use all of the facts $P(0), P(1), \ldots, P(n)$, not just $P(n)$. This leads to a second form that may be easier to use in some cases.

> **Natural Number Induction, Form 2:** To prove that $P(n)$ is true for every natural number n, it is sufficient to prove that for any natural number m, if $P(i)$ is true for all $i < m$, then $P(m)$ must also be true.

In this form of induction, there is no base case, only an induction step. In the induction step, we assume that $P(i)$ is true for all $i < m$, which is again called the *induction hypothesis,* and show that $P(m)$ is true. In practice, we often treat the special case $m = 0$ separately since there are no natural numbers less than 0. The second form of natural-number induction is sometimes called *strong induction* or *complete induction,* but from a logical point of view there is nothing stronger or more complete about it than the first form.

Example 1.8.2 The second form of natural-number induction is more convenient for proving that every natural number greater than 1 is the product of primes. The reason is that when we factor a number $n > 1$ that is not prime, we generally get numbers that are less than $n - 1$. Therefore, it is useful to have an induction hypothesis that covers all numbers less than n. (We say a natural number is *composite* if it is the product of two natural numbers greater than 1, and *prime* otherwise.)

Let P be the property

$$P(n) \stackrel{def}{=} \text{ if } n > 1 \text{ then there exist prime numbers } p_1, \ldots, p_k \text{ with } n = p_1 \ldots p_k.$$

Using the second form of induction, it suffices to show, for arbitrary m, if $P(i)$ for all $i < m$ then $P(m)$. Let m be any natural natural number. If $m \leq 1$, or m is prime, then it is easy to conclude $P(m)$. In the remaining case, m must be the product of two numbers, $m = m_1 m_2$, with both m_1 and m_2 greater than 1. The induction hypothesis is that $P(i)$ is true for all $i < m$. Since $m_1, m_2 < m$, it follows immediately from the induction hypothesis that m_1 and m_2 are both products of primes. Therefore m must also be a product of primes. Thus, by the second form of induction, we may conclude that every number greater than 1 is the product of primes. The reader may enjoy trying to prove this using the first form of induction. ■

It is worth taking the time to show that even though the second form of induction may look more powerful, the first form of natural number induction implies the second. Let us assume that the first form of induction holds for every property of the natural numbers and that, for some property P, we can prove that if $P(i)$ is true for all $i < m$ then $P(m)$. We will show $\forall n. P(n)$ using only the first form of induction. The trick is to let Q be the property

$$Q(m) \stackrel{def}{=} \text{ for all } i < m,\ P(i).$$

By the first form of induction, we can show $\forall n. Q(n)$ by showing the base case, $Q(0)$, and the induction step, $\forall m. (Q(m) \supset Q(m+1))$. Since there are no natural numbers less than 0, the base case is true, regardless of the property P. Therefore, we need only show the induction step, $Q(m) \supset Q(m+1)$ for every m. By definition of Q, $Q(m)$ means for all $i < m$, $P(i)$ and $Q(m+1)$ means for all $i < m+1$, $P(i)$. However, the only number covered by $Q(m+1)$ that is not already covered by $Q(m)$ is $P(m)$. Consequently, all we need to show is that if $P(i)$ for all $i < m$, then $P(m)$. But this was our original assumption about the property P. Therefore, the first form of induction implies the second. It is left to the reader, as Exercise 1.8.4, to show the converse.

Natural-number induction can also be used to prove properties of elements of other sets, using functions into the natural numbers. For example, we may establish some property of trees using natural-number induction on the size or height of trees. This is illustrated in Example 1.8.3 below. In more general terms, we can apply natural-number induction to a set A using any function $f : A \to \mathcal{N}$. (Of course, one function from A to $\mathcal{N}$ might make it possible to prove the property we are interested in while another function might not.) Using a function $f : A \to \mathcal{N}$, we can convert a property P of elements of A into a property on the natural numbers by defining

$$Q(n) \stackrel{def}{=} \forall a \in A. \text{if } f(a) = n \text{ then } P(a)$$

If we want to prove $P(a)$ for every $a \in A$, then since f maps every element of A to some natural number, it suffices to prove $Q(n)$ for every natural number n. If we use the first form of natural-number induction, then in the base case we must prove $P(a)$ for all a with $f(a) = 0$. In the induction step, we assume $P(a)$ for all a with $f(a) = n$ and prove $P(a)$ for all a with $f(a) = n + 1$. This general idea is used in Exercise 1.8.5.

Example 1.8.3 We illustrate natural-number induction on the height of trees. For the purposes of this example, a *binary tree* is either empty, a leaf or an "internal node" with two subtrees. Some examples are shown in Figure 1.1. The first, (a), is a single leaf, the second, (b), an internal node

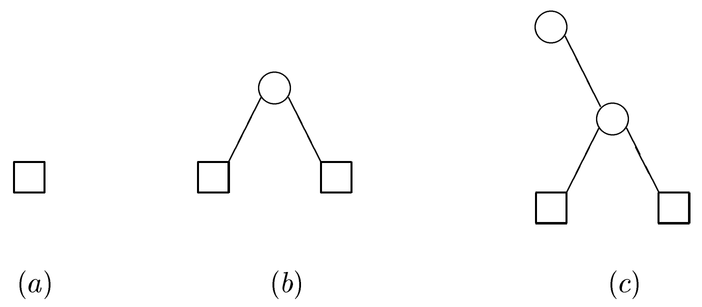


Figure 1.1: Binary trees.

with two subtrees and the third, (c), consists of an internal node with two subtrees, one of them empty. The *height of a tree* is the length of the longest path from an internal node downward to a leaf. The tree (a) in Figure 1.1 has height 0, tree (b) height 1, and tree (c) height 2. Note that this definition gives both an empty tree and a single leaf height 0.

We will prove that the number of leaves of any binary tree is at most one plus the number of internal nodes, using natural-number induction on the height of trees. More specifically, we define the property P of binary trees by

$$P(t) \quad \stackrel{def}{=} \quad \text{tree } t \text{ has at most one more leaf than internal nodes.}$$

Using the function $\mathit{height} : \mathit{Trees} \to \mathcal{N}$, we formulate the following property Q of natural numbers:

$$Q(n) \quad \stackrel{def}{=} \quad \forall \text{ trees } t, \text{ if } \mathit{height}(t) = n \text{ then } P(t).$$

We will prove $\forall n.\, Q(n)$ using the second form of natural-number induction since the two subtrees of an internal node may have different heights.

Let n be any natural number. We assume that for any $i < n$ and any tree t, if $\mathit{height}(t) = i$ then $P(t)$. We must show that for all trees t, if $\mathit{height}(t) = n$ then $P(t)$.

We first consider the special case $n = 0$. Since there is no natural number $i < 0$, we must prove directly that if $\mathit{height}(t) = 0$ then $P(t)$. However, this is easy to do. There are two trees with height 0, the empty tree and a single leaf. In each case, the number of leaves is clearly no greater than 1. This completes the proof for the special case $n = 0$.

If $n > 0$, then any tree t of height n must consist of an internal node with two subtrees, t_1 and t_2. Since the heights of t_1 and t_2 are both less than n, we may assume $P(t_1)$ and $P(t_2)$. The number of leaves of t is the sum of the leaves of the two subtrees, while the number of internal nodes is one greater than the sum of the internal nodes. Writing $\mathit{leaves}(s)$ for the number of leaves of tree s and $\mathit{nodes}(s)$ for the number of internal nodes, we therefore have

$$\mathit{leaves}(t) = \mathit{leaves}(t_1) + \mathit{leaves}(t_2) \leq \mathit{nodes}(t_1) + 1 + \mathit{nodes}(t_2) + 1 = \mathit{nodes}(t) + 1.$$

This completes the proof that the number of leaves of any tree is at most one plus the number of internal nodes. ∎

Exercise 1.8.4 Prove that the second form of natural-number induction implies the first. More specifically, assume that for some property P, we know that for all m, if $P(m)$ then $P(m+1)$. Use the second form of natural-number induction to prove that $P(n)$ is true for every natural number n.

Exercise 1.8.5 We can prove $P(n)$ for all of the positive and negative integers (including 0) using a form of "integer" induction. Specifically, we show $P(0)$ as the base case and, as the induction step, show that $P(n)$ implies both $P(n-1)$ and $P(n+1)$. Explain intuitively why this makes sense and prove that this principle follows from either the first or the second form of natural-number induction given in this section.

1.8.2 Induction on expressions and proofs

As illustrated in Example 1.8.3, we can use natural-number induction to prove properties of trees (or other objects), as long as we have a way of associating a natural number with each tree. We can also formulate independent induction principles for trees and certain other mathematical objects. The important property, as characterized mathematically in Section 1.8.3, is that we must be able to arrange the objects in some order such that each object occurs at most a finite number of steps above some minimal object. Intuitively, this kind of ordering would allow us to write out a form of infinite proof, eventually covering each object we are interested in. For the examples we consider, our direct induction principles can be derived from natural-number induction. Or, conversely, we can consider natural-number induction a special case of the more general principle.

Induction on expressions

As illustrated in Section 1.7, we often define sets of expressions using grammars. Let us use the grammar

$$e \quad ::= \quad 0 \mid 1 \mid v \mid e+e \mid e*e$$

as an example, where we assume that there is an infinite set $\mathcal{V}$ of variable symbols, and v in this grammar means that any element of $\mathcal{V}$ is an expression. The expressions generated by this grammar are precisely the strings that have a derivation or, equivalently, a parse tree, as explained in Section 1.7.2. Some examples are $0+1*x+y$ and $x+x+x+y$, assuming $x, y \in \mathcal{V}$. Since every expression has a parse tree, we can prove facts about expressions using induction on the height of parse trees, following the pattern illustrated in Example 1.8.3. More specifically, if P is a property of expressions, we can define a property Q of natural numbers by

$$Q(n) \quad \stackrel{def}{=} \quad \forall \text{ trees } t.\ \text{if } \mathit{height}(t) = n \text{ and } t \text{ is a parse tree of } e \text{ then } P(e)$$

Notice that this is a sensible property of natural numbers even when some expressions may have more than one parse tree. An alternative that often leads to cleaner proofs is to use a separate form of induction for expressions. We will explain this form of induction by considering the essential steps of an inductive proof using the natural numbers.

Suppose we begin with a property P on trees and define an associated property Q on the natural numbers as above. If we use natural-number induction to prove $\forall n.\ Q(n)$, then we will have to prove P directly for parse trees of height zero. For parse trees of height at least one, we can assume P for any expression with a shorter parse tree. For our example grammar, this means that for 0, 1 or a variable v, we must prove P directly. For a compound expression of the form $(e_1 + e_2)$ or $(e_1 * e_2)$, we could assume that P holds for subexpressions e_1 and e_2. Stating this generally for any grammar, we have the following form of induction on the structure of expressions:

> **Structural Induction, Form 1:** To prove that $P(n)$ is true for every expression generated by some grammar, it is sufficient to prove $P(e)$ for every atomic expression and, for any compound expression e with immediate subexpressions $e_1, \ldots, e_k$, prove that if $P(e_i)$ for $i = 1, \ldots, n$, then $P(e)$.

For the example grammar above, this gives us the following template:

> *Goal:* Prove $P(e)$ for every expression e.
> *Base cases:* Prove $P(0)$, $P(1)$ and $P(v)$ for any variable v.
> *Induction steps:* Prove that for any expressions e_1 and e_2, if $P(e_1)$ and $P(e_2)$ then $P(e_1 + e_2)$ and $P(e_1 * e_2)$.

We have three base cases since there are three forms of atomic expressions without subexpressions, and two induction steps since there are two compound expression forms.

Example 1.8.6 We illustrate structural induction on expressions by proving that every expression given by the grammar above defines a multi-variate function bounded by a certain form of polynomial. More specifically, let P be the property of expressions

> $P(e) \stackrel{def}{=}$ "for any list $v_0, \ldots, v_n$ of variables containing all the variables in e, there is a polynomial $cv_0^k v_1^k \ldots v_n^k$, such that for all natural number values of $v_0, \ldots, v_n$ greater than 0, the value of e is less than the value of the polynomial."

We show that for every expression e, we have $P(e)$, by induction on the structure of expressions. For clarity, we present this using the template above.

Goal: Prove $P(e)$ for every expression e.

Base cases: Prove $P(0)$, $P(1)$ and $P(v_i)$ for any variable v_i. These are immediate since $0 < v_0 v_1 \ldots v_n$, $1 < 2v_0 v_1 \ldots v_n$ and $v_i < 2v_0 v_1 \ldots v_n$ for any values of $v_0, \ldots, v_n$ greater than 0.

Induction steps: Prove that for any expressions e and e', if $P(e)$ and $P(e')$ then $P(e + e')$ and $P(e * e')$. Let $v_0, \ldots, v_n$ be a list containing all the variables in e and e'. Our induction hypothesis implies that there exist polynomials $cv_0^k \ldots v_n^k$ and $c' v_0^{k'} \ldots v_n^{k'}$ that are greater than or equal to the values of e and e', respectively, for all natural number values of $v_0, \ldots, v_n$ greater than 0. For the compound expression $e + e'$, we can establish $P(e + e')$ using the polynomial

$$(c + c') v_0^{\max(k,k')} \ldots v_n^{\max(k,k')}$$

and for $e * e'$ the polynomial

$$cc' v_0^{(k+k')} \ldots v_n^{(k+k')}$$

This completes the inductive proof. ∎

Although the difference is not usually emphasized as much as for natural-number induction, there is a second form of structural induction that includes all subexpressions in the induction hypothesis.

> **Structural Induction, Form 2:** To prove that $P(n)$ is true for every expression generated by some grammar, it is sufficient to prove that for any expression e, if $P(e')$ for every subexpression e' of e, then $P(e)$.

The difference between Form 1 and Form 2 is that in the second form, the induction hypothesis includes all subexpressions, not just the immediate subexpressions. (For the expression $x+(y+z)$, the immediate subexpressions are x and $y+z$; the variable y is a subexpression but not an immediate subexpression.)

We can regard each form of natural-number induction as a special case of the corresponding form of structural induction. Specifically, consider the grammar

$$n \quad ::= \quad 0 \mid succ\, n$$

where intuitively the successor $succ\, n$ of n is $n+1$. Every natural number can be written down using this grammar, and the two forms of induction on expressions give us exactly to the two forms of natural-number induction.

Induction on proofs

The main ideas behind induction on the structure of proofs are essentially the same as for induction on the structure of expressions. In many respects, both are really forms of induction on trees. Before stating induction on the structure of proofs, we review some basic concepts common to the most common form of proof systems.

A Hilbert-style *proof system* consists of axioms and proof rules. An *axiom* of a proof system is a formula that is provable by definition. An *inference rule* asserts that if some list of formulas are provable, then so is another formula. A *proof,* therefore, is a structured object built from formulas according to constraints established by a set of axioms and inference rules. Proofs are described more fully below.

Axioms and inference rules are generally written as schemes, representing all formulas or proof steps of a given form. For example, the reflexivity axiom for equality

$$(ref) \qquad e = e$$

is called a "scheme with metavariable e". This axiom scheme asserts that every equation of the form $e=e$ is an axiom. In particular, $x=x$, $y=y$ and $3=3$ are axioms, provided that x, y and 3 are all well-formed expressions of the language we have in mind. An inference rule scheme generally has the form

$$\frac{A_1 \quad \ldots \quad A_n}{B}$$

meaning that if we have proofs of formulas of the form $A_1, \ldots, A_n$, then we can combine these to obtain a proof of the corresponding formula B. For example, the inference rule for transitivity of equality is written

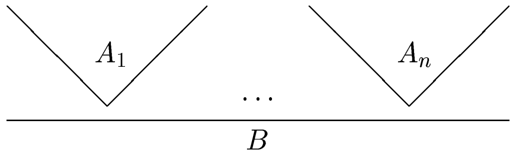
$$(trans) \qquad \frac{e_1 = e_2 \quad e_2 = e_3}{e_1 = e_3}$$

This means that if we have a proof of $3+5=8$ and a proof that $8=2^3$, for example, then we can combine these two proofs to form a proof of the equation $3+5=2^3$. Technically, the formulas above the horizontal line are called the *antecedents* of the proof rule, and the formula below the line the *consequent.*

Formally, a proof can be defined as a sequence of formulas, with each formula either an axiom or following from previous formulas by a single inference rule. This formal definition of proof is often useful; it lends itself to arguments by natural-number induction on the length of the proof (*i.e.*, the length of the sequence). An alternate view is often more insightful, however. Since an inference rule generally has a list of antecedents and one consequent, it is easy to visualize a proof as a form of tree with leaves and internal nodes labeled by formulas. More specifically, we think of each axiom as a possible leaf and each inference rule

$$\frac{A_1 \quad \dots \quad A_n}{B}$$

as a possible internal tree node whose subtrees must be proofs of $A_1, \dots, A_n$. Since this preserves the ordinary orientation of inference rules, it is common to draw proof trees with the trunk or root at the bottom. Thus if we construct a proof of B from proofs of $A_1, \dots, A_n$, we would draw the resulting tree in the form



where each of the triangular shapes above the line represents a proof tree whose conclusion is one of the antecedents of the proof rule. One useful consequence of thinking of proofs as trees is that it suggests a form of induction that is essentially the same as induction on the structure of trees.

If we use induction on the height of trees as in Example 1.8.3, the base case establishes the property for each axiom. The induction step will be to assume that the property holds for any shorter proof, and establish the property for a proof ending in an inference rule. This leads to the following form of structural induction on proofs:

> **Structural Induction on Proofs:** To prove that $P(\pi)$ is true for every proof π in some proof system, it is sufficient to show that P holds for every axiom of the proof system and then, assuming that P holds for proofs $\pi_1, \dots, \pi_k$, prove that $P(\pi)$ for any proof that ends by extending one or more of the proofs $\pi_1, \dots, \pi_k$ with one inference rule.

Example 1.8.7 We will illustrate induction on the structure of proofs using a simple proof system for inequalities $e \le e'$, where e and e' are generated by the example grammar

$$e \quad ::= \quad 0 \mid 1 \mid v \mid e + e \mid e * e$$

also used above. The proof system has two axioms, one stating that $\le$ is reflexive

$$(\textit{refl}) \qquad e \le e$$

and the other that 0 is less than any other expression.

$$(0 \le) \qquad 0 \le e$$

There is a transitivity inference rule and two additional inference rules giving monotonicity of addition and multiplication.

$$(\textit{trans}) \qquad \frac{e \le e' \quad e' \le e''}{e \le e''}$$

$$(+mon) \qquad \frac{e_1 \leq e_2 \quad e_3 \leq e_4}{e_1 + e_3 \leq e_2 + e_4}$$

$$(*mon) \qquad \frac{e_1 \leq e_2 \quad e_3 \leq e_4}{e_1 * e_3 \leq e_2 * e_4}$$

This is a relatively weak proof system for the ordinary ordering on natural numbers. However, it is sufficient to illustrate some basic ideas.

A very common property to establish for a proof system is that every provable formula is true (under some specific interpretation of formulas). This is called *soundness* of the proof system. We illustrate induction on the structure of proofs by proving soundness of our example proof system for $\leq$, interpreting arithmetic expressions over the natural numbers in the usual way. More precisely, we show that the property

$$P(\pi) \stackrel{def}{=} \begin{array}{l}\text{if } \pi \text{ is a proof of } e \leq e' \text{ then for all values of variables,}\\ \text{the value of } e \text{ is} \leq \text{the value of } e'\end{array}$$

holds for every proof π in our system. In other words, if we can prove a formula, then the formula is true, for all possible values of the variables that occur in the formula.

The base cases are to establish the property P for each of the axioms. This is easy. Whatever values we give to the variables occurring in an expression e, the value of e is some specific natural number. Therefore, we always have $e \leq e$ and $0 \leq e$.

There are three induction steps. We show the cases for $(+mon)$ and $(*mon)$ and leave the case for $(trans)$ to the reader. Suppose that we may prove inequalities $e_1 \leq e_2$ and $e_3 \leq e_4$. Let us pick values for the variables that occur in $e_1, \ldots, e_4$ and call the resulting values of the expressions $n_1, \ldots, n_4$. By the inductive hypothesis, we have $n_1 \leq n_2$ and $n_3 \leq n_4$. It is easy to see that therefore $n_1 + n_3 \leq n_2 + n_4$ and similarly $n_1 * n_3 \leq n_2 * n_4$. Since this reasoning applies for all possible values of the variables, the property holds for proofs ending in $(+mon)$ and $(*mon)$. Since the $(trans)$ case is given as Exercise 1.8.10, this concludes the inductive proof. ■

The proof system in Example 1.8.7 is not only useful for proving inequalities that are true for all values of the variables, but also for proving inequalities that follow from additional assumptions. Such additional assumptions are sometimes called *nonlogical axioms.* Intuitively, these are assumptions that are true about a certain situation, but not always true. For example, if we assume that $1 \leq x$ and $x \leq 5$, we could prove that (under these assumptions) $1 + x \leq x + 5$ using $(+mon)$. This idea also applies to other proof systems, of course. In general, a *proof from assumptions* or *proof from nonlogical axioms* is a proof tree as described above, with each leaf either a (logical) axiom of the proof system or a nonlogical axiom (one of the given assumptions) and each internal node an inference rule of the proof system.

Exercise 1.8.8 Consider the set of variable-free expressions given by the following grammar:

$$e ::= 0 \mid 2 \mid e + e \mid e * e$$

where $+$ is interpreted as addition and $*$ multiplication, as usual. Use induction on the structure of expressions to show that the value of every expression produced by this grammar is an even number.

Exercise 1.8.9 We can apply structural induction to trees if we think of them as generated by the grammar

$$t \quad ::= \quad nil \mid leaf \mid node(t, t)$$

Use structural induction on trees (with two base cases and one induction step) to prove that the size of a binary tree is at most 2^h, where h is the height of the tree.

Exercise 1.8.10 Prove the (*trans*) case of the induction on proofs given in Example 1.8.7.

Exercise 1.8.11 Use induction on the structure of proofs to show that if $e \leq e'$ is provable (using the axioms and rules given in this section), then e' is at least as long an expression (counting symbols) as e. Use the form of inductive proof illustrated in Example 1.8.7.

1.8.3 Well-founded induction

All of the forms of induction we have discussed so far are instances of a general form of induction on what are called "well-founded relations." Although it is seldom necessary to appeal to the general form of well-founded induction, this is occasionally the best way to carry out a proof. Well-founded relations are also important in computer science for their connection with termination of programs (see Section 3.7.3).

A *well-founded relation* on a set A is a binary relation $\prec$ on A with the property that there is no infinite descending sequence $a_0 \succ a_1 \succ a_2 \succ \ldots$. An example is the relation $i \prec j$ if $j = i + 1$ on the natural numbers. As this example illustrates, a well-founded relation does not have to be transitive. It is easy to see that every well-founded relation is irreflexive, *i.e.*, we do not have $a \prec a$ for any $a \in A$. The reason is that if $a \prec a$, then there is an infinite descending sequence $a \succ a \succ a \succ \ldots$

An equivalent definition is that a binary relation $\prec$ on A is well-founded iff every nonempty subset B of A has a minimal element, where $a \in B$ is *minimal* if there is no $a' \in B$ with $a' \prec a$. This is proved in the following lemma.

Lemma 1.8.12 *Let $\prec$ be a binary relation on set A. Then $\prec$ is well-founded iff every nonempty subset of A has a minimal element.*

Proof Suppose that $\prec$ is a well-founded relation on A and let $B \subseteq A$ be any nonempty subset. We will show that B has a minimal element. The easiest way to do this is to argue by contradiction. If B does not have a minimal element, then for every $a \in B$ we can find some $a' \in B$ with $a' \prec a$. But in this case, we can construct an infinite decreasing sequence $a_0 \succ a_1 \succ a_2 \succ \ldots$ starting with any $a_0 \in B$ and using the fact that no a_i can be minimal since B has no minimal element. This proves the first half of the lemma.

For the converse, suppose that every subset has a minimal element. Then there can be no infinite decreasing sequence $a_0 \succ a_1 \succ a_2 \succ \ldots$ since such a sequence would give us a set $\{a_0, a_1, a_2, \ldots\}$ without a minimal element. This completes the proof. ∎

Proposition 1.8.13 (Well-founded Induction) *Let $\prec$ be a well-founded binary relation on set A and let P be some property on A. If $P(a)$ holds whenever we have $P(b)$ for all $b \prec a$, then $P(a)$ is true for all $a \in A$.*

Proof Suppose that for every $a \in A$, we have $P(a)$ if $P(b)$ for all $b \prec a$. (In symbols, we are assuming $\forall a. (\forall b. (b \prec a \supset P(b)) \supset P(a))$.) We will show that $P(a)$ holds for all $a \in A$ by contradiction. If there is some $x \in A$ with $\neg P(x)$, then the set

$$B = \{ a \in A \mid \neg P(a) \}$$

Form of Induction	Well-founded Relation
Natural number induction, Form 1	$m \prec n$ if $m+1=n$
Natural number induction, Form 2	$m \prec n$ if $m<n$
Structural induction, Form 1	$e \prec e'$ if e is an immediate subexpression of e'
Structural induction, Form 2	$e \prec e'$ if e is a subexpression of e'
Induction on proofs	$\pi \prec \pi'$ if π is the subproof for some antecedent of the last inference rule in proof π'

Table 1.1: Well-founded relations for common forms of induction.

is nonempty. Therefore, by Lemma 1.8.12, the set B must have a minimal element $a \in B$. But since we therefore have $P(b)$ for all $b \prec a$, this contradicts the assumption $\forall b. (b \prec a \supset P(b)) \supset P(a)$. This proves the proposition. ∎

Table 1.1. lists the well-founded relation associated with each form of induction we have already considered. A property of well-founded relations that is easy to establish is that the transitive closure of any well-founded relation is also well-founded. This is helpful in understanding the parallel between the two forms of natural-number induction and the two forms of structural induction. In both cases, the well-founded relation giving rise to the second form is just the transitive closure of the relation associated with the first form of induction.

In each of the examples in Table 1.1, it is easy to see that the listed relation is in fact well-founded. For example, since each subexpression is shorter than the expression that contains it, there can be no infinite sequence of successively smaller subexpressions.

A useful class of well-founded relations that are more complicated than the ones in Table 1.1 are the *lexicographical orderings.* These are essentially dictionary-like orderings on sequences drawn from some ordered set. For simplicity, we just consider orderings on sequences of natural numbers.

A natural ordering on pairs of natural numbers is

$$\langle n,m\rangle \prec \langle n',m'\rangle \ \text{ iff } \ n<n' \text{ or } (n=n' \text{ and } m<m')$$

In words, we say one pair is less than another if either the first (or "most significant") numbers are ordered, or, if the first numbers are the same, the second numbers are ordered. If we only consider the single-digit numbers $0,\ldots,9$ and think of a pair $\langle n,m\rangle$ as the two-digit numeral nm, then this is the ordinary numeric ordering.

It is a little tricky to see that the relation $\prec$ on pairs of natural numbers is well-founded. We will argue that there is no infinite decreasing sequence. The trick is to arrange any decreasing sequence of pairs in a two-dimensional array, moving down when the first element decreases and across when the second decreases. Illustrated with specific numbers, the idea is to arrange any a decreasing sequence like this:

$$\begin{array}{lllllll}
\langle 5,6\rangle & \succ \langle 5,5\rangle & & & & & \\
 & \succ \langle 4,50\rangle & \succ \langle 4,40\rangle & \succ \langle 4,30\rangle & & & \\
 & & & \succ \langle 3,300\rangle & \succ \langle 3,250\rangle & & \\
 & & & & \succ \langle 2,500\rangle & \succ \langle 2,450\rangle & \succ \langle 2,449\rangle \quad \succ \langle 2,448\rangle \\
 & & & & & & \qquad\qquad\quad \cdots
\end{array}$$

For any decreasing sequence, we can see that no row can be infinite, since there is no infinite decreasing sequences of natural numbers. Therefore, if the table is going to be infinite, there must be infinitely many rows. But this is impossible since the first number must decrease each time we go down to another row.

We can generalize this ordering on pairs, which is technically called ω^2, to orderings on triples, sequences of length 4, and so on. We can also extend this ordering to sequences of differing lengths. Writing sequences of natural numbers in the form $\langle n_1, n_2, \ldots, n_k \rangle$, we define the more general ordering by

$$\langle n_1, n_2, \ldots, n_k \rangle \prec \langle m_1, m_2, \ldots, m_\ell \rangle \ \text{ iff } \left[\begin{array}{l} k < \ell \text{ or} \\ k = \ell \text{ and } \exists i \leq k \text{ with} \\ \qquad n_j = m_j \text{ for } j < i \text{ and } n_i < m_i \end{array} \right]$$

In words, we first order sequences by length. Then, for sequences of the same length, we use the natural generalization of the ordering on pairs above, comparing numbers from left to right until two differ. It is a bit harder to draw an "infinite dimensional matrix" showing that this order is well-founded, but the idea is essentially similar to the argument given for the pair ordering above.

Example 1.8.14 We illustrate well-founded induction by proving a "normalization property" of proofs in the system of Example 1.8.7. This example may seem contrived, but it actually corresponds quite closely to the kinds of reasoning about typing derivations and subtyping derivations that appear occasionally in later chapters of the book. The property we will prove is that if there is a proof of an inequality, possibly from some set of assumptions (as described at the end of Section 1.8.2), then there is a proof of the same inequality that does not use (*trans*) after two uses of $(+mon)$. (The same property also holds for $(*mon)$ but we will not take the time to show this.) This is best illustrated by example. If we assume $x \leq y \leq z$ and $x' \leq y' \leq z'$, then we can prove $x + x' \leq z + z'$ in two ways. The first,

$$\frac{\dfrac{x \leq y \quad x' \leq y'}{x + x' \leq y + y'} \qquad \dfrac{y \leq z \quad y' \leq z'}{y + y' \leq z + z'}}{x + x' \leq z + z'}$$

has two uses of transitivity before monotonicity of $+$, while the second

$$\frac{\dfrac{x \leq y \quad y \leq z}{x \leq z'} \qquad \dfrac{x' \leq y' \quad y' \leq z'}{x' \leq z'}}{x + x' \leq z + z'}$$

has two uses of (*trans*) first, followed by $(+mon)$. We will show that we can eliminate all occurrences of the first pattern of proof steps from any proof. The reason we might want to do this is that it makes it easier to search for proofs of an inequality. The general principle is that if we know that any provable formula has a proof of a certain form, then we can decide whether a formula is provable by looking for proofs of this certain form.

In outline, our inductive argument will proceed by showing that if a proof has two uses of $(+mon)$ before (*trans*), then we can eliminate this pattern of three steps in favor of two uses of (*trans*) followed by $(+mon)$. However, in order for this argument to be correct, we must have an inductive hypothesis that applies to the proof produced by this transformation. Since the result of the proof transformation has the same number of proof rules, we cannot use induction on the structure, length or number of proof steps in the proof. In addition, we cannot use induction on the number of patterns that we are trying to eliminate, since our proof transformation moves transitivity toward the leaves, and may introduce new patterns of the form we are trying to eliminate. We could try to do some form of induction on the number of uses of $(+mon)$ in the proof, which seems to involve some case analysis on the way in which these rules occur. However, the problem may be solved very simply by defining an appropriate well-founded relation on proofs.

We will order proofs by assigning a sequence of numbers to each proof and ordering these sequences lexicographically. For any proof π, we calculate the degree $deg(\pi)$ of π as follows. First, for each pattern of two uses of $(+mon)$ before $(trans)$, we count the number of $+$'s occurring in each of the expressions in the conclusion of $(trans)$. If the two expressions differ in the number of $+$'s, we can just take the maximum of the two numbers. Call this number the *measure* of this pattern of steps. Now, for each measure up to the maximum measure occurring, count up the number of patterns with this measure and sort this sequence of counts (some possibly zero) in order of decreasing associated measure. This sequence of numbers will be our "degree" of the proof. With this form of induction, we can easily carry out the proof. The reason why this works is that whenever we replace two uses of $(+mon)$ before $(trans)$ in favor of two uses of $(trans)$ followed by $(+mon)$, we reduce the measure of any new pattern of two uses of $(+mon)$ before $(trans)$ that is created. ∎

Exercise 1.8.15 A *string* s over alphabet Σ is a finite sequence of elements of Σ. In this problem, we use capitals of the ordinary (Roman) alphabet, so that any string s can be written $s = a_1 a_2 \dots a_k$ where each a_i is one of the twenty-six letters A, B, C, ..., Z. A special case is the empty string, written ϵ. Three relations on strings are given below, two well-founded and the other not. For each well-founded relation, give a short explanation of why it is well-founded. For the relation that is not well-founded, give an infinite decreasing sequence.

(a) Relation $\prec_1$ is the ordinary alphabetic order on strings (the "dictionary" ordering). This is characterized by the following two axioms.

$$\begin{array}{rcl} \epsilon \prec_1 s & \textit{iff} & s \neq \epsilon \\ a_1 s \prec_1 a_2 t & \textit{iff} & a_1 \text{ comes before } a_2 \text{ in the alphabet, } \textit{or} \\ & & [a_1 = a_2 \textit{ and } s \prec_1 t] \end{array}$$

(b) The second relation orders strings by length:

$$a_1 \dots a_k \prec_2 b_1 \dots b_\ell \quad \textit{iff} \quad k < \ell.$$

(c) The third relation, $\prec_3$, combines the previous two by using $\prec_2$ on strings of different length, and $\prec_1$ on strings of the same length.

$$a_1 \dots a_k \prec_3 b_1 \dots b_\ell \quad \textit{iff} \quad k < \ell \;\; \textit{or} \;\; [k = \ell \textit{ and } a_1 \dots a_k \prec_1 b_1 \dots b_\ell]$$