chapter 3

Construction Techniques

When we build, let us think that we build forever.

—John Ruskin (1819–1900)

To construct an object, we need some kind of description. If we're lucky, the description might include a construction technique. Otherwise, we may need to use our wits and our experience to construct the object. This chapter focuses on gaining some construction experience.

The only way to learn a technique is to use it on a wide variety of problems. We'll present each technique in the framework of objects that occur in computer science, and as we go along, we'll extend our knowledge of these objects. We'll begin by introducing the technique of inductive definition for sets. Then we'll discuss techniques for describing recursively defined functions and procedures. Last but not least, we'll introduce grammars for describing sets of strings.

There are usually two parts to solving a problem. The first is to guess at a solution and the second is to verify that the guess is correct. The focus of this chapter is to introduce techniques to help us make good guesses. We'll usually check a few cases to satisfy ourselves that our guesses are correct. In the next chapter we'll study inductive proof techniques that can be used to actually prove correctness of claims about objects constructed by the techniques of this chapter.

chapter guide

3.1 (Inductively Defined Sets) introduces the inductive definition technique. We'll apply the technique by defining various sets of numbers, strings, lists, binary trees, and Cartesian products.

3.2 (Recursive Functions and Procedures) introduces the technique of recursive definition for functions and procedures. We'll apply the technique to functions and procedures that process numbers, strings, lists, and binary trees.

We'll solve the repeated element problem and the power set problem, and we'll construct some functions for infinite sequences.

3.3 (Grammars) introduces the idea of a grammar as a way to describe a language. We'll see that grammars describe the strings of a language in an inductive fashion, and we'll see that they provide recursive rules for testing whether a string belongs to a language.

3.1 Inductively Defined Sets

When we write down an informal statement such as $A = \{3, 5, 7, 9, \ldots\}$, most of us will agree that we mean the set $A = \{2k + 3 \mid k \in \mathbb{N}\}$. Another way to describe A is to observe that $3 \in A$, that $x \in A$ implies $x + 2 \in A$, and that the only way an element gets in A is by the previous two steps. This description has three ingredients, which we'll state informally as follows:

1. There is a starting element (3 in this case).
2. There is a construction operation to build new elements from existing elements (addition by 2 in this case).
3. There is a statement that no other elements are in the set.

Definition of Inductive Definition

This process is an example of an *inductive definition* of a set. The set of objects defined is called an *inductive set.* An inductive set consists of objects that are constructed, in some way, from objects that are already in the set. So nothing can be constructed unless there is at least one object in the set to start the process. Inductive sets are important in computer science because the objects can be used to represent information and the construction rules can often be programmed. We give the following formal definition.

An inductive definition of a set S consists of three steps: **(3.1)**

Basis: Specify one or more elements of S.

Induction: Give one or more rules to construct new elements of S from existing elements of S.

Closure: State that S consists exactly of the elements obtained by the basis and induction steps. This step is usually assumed rather than stated explicitly.

The closure step is a very important part of the definition. Without it, there could be lots of sets satisfying the first two steps of an inductive definition. For example, the two sets $\mathbb{N}$ and $\{3, 5, 7, \ldots\}$ both contain the number 3, and if x is in either set, then so is $x + 2$. It's the closure statement that tells us that the only set defined by the basis and induction steps is $\{3, 5, 7, \ldots\}$. So the closure statement tells us that we're defining exactly one set, namely, the smallest set satisfying the basis and induction steps. We'll always omit the specific mention of closure in our inductive definitions.

The *constructors* of an inductive set are the basis elements and the rules for constructing new elements. For example, the inductive set $\{3, 5, 7, 9, \ldots\}$ has two constructors, the number 3 and the operation of adding 2 to a number.

For the rest of this section we'll use the technique of inductive definition to construct sets of objects that are often used in computer science.

3.1.1 Numbers

The set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ is an inductive set. Its basis element is 0, and we can construct a new element from an existing one by adding the number 1. So we can write an inductive definition for $\mathbb{N}$ in the following way.

Basis: $0 \in \mathbb{N}$.

Induction: If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

The constructors of $\mathbb{N}$ are the integer 0 and the operation that adds 1 to an element of $\mathbb{N}$. The operation of adding 1 to n is called the *successor* function, which we write as

$$\text{succ}(n) = n + 1.$$

Using the successor function, we can rewrite the induction step in the above definition of $\mathbb{N}$ in the alternative form

$$\text{If } n \in \mathbb{N}, \text{ then } \text{succ}(n) \in \mathbb{N}.$$

So we can say that $\mathbb{N}$ is an inductive set with two constructors, 0 and succ.

example 3.1 Some Familiar Odd Numbers

We'll give an inductive definition of $A = \{1, 3, 7, 15, 31, \ldots\}$. Of course, the basis case should place 1 in A. If $x \in A$, then we can construct another element of A with the expression $2x + 1$. So the constructors of A are the number 1 and the operation of multiplying by 2 and adding 1. An inductive definition of A can be written as follows:

Basis: $1 \in A$.

Induction: If $x \in A$, then $2x + 1 \in A$.

end example

example 3.2 Some Even and Odd Numbers

Is the following set inductive?

$$A = \{2, 3, 4, 7, 8, 11, 15, 16, \ldots\}.$$

It might be easier if we think of A as the union of the two sets

$$B = \{2, 4, 8, 16, \ldots\} \text{ and } C = \{3, 7, 11, 15, \ldots\}.$$

Both of these sets are inductive. The constructors of B are the number 2 and the operation of multiplying by 2. The constructors of C are the number 3 and the operation of adding by 4. We can combine these definitions to give an inductive definition of A.

Basis: $2, 3 \in A$.

Induction: If $x \in A$ and x is odd, then $x + 4 \in A$.

If $x \in A$ and x is even, then $2x \in A$.

This example shows that there can be more than one basis element, more than one induction rule, and tests can be included.

end example

example 3.3 Communicating with a Robot

Suppose we want to communicate the idea of the natural numbers to a robot that knows about functions, has a loose notion of sets, and can follow an inductive definition. Symbols like 0, 1, ..., and + make no sense to the robot. How can we convey the idea of $\mathbb{N}$? We'll tell the robot that N is the name of the set we want to construct.

Suppose we start by telling the robot to put the symbol 0 in N. For the induction case we need to tell the robot about the successor function. We tell the robot that $s : N \to N$ is a function, and whenever an element $x \in N$, then put the element $s(x) \in N$. After a pause, the robot says, "$N = \{0\}$ because I'm letting s be the function defined by $s(0) = 0$."

Since we don't want $s(0) = 0$, we have to tell the robot that $s(0) \neq 0$. Then the robot says, "$N = \{0, s(0)\}$ because $s(s(0)) = 0$." So we tell the robot that $s(s(0)) \neq 0$. Since this could go on forever, let's tell the robot that $s(x)$ does not equal any previously defined element. Do we have it? Yes. The robot responds with "$N = \{0, s(0), s(s(0)), s(s(s(0))), \ldots\}$." So we can give the robot the following definition:

Basis: $0 \in N$.

Induction: If $x \in N$, then put $s(x) \in N$, where $s(x) \neq 0$ and $s(x)$ is not equal to any previously defined element of N.

This definition of the natural numbers—along with a closure statement—is due to the mathematician and logician Giuseppe Peano (1858–1932).

end example

example 3.4 Communicating with Another Robot

Suppose we want to define the natural numbers for a robot that knows about sets and can follow an inductive definition. How can we convey the idea of $\mathbb{N}$ to the robot? Since we can use only the notation of sets, let's use $\varnothing$ to stand for the number 0.

What about the number 1? Can we somehow convey the idea of 1 using the empty set? Let's let $\{\varnothing\}$ stand for 1. What about 2? We can't use $\{\varnothing, \varnothing\}$, because $\{\varnothing, \varnothing\} = \{\varnothing\}$. Let's let $\{\varnothing, \{\varnothing\}\}$ stand for 2 because it has two distinct elements. Notice the little pattern we have going: If s is the set standing for a number, then $s \cup \{s\}$ stands for the successor of the number.

Starting with $\varnothing$ as the basis element, we have an inductive definition. Letting *Nat* be the set that we are defining for the robot, we have the following inductive definition.

Basis: $\varnothing \in Nat$.

Induction: If $s \in Nat$, then $s \cup \{s\} \in Nat$.

For example, since 2 is represented by the set $\{\varnothing, \{\varnothing\}\}$, the number 3 is represented by the set

$$\{\varnothing, \{\varnothing\}\} \cup \{\{\varnothing, \{\varnothing\}\}\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}.$$

This is not fun. After a while we might try to introduce some of our own notation to the robot. For example, we'll introduce the decimal numerals in the following way.

$$\begin{aligned} 0 &= \varnothing, \\ 1 &= 0 \cup \{0\}, \\ 2 &= 1 \cup \{1\}, \\ &\vdots \end{aligned}$$

Now we can think about the natural numbers in the following way.

$$\begin{aligned} 0 &= \varnothing, \\ 1 &= 0 \cup \{0\} = \varnothing \cup \{0\} = \{0\}, \\ 2 &= 1 \cup \{1\} = \{0\} \cup \{1\} = \{0, 1\}, \\ &\vdots \end{aligned}$$

Therefore, each number is the set of numbers that precedes it.

end example

3.1.2 Strings

We often define strings of things inductively without even thinking about it. For example, in high school algebra we might say that an algebraic expression is either a number or a variable, and if A and B are algebraic expressions, then so are (A), $A + B$, $A - B$, AB, and $A \div B$. So the set of algebraic expressions is a set of strings. For example, if x and y are variables, then the following strings are algebraic expressions.

$$x, \quad y, \quad 25, \quad 25x, \quad x + y, \quad (4x + 5y), \quad (x + y)(2yx), \quad 3x \div 4.$$

If we like, we can make our definition more formal by specifying the basis and induction parts. For example, if we let E denote the set of algebraic expressions as we have described them, then we have the following inductive definition for E.

Basis: If x is a variable or a number, then $x \in E$.

Induction: If $A, B \in E$, then (A), $A + B$, $A - B$, AB, $A \div B \in E$.

Let's recall that for an alphabet A, the set of all strings over A is denoted by A^*. This set has the following inductive definition.

All Strings over A (3.2)

Basis: $\Lambda \in A^*$.

Induction: If $s \in A^*$ and $a \in A$, then $as \in A^*$.

We should note that when we place two strings next to each other in juxtaposition to form a new string, we are concatenating the two strings. So, from a computational point of view, concatenation is the operation we are using to construct new strings.

Recall that any set of strings is called a *language*. If A is an alphabet, then any language over A is one of the subsets of A^*. Many languages can be defined inductively. Here are some examples.

example 3.5 Three Languages

We'll give an inductive definition for each of three languages.

1. $S = \{a, ab, abb, abbb, \ldots\} = \{ab^n \mid n \in \mathbb{N}\}$.

Informally, we can say that the strings of S consist of the letter a followed by zero or more b's. But we can also say that the letter a is in S, and if x is a string in S, then so is xb. This gives us an inductive definition for S.

Basis: $a \in S$.

Induction: If $x \in S$, then $xb \in S$.

2. $S = \{\Lambda, ab, aabb, aaabbb, \ldots\} = \{a^n b^n \mid n \in \mathbb{N}\}$.

Informally, we can say that the strings of S consist of any number of a's followed by the same number of b's. But we can also say that the empty string Λ is in S, and if x is a string in S, then so is axb. This gives us an inductive definition for S.

Basis: $\Lambda \in S$.

Induction: If $x \in S$, then $axb \in S$.

3. $S = \{\Lambda, ab, abab, ababab, \ldots\} = \{(ab)^n \mid n \in \mathbb{N}\}$.

Informally, we can say that the strings of S consist of any number of ab pairs. But we can also say that the empty string Λ is in S, and if x is a string in S, then so is abx. This gives us an inductive definition for S.

Basis: $\Lambda \in S$.

Induction: If $x \in S$, then $abx \in S$.

end example

example 3.6 Decimal Numerals

Let's give an inductive definition for the set of decimal numerals. Recall that a decimal numeral is a nonempty string of decimal digits. For example, 2340 and 002965 are decimal numerals. If we let D denote the set of decimal numerals, we can describe D by saying that any decimal digit is in D, and if x is in D and d is a decimal digit, then dx is in D. This gives us the following inductive definition for D:

Basis: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \subseteq D$.

Induction: If $x \in D$ and d is a decimal digit, then $dx \in D$.

end example

3.1.3 Lists

Recall that a list is an ordered sequence of elements. Let's try to find an inductive definition for the set of lists with elements from a set A. In Chapter 1 we denoted the set of all lists over A by lists(A), and we'll continue to do so. We also mentioned that from a computational point of view the only parts of a nonempty list that can be accessed randomly are its *head* and its *tail*. Head and tail are sometimes called *destructors*, since they are used to destroy a list (take it apart). For example, the list $\langle x, y, z\rangle$ has x as its head and $\langle y, z\rangle$ as its tail, which we write as

$$\text{head}(\langle x, y, z\rangle) = x \quad \text{and} \quad \text{tail}(\langle x, y, z\rangle) = \langle y, z\rangle.$$

We also introduced the operation "cons" to construct lists, where if h is an element and t is a list, the new list whose head is h and whose tail is t is represented by the expression

$$\text{cons}(h, t).$$

So cons is a constructor of lists. For example, we have

$$\text{cons}(x, \langle y, z\rangle) = \langle x, y, z\rangle$$
$$\text{cons}(x, \langle\ \rangle) = \langle x\rangle.$$

The operations cons, head, and tail work nicely together. For example, we can write

$$\langle x, y, z\rangle = \text{cons}(x, \langle y, z\rangle) = \text{cons}(\text{head}(\langle x, y, z\rangle), \text{tail}(\langle x, y, z\rangle)).$$

So if L is any nonempty list, then we have the equation

$$L = \text{cons}(\text{head}(L), \text{tail}(L)).$$

Now we have the proper tools, so let's get down to business and write an inductive definition for lists(A). Informally, we can say that lists(A) is the set of all ordered sequences of elements taken from the set A. But we can also say that $\langle\ \rangle$ is in lists(A), and if L is in lists(A), then so is cons(a, L) for any a in A. This gives us an inductive definition for lists(A), which we can state formally as follows.

All Lists over A (3.3)

Basis: $\langle\ \rangle \in \text{lists}(A)$.

Induction: If $x \in A$ and $L \in \text{lists}(A)$, then $\text{cons}(x, L) \in \text{lists}(A)$.

example 3.7 List Membership

Let $A = \{a, b\}$. We'll use (3.3) to show how some lists become members of lists(A). The basis case puts $\langle\ \rangle \in \text{lists}(A)$. Since $a \in A$ and $\langle\ \rangle \in \text{lists}(A)$, the induction step gives

$$\langle a\rangle = \text{cons}(a, \langle\ \rangle) \in \text{lists}(A).$$

In the same way we get $\langle b\rangle \in \text{lists}(A)$. Now since $a \in A$ and $\langle a\rangle \in \text{lists}(A)$, the induction step puts $\langle a, a\rangle \in \text{lists}(A)$. Similarly, we get $\langle b, a\rangle$, $\langle a, b\rangle$, and $\langle b, b\rangle$ as elements of lists(A), and so on.

end example

A Notational Convenience

It's convenient when working with lists to use an infix notation for cons to simplify the notation for list expressions. We'll use the double colon symbol ::, so that the infix form of cons(x, L) is

$$x :: L.$$

For example, the list $\langle a, b, c\rangle$ can be constructed using cons as

$$\begin{aligned}\text{cons}(a, \text{cons}(b, \text{cons}(c, \langle\ \rangle))) &= \text{cons}(a, \text{cons}(b, \langle c\rangle)) \\ &= \text{cons}(a, \langle b, c\rangle) \\ &= \langle a, b, c\rangle.\end{aligned}$$

Using the infix form, we construct $\langle a, b, c\rangle$ as follows:

$$a :: (b :: (c :: \langle\ \rangle)) = a :: (b :: \langle c\rangle) = a :: \langle b, c\rangle = \langle a, b, c\rangle.$$

The infix form of cons allows us to omit parentheses by agreeing that :: is right associative. In other words, $a :: b :: L = a :: (b :: L)$. Thus we can represent the list $\langle a, b, c\rangle$ by writing

$$a :: b :: c :: \langle\ \rangle \text{ instead of } a :: (b :: (c :: \langle\ \rangle)).$$

Many programming problems involve processing data represented by lists. The operations cons, head, and tail provide basic tools for writing programs to create and manipulate lists. So they are necessary for programmers. Now let's look at a few examples.

example 3.8 Lists of Binary Digits

Suppose we need to define the set S of all nonempty lists over the set $\{0, 1\}$ with the property that adjacent elements in each list are distinct. We can get an idea about S by listing a few elements:

$$S = \{\langle 0\rangle, \langle 1\rangle, \langle 1, 0\rangle, \langle 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 1\rangle, \ldots\}.$$

Let's try $\langle 0 \rangle$ and $\langle 1 \rangle$ as basis elements of S. Then we can construct a new list from a list $L \in S$ by testing whether head(L) is 0 or 1. If head(L) = 0, then we place 1 at the left of L. Otherwise, we place 0 at the left of L. So we can write the following inductive definition for S.

Basis: $\langle 0 \rangle, \langle 1 \rangle \in S$.

Induction: If $L \in S$ and head(L) = 0, then cons(1, L) $\in S$.
If $L \in S$ and head(L) = 1, then cons(0, L) $\in S$.

The infix form of these induction rules looks like

If $L \in S$ and head(L) = 0, then $1 :: L \in S$.
If $L \in S$ and head(L) = 1, then $0 :: L \in S$.

end example

example 3.9 Lists of Letters

Suppose we need to define the set S of all lists over $\{a, b\}$ that begin with the single letter a followed by zero or more occurrences of b. We can describe S informally by writing a few of its elements:

$$S = \{\langle a \rangle, \langle a, b \rangle, \langle a, b, b \rangle, \langle a, b, b, b \rangle, \ldots\}.$$

It seems appropriate to make $\langle a \rangle$ the basis element of S. Then we can construct a new list from any list $L \in S$ by attaching the letter b on the right end of L. But cons places new elements at the left end of a list. We can overcome the problem in the following way:

$$\text{If } L \in S, \text{ then cons}(a, \text{cons}(b, \text{tail}(L)) \in S.$$

In infix form the statement reads as follows:

$$\text{If } L \in S, \text{ then } a :: b :: \text{tail}(L) \in S.$$

For example, if $L = \langle a \rangle$, then we construct the list

$$a :: b :: \text{tail}(\langle a \rangle) = a :: b :: \langle\ \rangle = a :: \langle b \rangle = \langle a, b \rangle.$$

So we have the following inductive definition of S:

Basis: $\langle a \rangle \in S$.

Induction: If $L \in S$, then $a :: b :: \text{tail}(L) \in S$.

end example

example 3.10 All Possible Lists

Can we find an inductive definition for the set of all possible lists over $\{a, b\}$, including lists that can contain other lists? Suppose we start with lists having a small number of symbols, including the symbols $\langle$ and $\rangle$. Then, for each $n \geq 2$, we can write down the lists made up of n symbols (not including commas). Figure 3.1 shows these listings for the first few values of n.

If we start with the empty list $\langle\ \rangle$, then with a and b we can construct three more lists as follows:

$$a :: \langle\ \rangle = \langle a \rangle,$$
$$b :: \langle\ \rangle = \langle b \rangle,$$
$$\langle\ \rangle :: \langle\ \rangle = \langle\langle\ \rangle\rangle.$$

Now if we take these three lists together with $\langle\ \rangle$, then with a and b we can construct many more lists. For example,

$$a :: \langle a \rangle = \langle a, a \rangle,$$
$$\langle a \rangle :: \langle\ \rangle = \langle\langle a \rangle\rangle,$$
$$\langle\langle\ \rangle\rangle :: \langle b \rangle = \langle\langle\langle\ \rangle\rangle, b \rangle,$$
$$\langle b \rangle :: \langle\langle\ \rangle\rangle = \langle\langle b \rangle, \langle\ \rangle\rangle.$$

Using this idea, we'll make an inductive definition for the set S of all possible lists over $\{a, b\}$.

Basis: $\langle\ \rangle \in S$. (3.4)

Induction: If $x \in \{a, b\} \cup S$ and $L \in S$, then $x :: L \in S$.

end example

2	3	4	5	6
$\langle\ \rangle$	$\langle a \rangle$	$\langle\langle\ \rangle\rangle$	$\langle\langle a \rangle\rangle$	$\langle\langle\langle\ \rangle\rangle\rangle$
	$\langle b \rangle$	$\langle a, a \rangle$	$\langle\langle b \rangle\rangle$	$\langle\langle\ \rangle, \langle\ \rangle\rangle$
		$\langle a, b \rangle$	$\langle\langle\ \rangle, a \rangle$	$\langle a, a, \langle\ \rangle\rangle$
		$\langle b, a \rangle$	$\langle\langle\ \rangle, b \rangle$	$\langle a, \langle\ \rangle, a \rangle$
		$\langle b, b \rangle$	$\langle a, \langle\ \rangle\rangle$	$\langle\langle\ \rangle, a, a \rangle$
			$\langle b, \langle\ \rangle\rangle$	$\langle a, b, \langle\ \rangle\rangle$
			$\langle a, a, a \rangle$	$\langle a, b, a, b \rangle$
			⋮	⋮

Figure 3.1 A listing of lists by size.

3.1.4 Binary Trees

Recall that a binary tree is either empty or it has a root with a left and right subtree, each of which is a binary tree. This is an informal inductive description of the set of binary trees. To give a formal definition and to work with binary trees, we need some operations to pick off parts of a tree and to construct new trees.

In Chapter 1 we represented binary trees by lists, where the empty binary tree is denoted by $\langle\ \rangle$ and a nonempty binary tree is denoted by the list $\langle L, x, R\rangle$, where x is the root, L is the left subtree, and R is the right subtree. This gives us the ingredients for a more formal inductive definition of the set of all binary trees.

For convenience we'll let tree(L, x, R) denote the binary tree with root x, left subtree L, and right subtree R. If we still want to represent binary trees as lists, then of course we can write

$$\text{tree}(L, x, R) = \langle L, x, R\rangle.$$

Now suppose A is any set. Then we can describe the set B of all binary trees whose nodes come from A by saying that $\langle\ \rangle$ is in B, and if L and R are in B, then so is tree(L, a, R) for any a in A. This gives us an inductive definition, which we can state formally as follows.

All Binary Trees over A (3.5)

Basis: $\langle\ \rangle \in B$.

Induction: If $x \in A$ and $L, R \in B$, then tree$(L, x, R) \in B$.

We also have destructor operations for binary trees. We'll let *left*, *root*, and *right* denote the operations that return the left subtree, the root, and the right subtree, respectively, of a nonempty tree. For example, if

$$T = \text{tree}(L, x, R), \text{ then } \text{left}(T) = L, \text{ root}(T) = x, \text{ and } \text{right}(T) = R.$$

So for any nonempty binary tree T we have

$$T = \text{tree}(\text{left}(T), \text{root}(T), \text{right}(T)).$$

example 3.11 Binary Trees of Twins

Let $A = \{0, 1\}$. Suppose we need to work with the set *Twins* of all binary trees T over A that have the following property: The left and right subtrees of each node in T are identical in structure and node content. For example, *Twins* contains the empty tree and any single-node tree. *Twins* also contains the two trees shown in Figure 3.2.

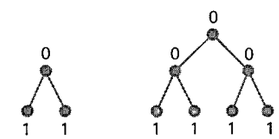


Figure 3.2 Twins as subtrees.

We can give an inductive definition of *Twins* by simply making sure that each new tree has the same left and right subtrees. Here's the definition:

Basis: $\langle\ \rangle \in$ *Twins*.

Induction: If $x \in A$ and $T \in$ *Twins*, then tree$(T, x, T) \in$ *Twins*.

end example

example 3.12 Binary Trees of Opposites

Let $A = \{0, 1\}$, and suppose that *Opps* is the set of all nonempty binary trees T over A with the following property: The left and right subtrees of each node of T have identical structures, but the 0's and 1's are interchanged. For example, each single node tree is in *Opps*, as well as the two trees shown in Figure 3.3.

Since our set does not include the empty tree, the two singleton trees with nodes 1 and 0 should be the basis trees in *Opps*. The inductive definition of *Opps* can be given as follows:

Basis: tree$(\langle\ \rangle, 0, \langle\ \rangle)$, tree$(\langle\ \rangle, 1, \langle\ \rangle) \in$ *Opps*.

Induction: Let $x \in A$ and $T \in$ *Opps*.
If root$(T) = 0$, then
tree$(T, x,$ tree$(\text{right}(T), 1, \text{left}(T))) \in$ *Opps*.
Otherwise,
tree$(T, x,$ tree$(\text{right}(T), 0, \text{left}(T))) \in$ *Opps*.

Does this definition work? Try out some examples. See whether the definition builds the four possible three-node trees.

end example

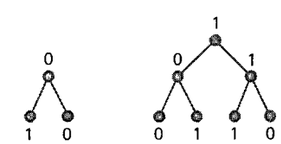


Figure 3.3 Opposites as subtrees.

3.1.5 Cartesian Products of Sets

Let's consider the problem of finding inductive definitions for subsets of the Cartesian product of two sets. For example, the set $\mathbb{N} \times \mathbb{N}$ can be defined inductively by starting with the pair $(0, 0)$ as the basis element. Then, for any pair (x, y) in the set, we can construct the following two pairs.

$$(x, y + 1) \text{ and } (x + 1, y).$$

It seems clear that this definition will define all elements of $\mathbb{N} \times \mathbb{N}$, although some pairs will be defined more than once. For example, the graph in Figure 3.4 shows four points. Notice that the point for the pair $(x + 1, y + 1)$ is constructed from $(x, y + 1)$, but it is also constructed from $(x + 1, y)$.

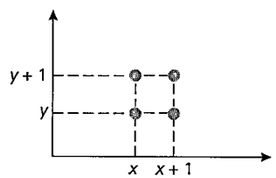


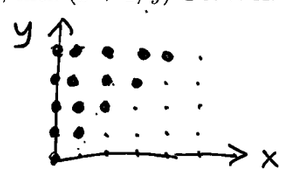
Figure 3.4 Four integer points.

example 3.13 Cartesian Product

A Cartesian product can be defined inductively if at least one set of the product can be defined inductively. For example, if A is any set, then we have the following inductive definition of $\mathbb{N} \times A$:

Basis: $(0, a) \in \mathbb{N} \times A$ for all $a \in A$.

Induction: If $(x, y) \in \mathbb{N} \times A$, then $(x + 1, y) \in \mathbb{N} \times A$.



end example

example 3.14 Part of a Plane

Let $S = \{(x, y) | \ x, y \in \mathbb{N} \text{ and } x \leq y\}$. From the point of view of a plane, S is the set of points in the first quadrant with integer coordinates on or above the main diagonal. We can define S inductively as follows:

Basis: $(0, 0) \in S$.

Induction: If $(x, y) \in S$, then $(x, y + 1), (x + 1, y + 1) \in S$.

For example, we can use $(0, 0)$ to construct $(0, 1)$ and $(1, 1)$. From $(0, 1)$ we construct $(0, 2)$ and $(1, 2)$. From $(1, 1)$ we construct $(1, 2)$ and $(2, 2)$. So some pairs get defined more than once.

end example

example 3.15 Describing an Area

Suppose we need to describe some area as a set of points. From a computational point of view, the area will be represented by discrete points, like pixels on a computer screen. So we can think of the area as a set of ordered pairs (x, y) forming a subset of $\mathbb{N} \times \mathbb{N}$.

To keep things simple we'll describe the area A under the curve of a function f between two points a and b on the x-axis. Figure 3.5 shows a general picture of the area A.

So the area A can be described as the following set of points, where $a, b \in \mathbb{N}$, and $f : \mathbb{N} \to \mathbb{N}$.

$$A = \{(x, y) \ | x, y \in \mathbb{N}, \ a \leq x \leq b, \text{ and } 0 \leq y \leq f(x)\}.$$

There are several ways we might proceed to give an inductive definition of A. For example, we can start with the point $(a, 0)$ on the x-axis. From $(a, 0)$ we can construct the column of points above it and the point $(a + 1, 0)$, from which the next column of points can be constructed. Here's the definition.

Basis: $(a, 0) \in A$.

Induction: If $(x, 0) \in A$ and $x < b$, then $(x + 1, 0) \in A$.
If $(x, y) \in A$ and $y < f(x)$, then $(x, y + 1) \in A$.

For example, the column of points $(a, 0), (a, 1), (a, 2), \ldots, (a, f(a))$ is constructed by starting with the basis point $(a, 0)$ and by repeatedly using the second if-then statement. The first if-then statement constructs the points on the x-axis that are then used to construct the other columns of points. Notice with this definition that each pair is constructed exactly once.

end example

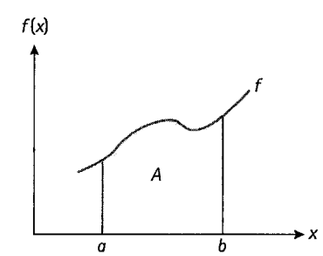


Figure 3.5 Area under a curve.

Exercises

Numbers

1. For each of the following inductive definitions, start with the basis element and construct ten elements in the set.

   a. *Basis:* $3 \in S$.

      Induction: If $x \in S$, then $2x - 1 \in S$.

   b. *Basis:* $1 \in S$.

      Induction: If $x \in S$, then $2x, 2x + 1 \in S$.

2. Find an inductive definition for each set S.

   a. $\{1, 3, 5, 7, \ldots\}$.
   b. $\{0, 2, 4, 6, 8, \ldots\}$.
   c. $\{-3, -1, 1, 3, 5, \ldots\}$.
   d. $\{\ldots, -7, -4, -1, 2, 5, 8, \ldots\}$.
   e. $\{1, 4, 9, 16, 25, \ldots\}$.
   f. $\{1, 3, 7, 15, 31, 63, \ldots\}$.

3. Find an inductive definition for each set S.

   a. $\{4, 7, 10, 13, \ldots\} \cup \{3, 6, 9, 12, \ldots\}$.
   b. $\{3, 4, 5, 8, 9, 12, 16, 17, \ldots\}$. *Hint:* Write the set as a union.

4. Find an inductive definition for each set S.

   a. $\{x \in \mathbb{N} \mid \text{floor}(x/2) \text{ is even}\}$.
   b. $\{x \in \mathbb{N} \mid \text{floor}(x/2) \text{ is odd}\}$.
   c. $\{x \in \mathbb{N} \mid x \bmod 5 = 2\}$.
   d. $\{x \in \mathbb{N} \mid 2x \bmod 7 = 3\}$.

5. The following inductive definition was given in Example 3.4, the second robot example.

   Basis: $\varnothing \in Nat$.

   Induction: If $s \in Nat$, then $s \cup \{s\} \in Nat$.

   In Example 3.4 we identified natural numbers with the elements of Nat by setting $0 = \varnothing$ and $n = n \cup \{n\}$ for $n \neq 0$. Show that $4 = \{0, 1, 2, 3\}$.

Strings

6. Find an inductive definition for each set S of strings.

6. FIND A INDUCTIVE DEFINITION FOR EACH SET S OF STRINGS

   a. $\{a^n bc^n \mid n \in \mathbb{N}\}$.
   b. $\{a^{2n} \mid n \in \mathbb{N}\}$.
   c. $\{a^{2n+1} \mid n \in \mathbb{N}\}$.
   d. $\{a^m b^n \mid m, n \in \mathbb{N}\}$.
   e. $\{a^m bc^n \mid m, n \in \mathbb{N}\}$.
   f. $\{a^m b^n \mid m, n \in \mathbb{N}, \text{ where } m > 0\}$.
   g. $\{a^m b^n \mid m, n \in \mathbb{N}, \text{ where } n > 0\}$.
   h. $\{a^m b^n \mid m, n \in \mathbb{N}, \text{ where } m > 0 \text{ and } n > 0\}$.
   i. $\{a^m b^n \mid m, n \in \mathbb{N}, \text{ where } m > 0 \text{ or } n > 0\}$.
   j. $\{a^{2n} \mid n \in \mathbb{N}\} \cup \{b^{2n+1} \mid n \in \mathbb{N}\}$.
   k. $\{s \in \{a, b\}^* \mid s \text{ has the same number of } a\text{'s and } b\text{'s}\}$.

7. Find an inductive definition for each set S of strings. Note that a *palindrome* is a string that is the same when written in reverse order.

   a. Even palindromes over the set $\{a, b\}$.
   b. Odd palindromes over the set $\{a, b\}$.
   c. All palindromes over the set $\{a, b\}$.
   d. The binary numerals.

8. Let the letters a, b, and c be constants; let the letters x, y, and z be variables; and let the letters f and g be functions of arity 1. We can define the set of terms over these symbols by saying that any constant or variable is a term and if t is a term, then so are $f(t)$ and $g(t)$. Find an inductive definition for the set T of terms.

Lists

9. For each of the following inductive definitions, start with the basis element and construct five elements in the set.

   a. *Basis:* $\langle a \rangle \in S$.

      Induction: If $x \in S$, then $b :: x \in S$.

   b. *Basis:* $\langle 1 \rangle \in S$.

      Induction: If $x \in S$, then $2 \cdot \text{head}(x) :: x \in S$.

10. Find an inductive definition for each set S of lists. Use the cons constructor.

    a. $\{\langle a \rangle, \langle a, a \rangle, \langle a, a, a \rangle, \ldots\}$.
    b. $\{\langle 1 \rangle, \langle 2, 1 \rangle, \langle 3, 2, 1 \rangle, \ldots\}$.
    c. $\{\langle a, b \rangle, \langle b, a \rangle, \langle a, a, b \rangle, \langle b, b, a \rangle, \langle a, a, a, b \rangle, \langle b, b, b, a \rangle, \ldots\}$.
    d. $\{L \mid L \text{ has even length over } \{a\}\}$.
    e. $\{L \mid L \text{ has even length over } \{0, 1, 2\}\}$.
    f. $\{L \mid L \text{ has even length over a set } A\}$.

g. $\{L \mid L$ has odd length over $\{a\}\}$.
h. $\{L \mid L$ has odd length over $\{0, 1, 2\}\}$.
i. $\{L \mid L$ has odd length over a set $A\}$.

11. Find an inductive definition for each set S of lists. You may use the "consR" operation, where consR(L, x) is the list constructed from the list L by adding a new element x on the right end. Similarly, you may use the "headR" and "tailR" operations, which are like head and tail but look at things from the right side of a list.
    a. $\{\langle a\rangle, \langle a, b\rangle, \langle a, b, b\rangle, \ldots\}$.
    b. $\{\langle 1\rangle, \langle 1, 2\rangle, \langle 1, 2, 3\rangle, \ldots\}$.
    c. $\{ L \in \text{lists}(\{a, b\}) \mid L$ has the same number of a's and b's$\}$.

12. Find an inductive definition for the set S of all lists over $A = \{a, b\}$ that alternate a's and b's. For example, the lists $\langle\ \rangle$, $\langle a\rangle$, $\langle b\rangle$, $\langle a, b, a\rangle$, and $\langle b, a\rangle$ are in S. But $\langle a, a\rangle$ is not in S.

Binary Trees

13. Given the following inductive definition for a set S of binary trees. Start with the basis element and draw pictures of four binary trees in the set. Don't draw the empty subtrees.

    Basis: tree$(\langle\ \rangle, a, \langle\ \rangle) \in S$.
    Induction: If $T \in S$, then tree(tree$(\langle\ \rangle, a, \langle\ \rangle), a, T) \in S$.

14. Find an inductive definition for the set B of binary trees that represent arithmetic expressions that are either numbers in $\mathbb{N}$ or expressions that use operations $+$ or $-$.

15. Find an inductive definition for the set B of nonempty binary trees over $\{a\}$ in which each non-leaf node has two subtrees, one of which is a leaf and the other of which is either a leaf or a member of B.

Cartesian Products

16. Given the following inductive definition for a subset B of $\mathbb{N} \times \mathbb{N}$.

    Basis: $(0, 0) \in B$.
    Induction: If $(x, y) \in B$, then $(x + 1, y), (x + 1, y + 1) \in B$.

    a. Describe the set B as a set of the form $\{(x, y) \mid$ some property holds$\}$.
    b. Describe those elements in B that get defined in more than one way.

17. Find an inductive definition for each subset S of $\mathbb{N} \times \mathbb{N}$.
    a. $S = \{(x, y) \mid y = x$ or $y = x + 1\}$.
    b. $S = \{(x, y) \mid x$ is even and $y \leq x/2\}$.

18. Find an inductive definition for each product set S.
    a. $S = \text{lists}(A) \times \text{lists}(A)$ for some set A.
    b. $S = A \times \text{lists}(A)$.
    c. $S = \mathbb{N} \times \text{lists}(\mathbb{N})$.
    d. $S = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

Proofs and Challenges

19. Let A be a set. Suppose O is the set of binary trees over A that contains an odd number of nodes. Similarly, let E be the set of binary trees over A that contains an even number of nodes. Find inductive definitions for O and E. *Hint:* You can use O when defining E, and you can use E when defining O.

20. Use Example 3.15 as a guide to construct an inductive definition for the set of points in $\mathbb{N} \times \mathbb{N}$ that describe the area A between two curves f and g defined as follows for two natural numbers a and b:

$$A = \{(x, y) \mid x, y \in \mathbb{N},\ a \leq x \leq b, \text{ and } g(x) \leq y \leq f(x)\}.$$

21. Prove that a set defined by (3.1) is countable if the basis elements in Step 1 are countable, the outside elements, if any, used in Step 2 are countable, and the rules specified in Step 2 are finite.

3.2 Recursive Functions and Procedures

Since we're going to be constructing functions and procedures in this section, we'd better agree on the idea of a procedure. From a computer science point of view a *procedure* is a program that performs one or more actions. So there is no requirement to return a specific value. For example, the execution of a statement like print(x, y) will cause the values of x and y to be printed. In this case, two actions are performed, and no values are returned. A procedure may also return one or more values through its argument list. For example, a statement like allocate(m, a, s) might perform the action of allocating a block of m memory cells and return the values a and s, where a is the beginning address of the block and the s tells whether the allocation was successful.

Definition of Recursively Defined

A function or a procedure is said to be *recursively defined* if it is defined in terms of itself. In other words, a function f is recursively defined if at least one value $f(x)$ is defined in terms of another value $f(y)$, where $x \neq y$. Similarly, a procedure P is recursively defined if the actions of P for some argument x are defined in terms of the actions of P for another argument y, where $x \neq y$.

Many useful recursively defined functions have domains that are inductively defined sets. Similarly, many recursively defined procedures process elements from inductively defined sets. For these cases there are very useful construction techniques. Let's describe the two techniques.

Constructing a Recursively Defined Function **(3.6)**

If S is an inductively defined set, then we can construct a function f with domain S as follows:

1. For each basis element $x \in S$, specify a value for $f(x)$.
2. Give rules that, for any inductively defined element $x \in S$, will define $f(x)$ in terms of previously defined values of f.

Any function constructed by (3.6) is recursively defined because it is defined in terms of itself by the induction part of the definition. In a similar way we can construct a recursively defined procedure to process the elements of an inductively defined set.

Constructing a Recursively Defined Procedure **(3.7)**

If S is an inductively defined set, we can construct a procedure P to process the elements of S as follows:

1. For each basis element $x \in S$, specify a set of actions for $P(x)$.
2. Give rules that, for any inductively defined element $x \in S$, will define the actions of $P(x)$ in terms of previously defined actions of P.

In the following paragraphs we'll see how (3.6) and (3.7) can be used to construct recursively defined functions and procedures over a variety of inductively defined sets. Most of our examples will be functions. But we'll define a few procedures too.

3.2.1 Numbers

Let's see how some number functions can be defined recursively. To illustrate the idea, suppose we want to calculate the sum of the first n natural numbers for any $n \in \mathbb{N}$. Letting $f(n)$ denote the desired sum, we can write the informal definition

$$f(n) = 0 + 1 + 2 + \cdots + n.$$

We can observe, for example, that $f(0) = 0$, $f(1) = 1$, $f(2) = 3$, and so on. After a while we might notice that $f(3) = f(2) + 3 = 6$ and $f(4) = f(3) + 4 = 10$.

In other words, when $n > 0$, the definition can be transformed in the following way:

$$\begin{aligned} f(n) &= 0 + 1 + 2 + \cdots + n \\ &= (0 + 1 + 2 + \cdots + (n-1)) + n \\ &= f(n-1) + n. \end{aligned}$$

This gives us the recursive part of a definition of f for any $n > 0$. For the basis case we have $f(0) = 0$. So we can write the following recursive definition for f:

$$\begin{aligned} f(0) &= 0, \\ f(n) &= (n-1) + n \quad \text{for } n > 0. \end{aligned}$$

There are two alternative forms that can be used to write a recursive definition. One form expresses the definition as an *if-then-else* equation. For example, f can be described in the following way.

$$f(n) = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1) + n.$$

Another form expresses the definition as equations whose left sides determine which equation to use in the evaluation of an expression rather than a conditional like $n > 0$. Such a form is called a *pattern-matching* definition because the equation chosen to evaluate $f(x)$ is determined uniquely by which left side $f(x)$ matches. For example, f can be described in the following way.

$$\begin{aligned} f(0) &= 0, \\ f(n+1) &= f(n) + n + 1. \end{aligned}$$

For example, $f(3)$ matches $f(n+1)$ with $n = 2$, so we would choose the second equation to evaluate $f(3) = f(2) + 3$, and so on.

A recursively defined function can be evaluated by a technique called *unfolding* the definition. For example, we'll evaluate the expression $f(4)$.

$$\begin{aligned} f(4) &= f(3) + 4 \\ &= f(2) + 3 + 4 \\ &= f(1) + 2 + 3 + 4 \\ &= f(0) + 1 + 2 + 3 + 4 \\ &= 0 + 1 + 2 + 3 + 4 \\ &= 10. \end{aligned}$$

example 3.16 Using the Floor Function

Let $f : \mathbb{N} \to \mathbb{N}$ be defined in terms of the floor function as follows:

$$f(0) = 0,$$
$$f(n) = f(\text{floor}(n/2)) + n \quad \text{for } n > 0.$$

Notice in this case that $f(n)$ is not defined in terms of $f(n-1)$ but rather in terms of $f(\text{floor}(n/2))$. For example, $f(16) = f(8) + 16$. The first few values are $f(0) = 0$, $f(1) = 1$, $f(2) = 3$, $f(3) = 4$, and $f(4) = 7$. We'll calculate $f(25)$.

$$\begin{aligned} f(25) &= f(12) + 25 \\ &= f(6) + 12 + 25 \\ &= f(3) + 6 + 12 + 25 \\ &= f(1) + 3 + 6 + 12 + 25 \\ &= f(0) + 1 + 3 + 6 + 12 + 25 \\ &= 0 + 1 + 3 + 6 + 12 + 25 \\ &= 47. \end{aligned}$$

end example

example 3.17 Adding Odd Numbers

Let $f : \mathbb{N} \to \mathbb{N}$ denote the function to add up the first n odd natural numbers. So f has the following informal definition.

$$f(n) = 1 + 3 + \cdots + (2n + 1).$$

For example, the definition tells us that $f(0) = 1$. For $n > 0$ we can make the following transformation of $f(n)$ into an expression in terms of $f(n-1)$:

$$\begin{aligned} f(n) &= 1 + 3 + \cdots + (2n + 1) \\ &= (1 + 3 + \cdots + (2(n-1) + 1)) + (2n + 1) \\ &= f(n-1) + 2n + 1. \end{aligned}$$

So we can make the following recursive definition of f:

$$f(0) = 1,$$
$$f(n) = f(n-1) + 2n + 1 \quad \text{for } n > 0.$$

Alternatively, we can write the recursive part of the definition as

$$f(n + 1) = f(n) + 2n + 3.$$

We can also write the definition in the following if-then-else form.

$$f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } f(n-1) + 2n + 1.$$

Here is the evaluation of $f(3)$ using the if-then-else definition:

$$\begin{aligned} f(3) &= f(2) + 2(3) + 1 \\ &= f(1) + 2(2) + 1 + 2(3) + 1 \\ &= f(0) + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\ &= 1 + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\ &= 1 + 3 + 5 + 7 \\ &= 16. \end{aligned}$$

end example

example 3.18 The Rabbit Problem

The *Fibonacci numbers* are the numbers in the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

where each number after the first two is computed by adding the preceding two numbers. These numbers are named after the mathematician Leonardo Fibonacci, who in 1202 introduced them in his book *Liber Abaci*, in which he proposed and solved the following problem: Starting with a pair of rabbits, how many pairs of rabbits can be produced from that pair in a year if it is assumed that every month each pair produces a new pair that becomes productive after one month?

For example, if we don't count the original pair and assume that the original pair needs one month to mature and that no rabbits die, then the number of new pairs produced each month for 12 consecutive months is given by the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.$$

The sum of these numbers, which is 232, is the number of pairs of rabbits produced in one year from the original pair.

Fibonacci numbers seem to occur naturally in many unrelated problems. Of course, they can also be defined recursively. For example, letting $\text{fib}(n)$ be the nth Fibonacci number, we can define fib recursively as follows:

$$\text{fib}(0) = 0,$$
$$\text{fib}(1) = 1,$$
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \quad \text{for } n \geq 2.$$

The third line could be written in pattern-matching form as

$$\text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n).$$

The definition of fib in if-then-else form looks like

$$\begin{aligned}\text{fib}(n) = {} & \text{if } n = 0 \text{ then } 0 \\ & \text{else if } n = 1 \text{ then } 1 \\ & \text{else fib}(n-1) + \text{fib}(n-2).\end{aligned}$$

end example

3.2.2 Strings

Let's see how some string functions can be defined recursively. To illustrate the idea, suppose we want to calculate the complement of any string over the alphabet $\{a, b\}$. For example, the complement of the string *bbab* is *aaba*.

Let $f(x)$ be the complement of x. To find a recursive definition for f we'll start by observing that an arbitrary string over $\{a, b\}$ is either Λ or has the form ay or by for some string y. So we'll define the result of f applied to each of these forms as follows:

$$\begin{aligned}f(\Lambda) &= \Lambda, \\ f(ax) &= bf(x), \\ f(bx) &= af(x).\end{aligned}$$

For example, we'll evaluate $f(bbab)$:

$$\begin{aligned}f(bbab) &= af(bab) \\ &= aaf(ab) \\ &= aabf(b) \\ &= aabaf(\Lambda) \\ &= abba\Lambda \\ &= aaba.\end{aligned}$$

Here are some more examples.

example 3.19 Prefixes of Strings

Consider the problem of finding the longest common prefix of two strings. A string p is a *prefix* of the string x if x can be written in the form $x = ps$ for some string s. For example, *aab* is the longest common prefix of the two strings *aabbab* and *aababb*.

For two strings x and y over $\{a, b\}$, let $f(x, y)$ be the longest common prefix of x and y. To find a recursive definition for f we can start by observing that an arbitrary string over $\{a, b\}$ is either the empty string Λ or has the form as or

bs for some string s. In other words, the strings over $\{a, b\}$ are an inductively defined set. Here is a definition of f in pattern-matching form:

$$\begin{aligned}f(\Lambda, x) &= \Lambda, \\ f(x, \Lambda) &= \Lambda, \\ f(as, bt) &= \Lambda, \\ f(bs, at) &= \Lambda, \\ f(as, at) &= af(s, t), \\ f(bs, bt) &= bf(s, t).\end{aligned}$$

We can put the definition in if-then-else form as follows:

$$\begin{aligned}f(x, y) = {} & \text{if } x = \Lambda \text{ or } y = \Lambda \text{ then } \Lambda \\ & \text{else if } x = as \text{ and } y = at \text{ then } af(s, t) \\ & \text{else if } x = bs \text{ and } y = bt \text{ then } bf(s, t) \\ & \text{else } \Lambda.\end{aligned}$$

We'll demonstrate the definition of f by calculating $f(aabbab, aababb)$:

$$\begin{aligned}f(aabbab, aababb) &= af(abbab, ababb) \\ &= aaf(bbab, babb) \\ &= aabf(bab, abb) \\ &= aab\Lambda \\ &= aab.\end{aligned}$$

end example

example 3.20 Converting Natural Numbers to Binary

Recall from Section 2.1 that we can represent a natural number x as

$$x = 2(\text{floor}(x/2)) + x \bmod 2.$$

This formula can be used to create a binary representation of x because $x \bmod 2$ is the rightmost bit of the representation. The next bit is found by computing $\text{floor}(x/2) \bmod 2$. The next bit is $\text{floor}(\text{floor}(x/2)/2) \bmod 2$, and so on. For example, we'll compute the binary representation of 13.

$$\begin{aligned}13 &= 2\,\text{floor}(13/2) + 13 \bmod 2 &&= 2(6) + 1 \\ 6 &= 2\,\text{floor}(6/2) + 6 \bmod 2 &&= 2(3) + 0 \\ 3 &= 2\,\text{floor}(3/2) + 3 \bmod 2 &&= 2(1) + 1 \\ 1 &= 2\,\text{floor}(1/2) + 1 \bmod 2 &&= 2(0) + 1\end{aligned}$$

We can read off the remainders in reverse order to obtain 1101, which is the binary representation of 13.

Let's try to use this idea to write a recursive definition for the function "binary" to compute the binary representation for a natural number. If $x = 0$ or $x = 1$, then x is its own binary representation. If $x > 1$, then the binary representation of x is that of floor$(x/2)$ with the bit x mod 2 attached on the right end. So our recursive definition of binary can be written as follows, where "cat" is the string concatenation function.

$$\begin{aligned} &\text{binary}(0) = 0, \\ &\text{binary}(1) = 1, \\ &\text{binary}(x) = \text{cat}(\text{binary}(\text{floor}(x/2)), x \bmod 2) \quad \text{for } x > 1. \end{aligned} \tag{3.8}$$

The definition can be written in if-then-else form as

$$\begin{aligned} \text{binary}(x) = {} & \text{if } x = 0 \text{ or } x = 1 \text{ then } x \\ & \text{else cat}(\text{binary}(\text{floor}(x/2)), x \bmod 2). \end{aligned}$$

For example, we'll unfold the definition to calculate binary(13):

$$\begin{aligned} \text{binary}(13) &= \text{cat}(\text{binary}(6), 1) \\ &= \text{cat}(\text{cat}(\text{binary}(3), 0), 1) \\ &= \text{cat}(\text{cat}(\text{cat}(\text{binary}(1), 1), 0), 1) \\ &= \text{cat}(\text{cat}(\text{cat}(1, 1), 0), 1) \\ &= \text{cat}(\text{cat}(11, 0), 1) \\ &= \text{cat}(110, 1) \\ &= 1101. \end{aligned}$$

end example

3.2.3 Lists

Let's see how some functions that use lists can be defined recursively. To illustrate the idea, suppose we need to define the function $f : \mathbb{N} \to \text{lists}(\mathbb{N})$ that computes the following backward sequence:

$$f(n) = \langle n, n-1, \ldots, 1, 0 \rangle.$$

With a little help from the cons function for lists, we can transform the informal definition of $f(n)$ into a computable expression in terms of $f(n-1)$:

$$\begin{aligned} f(n) &= \langle n, n-1, \ldots, 1, 0 \rangle \\ &= \text{cons}(n, \langle n-1, \ldots, 1, 0 \rangle) \\ &= \text{cons}(n, f(n-1)). \end{aligned}$$

Therefore, f can be defined recursively by

$$\begin{aligned} &f(0) = \langle 0 \rangle, \\ &f(n) = \text{cons}(n, f(n-1)) \text{ for } n > 0. \end{aligned}$$

This definition can be written in if-then-else form as

$$f(n) = \text{if } n = 0 \text{ then } \langle 0 \rangle \text{ else cons}(n, f(n-1)).$$

To see how the evaluation works, look at the unfolding that results when we evaluate $f(3)$:

$$\begin{aligned} f(3) &= \text{cons}(3, f(2)) \\ &= \text{cons}(3, \text{cons}(2, f(1))) \\ &= \text{cons}(3, \text{cons}(2, \text{cons}(1, f(0)))) \\ &= \text{cons}(3, \text{cons}(2, \text{cons}(1, \langle 0 \rangle))) \\ &= \text{cons}(3, \text{cons}(2, \langle 1, 0 \rangle)) \\ &= \text{cons}(3, \langle 2, 1, 0 \rangle) \\ &= \langle 3, 2, 1, 0 \rangle. \end{aligned}$$

We haven't given a recursively defined procedure yet. So let's give one for the problem we've been discussing. For example, suppose that $P(n)$ prints out the numbers in the list $\langle n, n-1, \ldots, 0 \rangle$. A recursive definition of P can be written as follows.

```
P(n):  if n = 0 then print(0)
       else
          print(n);
          P(n - 1)
       fi.
```

example **3.21 Length of a List**

Let S be a set and let "length" be the function of type lists$(S) \to \mathbb{N}$, which returns the number of elements in a list. We can define length recursively by noticing that the length of an empty list is zero and the length of a nonempty list is one plus the length of its tail. A definition follows.

$$\begin{aligned} \text{length}(\langle\ \rangle) &= 0, \\ \text{length}(\text{cons}(x, t)) &= 1 + \text{length}(t). \end{aligned}$$

Recall that the infix form of cons(x, t) is $x :: t$. So we could just as well write the second equation as

$$\text{length}(x :: t) = 1 + \text{length}(t).$$

Also, we could write the recursive part of the definition with a condition as follows:

$$\text{length}\,(L) = 1 + \text{length}\,(\text{tail}\,(L)) \quad \text{for } L \neq \langle\,\rangle\,.$$

In if-then-else form the definition can be written as follows:

$$\text{length}(L) = \text{if } L = \langle\,\rangle \text{ then } 0 \text{ else } 1 + \text{length}(\text{tail}(L)).$$

The length function can be evaluated by unfolding its definition. For example, we'll evaluate length($\langle a, b, c\rangle$).

$$\begin{aligned}\text{length}\,(\langle a, b, c\rangle) &= 1 + \text{length}\,(\langle b, c\rangle)\\ &= 1 + 1 + \text{length}\,(\langle c\rangle)\\ &= 1 + 1 + 1 + \text{length}\,(\langle\,\rangle)\\ &= 1 + 1 + 1 + 0\\ &= 3.\end{aligned}$$

end example

example 3.22 The Distribute Function

Suppose we want to write a recursive definition for the distribute function, which we'll denote by "dist." Recall, for example, that

$$\text{dist}(a,\ \langle b,\ c,\ d,\ e\rangle) = \langle(a,\ b),\ (a,\ c),\ (a,\ d),\ (a,\ e)\rangle.$$

To discover the recursive part of the definition, we'll rewrite the example equation by splitting up the lists into head and tail components as follows:

$$\begin{aligned}\text{dist}\,(a, \langle b, c, d, e\rangle) &= \langle(a, b), (a, c), (a, d), (a, e)\rangle\\ &= (a, b) :: \langle(a, c), (a, d), (a, e)\rangle\\ &= (a, b) :: \text{dist}(a, \langle c, d, e\rangle).\end{aligned}$$

That's the key to the recursive part of the definition. Since we are working with lists, the basis case is dist(a, $\langle\,\rangle$), which we define as $\langle\,\rangle$. So the recursive definition can be written as follows:

$$\begin{aligned}\text{dist}(x, \langle\,\rangle) &= \langle\,\rangle,\\ \text{dist}(x, h :: T) &= (x, h) :: \text{dist}\,(x, T)\,.\end{aligned}$$

For example, we'll evaluate the expression dist(3, $\langle 10,\ 20\rangle$):

$$\begin{aligned}\text{dist}\,(3, \langle 10, 20\rangle) &= (3, 10) :: \text{dist}\,(3, \langle 20\rangle)\\ &= (3, 10) :: (3, 20) :: \text{dist}\,(3, \langle\,\rangle)\\ &= (3, 10) :: (3, 20) :: \langle\,\rangle\\ &= (3, 10) :: \langle(3, 20)\rangle\\ &= \langle(3, 10), (3, 20)\rangle\,.\end{aligned}$$

An if-then-else definition of dist takes the following form:

$$\begin{aligned}\text{dist}(x, L) = {}&\text{if } L = \langle\,\rangle \text{ then } \langle\,\rangle\\ &\text{else } (x, \text{head}(L)) :: \text{dist}(x, \text{tail}(L)).\end{aligned}$$

end example

example 3.23 The Pairs Function

Recall that the "pairs" function creates a list of pairs of corresponding elements from two lists. For example,

$$\text{pairs}(\langle a,\ b,\ c\rangle,\ \langle 1,\ 2,\ 3\rangle) = \langle(a,\ 1),\ (b,\ 2),\ (c,\ 3)\rangle.$$

To discover the recursive part of the definition, we'll rewrite the example equation by splitting up the lists into head and tail components as follows:

$$\begin{aligned}\text{pairs}(\langle a, b, c\rangle, \langle 1, 2, 3\rangle) &= \langle(a, 1), (b, 2), (c, 3)\rangle\\ &= (a, 1) :: \langle(b, 2), (c, 3)\rangle\\ &= (a, 1) :: \text{pairs}(\langle b, c\rangle, \langle 2, 3\rangle).\end{aligned}$$

Now the pairs function can be defined recursively by the following equations:

$$\begin{aligned}\text{pairs}\,(\langle\,\rangle, \langle\,\rangle) &= \langle\,\rangle\,,\\ \text{pairs}\,(x :: T, y :: U) &= (x, y) :: \text{pairs}(T, U).\end{aligned}$$

For example, we'll evaluate the expression pairs($\langle a,\ b\rangle$, $\langle 1,\ 2\rangle$):

$$\begin{aligned}\text{pairs}\,(\langle a, b\rangle, \langle 1, 2\rangle) &= (a, 1) :: \text{pairs}\,(\langle b\rangle, \langle 2\rangle)\\ &= (a, 1) :: (b, 2) :: \text{pairs}\,(\langle\,\rangle, \langle\,\rangle)\\ &= (a, 1) :: (b, 2) :: \langle\,\rangle\\ &= (a, 1) :: \langle(b, 2)\rangle\\ &= \langle(a, 1), (b, 2)\rangle\,.\end{aligned}$$

end example

example 3.24 The ConsRight Function

Suppose we need to give a recursive definition for the sequence function. Recall, for example, that seq(4) = $\langle 0,\ 1,\ 2,\ 3,\ 4\rangle$. Good old "cons" doesn't seem up to the task. For example, if we somehow have computed seq(3), then cons(4, seq(3)) = $\langle 4,\ 0,\ 1,\ 2,\ 3\rangle$. It would be nice if we had a constructor to place an element on the right of a list, just as cons places an element on the left

of a list. We'll write a definition for the function "consR" to do just that. For example, we want

$$\text{consR}(\langle a,\ b,\ c\rangle,\ d) = \langle a,\ b,\ c,\ d\rangle.$$

We can get an idea of how to proceed by rewriting the previous equation as follows in terms of the infix form of cons:

$$\begin{aligned}\text{consR}\left(\langle a,b,c\rangle,d\right) &= \langle a,b,c,d\rangle\\ &= a :: \langle b,c,d\rangle\\ &= a :: \text{consR}\left(\langle b,c\rangle,d\right).\end{aligned}$$

So the clue is to split the list $\langle a,\ b,\ c\rangle$ into its head and tail. We can write the recursive definition of consR using the if-then-else form as follows:

$$\begin{aligned}\text{consR}\left(L,a\right) = {}&\text{if } L = \langle\ \rangle \text{ then } \langle a\rangle\\ &\text{else head}\left(L\right) :: \text{consR}\left(\text{tail}\left(L\right),a\right).\end{aligned}$$

This definition can be written in pattern-matching form as follows:

$$\begin{aligned}\text{consR}\left(\langle\ \rangle,a\right) &= a :: \langle\ \rangle,\\ \text{consR}\left(b :: T,a\right) &= b :: \text{consR}\left(T,a\right).\end{aligned}$$

For example, we can construct the list $\langle x,\ y\rangle$ with consR as follows:

$$\begin{aligned}\text{consR}\left(\text{consR}\left(\langle\ \rangle,x\right),y\right) &= \text{consR}\left(x :: \langle\ \rangle,y\right)\\ &= x :: \text{consR}\left(\langle\ \rangle,y\right)\\ &= x :: y :: \langle\ \rangle\\ &= x :: \langle y\rangle\\ &= \langle x,y\rangle.\end{aligned}$$

end example

example 3.25 Concatenation of Lists

An important operation on lists is the concatenation of two lists into a single list. Let "cat" denote the concatenation function. Its type is lists(A) $\times$ lists(A) $\rightarrow$ lists(A). For example,

$$\text{cat}(\langle a,\ b\rangle,\ \langle c,\ d\rangle) = \langle a,\ b,\ c,\ d\rangle.$$

Since both arguments are lists, we have some choices to make. Notice, for example, that we can rewrite the equation as follows:

$$\begin{aligned}\text{cat}(\langle a,b\rangle,\langle c,d\rangle) &= \langle a,b,c,d\rangle\\ &= a :: \langle b,c,d\rangle\\ &= a :: \text{cat}(\langle b\rangle,\langle c,d\rangle).\end{aligned}$$

So the recursive part can be written in terms of the head and tail of the first argument list. Here's an if-then-else definition for cat.

$$\begin{aligned}\text{cat}(L,M) = {}&\text{if } L = \langle\ \rangle \text{ then } M\\ &\text{else head}(L) :: \text{cat}(\text{tail}(L),M).\end{aligned}$$

We'll unfold the definition to evaluate the expression cat($\langle a,\ b\rangle$, $\langle c,\ d\rangle$):

$$\begin{aligned}\text{cat}\left(\langle a,b\rangle,\langle c,d\rangle\right) &= a :: \text{cat}\left(\langle b\rangle,\langle c,d\rangle\right)\\ &= a :: b :: \text{cat}\left(\langle\ \rangle,\langle c,d\rangle\right)\\ &= a :: b :: \langle c,d\rangle\\ &= a :: \langle b,c,d\rangle\\ &= \langle a,b,c,d\rangle.\end{aligned}$$

end example

example 3.26 Sorting a List by Insertion

Let's define a function to sort a list of numbers by repeatedly inserting a new number into an already sorted list of numbers. Suppose "insert" is a function that does this job. Then the sort function itself is easy. For a basis case, notice that the empty list is already sorted. For the recursive case we sort the list $x :: L$ by inserting x into the list obtained by sorting L. The definition can be written as follows:

$$\begin{aligned}\text{sort}\left(\langle\ \rangle\right) &= \langle\ \rangle,\\ \text{sort}\left(x :: L\right) &= \text{insert}\left(x,\text{sort}\left(L\right)\right).\end{aligned}$$

Everything seems to make sense as long as insert does its job. We'll assume that whenever the number to be inserted is already in the list, then a new copy will be placed to the left of the one already there. Now let's define insert. Again, the basis case is easy. The empty list is sorted, and to insert x into $\langle\ \rangle$, we simply create the singleton list $\langle x\rangle$. Otherwise—if the sorted list is not empty—either x belongs on the left of the list, or it should actually be inserted somewhere else in the list. An if-then-else definition can be written as follows:

$$\begin{aligned}\text{insert}\left(x,S\right) = {}&\text{if } S = \langle\ \rangle \text{ then } \langle x\rangle\\ &\text{else if } x \leq \text{ head}\left(S\right) \text{ then } x :: S\\ &\text{else head}\left(S\right) :: \text{insert}\left(x,\text{tail}\left(S\right)\right).\end{aligned}$$

Notice that insert works only when S is already sorted. For example, we'll unfold the definition of insert(3, ⟨1, 2, 6, 8⟩):

$$\begin{aligned}\text{insert}\,(3, \langle 1,2,6,8\rangle) &= 1 :: \text{insert}\,(3, \langle 2,6,8\rangle)\\ &= 1 :: 2 :: \text{insert}\,(3, \langle 6,8\rangle)\\ &= 1 :: 2 :: 3 :: \langle 6,8\rangle\\ &= \langle 1,2,3,6,8\rangle\,.\end{aligned}$$

end example

example 3.27 The Map Function

We'll construct a recursive definition for the map function. Recall, for example that

$$\text{map}(f, \langle a,b,c\rangle) = \langle f(a), f(b), f(c)\rangle.$$

Since the second argument is a list, it makes sense to define the basis case as $\text{map}(f, \langle\,\rangle) = \langle\,\rangle$. To discover the recursive part of the definition, we'll rewrite the example equation as follows:

$$\begin{aligned}\text{map}(f, \langle a,b,c\rangle) &= \langle f(a), f(b), f(c)\rangle\\ &= f(a) :: \langle f(b), f(c)\rangle\\ &= f(a) :: \text{map}(f, \langle b,c\rangle).\end{aligned}$$

So the recursive part can be written in terms of the head and tail of the input list. Here's an if-then-else definition for map.

$$\begin{aligned}\text{map}(f, L) = {}&\text{if } L = \langle\,\rangle \text{ then } \langle\,\rangle\\ &\text{else } f(\text{head}(L)) :: \text{map}(f, \text{tail}(L)).\end{aligned}$$

For example, we'll evaluate the expression map(f, ⟨a, b, c⟩).

$$\begin{aligned}\text{map}\,(f, \langle a,b,c\rangle) &= f(a) :: \text{map}\,(f, \langle b,c\rangle)\\ &= f(a) :: f(b) :: \text{map}\,(f, \langle c\rangle)\\ &= f(a) :: f(b) :: f(c) :: \text{map}\,(f, \langle\,\rangle)\\ &= f(a) :: f(b) :: f(c) :: \langle\,\rangle\\ &= \langle f(a)\,, f(b)\,, f(c)\rangle\,.\end{aligned}$$

end example

3.2.4 Binary Trees

Let's look at some functions that compute properties of binary trees. To start, suppose we need to know the number of nodes in a binary tree. Since the set of binary trees over a particular set can be defined inductively, we should be able to come up with a recursively defined function that suits our needs. Let "nodes" be the function that returns the number of nodes in a binary tree. Since the empty tree has no nodes, we have nodes(⟨ ⟩) = 0. If the tree is not empty, then the number of nodes can be computed by adding 1 to the number of nodes in the left and right subtrees. The definition of nodes can be written as follows:

$$\begin{aligned}\text{nodes}\,(\langle\,\rangle) &= 0,\\ \text{nodes}\,(\text{tree}\,(L, a, R)) &= 1 + \text{nodes}\,(L) + \text{nodes}\,(R)\,.\end{aligned}$$

If we want the corresponding if-then-else form of the definition, it looks like

$$\begin{aligned}\text{nodes}\,(T) = {}&\text{if } T = \langle\,\rangle \text{ then } 0\\ &\text{else } 1 + \text{nodes}\,(\text{left}\,(T)) + \text{nodes}\,(\text{right}\,(T))\,.\end{aligned}$$

For example, we'll evaluate nodes(T) for $T = \langle\langle\langle\,\rangle, a, \langle\,\rangle\rangle, b, \langle\,\rangle\rangle$:

$$\begin{aligned}\text{nodes}\,(T) &= 1 + \text{nodes}\,(\langle\langle\,\rangle\,, a, \langle\,\rangle\rangle) + \text{nodes}\,(\langle\,\rangle)\\ &= 1 + 1 + \text{nodes}\,(\langle\,\rangle) + \text{nodes}\,(\langle\,\rangle) + \text{nodes}\,(\langle\,\rangle)\\ &= 1 + 1 + 0 + 0 + 0\\ &= 2.\end{aligned}$$

example 3.28 A Binary Search Tree

Suppose we have a binary search tree whose nodes are numbers, and we want to add a new number to the tree, under the assumption that the new tree is still a binary search tree. A function to do the job needs two arguments, a number x and a binary search tree T. Let the name of the function be "insert."

The basis case is easy. If $T = \langle\,\rangle$, then return tree(⟨ ⟩, x, ⟨ ⟩). The recursive part is straightforward. If $x <$ root(T), then we need to replace the subtree left(T) by insert(x, left(T)). Otherwise, we replace right(T) by insert(x, right(T)). Notice that repeated elements are entered to the right. If we didn't want to add repeated elements, then we could simply return T whenever x = root(T). The if-then-else form of the definition is

$$\begin{aligned}\text{insert}\,(x, T) = {}&\text{if } T = \langle\,\rangle \text{ then tree}\,(\langle\,\rangle\,, x, \langle\,\rangle)\\ &\text{else if } x < \text{root}\,(T) \text{ then}\\ &\quad\text{tree}\,(\text{insert}\,(x, \text{left}\,(T))\,, \text{root}\,(T)\,, \text{right}\,(T))\\ &\text{else}\\ &\quad\text{tree}\,(\text{left}\,(T)\,, \text{root}\,(T)\,, \text{insert}\,(x, \text{right}\,(T)))\,.\end{aligned}$$

Now suppose we want to build a binary search tree from a given list of numbers in which the numbers are in no particular order. We can use the insert function as the main ingredient in a recursive definition. Let "makeTree" be the name of the function. We'll use two variables to describe the function, a binary search tree T and a list of numbers L.

$$\begin{aligned}\text{makeTree}\,(T, L) = {} & \text{if } L = \langle\,\rangle \text{ then } T \\ & \text{else makeTree}\,(\text{insert}\,(\text{head}\,(L)\,, T)\,, \text{tail}\,(L))\,.\end{aligned} \tag{3.9}$$

To construct a binary search tree with this function, we apply makeTree to the pair of arguments $(\langle\,\rangle, L)$. As an example, the reader should unfold the definition for makeTree$(\langle\,\rangle, \langle 3, 2, 4\rangle)$.

The function makeTree can be defined another way. Suppose we consider the following definition for constructing a binary search tree:

$$\begin{aligned}\text{makeTree}\,(T, L) = {} & \text{if } L = \langle\,\rangle \text{ then } T \\ & \text{else insert}\,(\text{head}\,(L)\,, \text{makeTree}\,(T, \text{tail}\,(L)))\,.\end{aligned} \tag{3.10}$$

You should evaluate the expression makeTree$(\langle\,\rangle, \langle 3, 2, 4\rangle)$ by unfolding this alternative definition. It should help explain the difference between the two definitions.

end example

Traversing Binary Trees

There are several useful ways to list the nodes of a binary tree. The three most popular methods of traversing a binary tree are called *preorder, inorder,* and *postorder*. We'll start with the definition of a preorder traversal.

> **Preorder Traversal**
>
> The *preorder* traversal of a binary tree starts by visiting the root. Then there is a preorder traversal of the left subtree followed by a preorder traversal of the right subtree.

For example, the preorder listing of the nodes of the binary tree in Figure 3.6 is $\langle a, b, c, d, e\rangle$. It's common practice to write the listing without any punctuation symbols as

$$a\ b\ c\ d\ e.$$

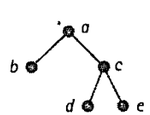


Figure 3.6 A binary tree.

example 3.29 A Preorder Procedure

Since binary trees are inductively defined, we can easily write a recursively defined procedure to output the preorder listing of a binary tree. For example, the following recursively defined procedure prints the preorder listing of its argument T.

```
Preorder(T):  if T ≠ ⟨⟩ then
                 print(root(T));
                 Preorder(left(T));
                 Preorder(right(T))
              fi.
```

end example

example 3.30 A Preorder Function

Now let's write a function to compute the preorder listing of a binary tree. Letting "preOrd" be the name of the preorder function, a definition can be written as follows:

$$\begin{aligned}\text{preOrd}\,(\langle\,\rangle) &= \langle\,\rangle, \\ \text{preOrd}\,(\text{tree}\,(L, x, R)) &= x :: \text{cat}\,(\text{preOrd}\,(L), \text{preOrd}\,(R))\,.\end{aligned}$$

The if-then-else form of preOrd can be written as follows:

$$\begin{aligned}\text{preOrd}\,(T) = {} & \text{if } T = \langle\,\rangle \text{ then } \langle\,\rangle \\ & \text{else root}\,(T) :: \text{cat}\,(\text{preOrd}\,(\text{left}\,(T)), \text{preOrd}\,(\text{right}\,(T)))\,.\end{aligned}$$

We'll evaluate the expression preOrd(T) for the tree $T = \langle\langle\langle\,\rangle, a, \langle\,\rangle\rangle, b, \langle\,\rangle\rangle$:

$$\begin{aligned}\text{preOrd}\,(T) &= b :: \text{cat}\,(\text{preOrd}\,(\langle\langle\,\rangle\,, a\,\langle\,\rangle\rangle), \text{preOrd}\,(\langle\,\rangle)) \\ &= b :: \text{cat}\,(a :: \text{cat}\,(\text{preOrd}\,(\langle\,\rangle), \text{preOrd}\,(\langle\,\rangle)), \text{preOrd}\,(\langle\,\rangle)) \\ &= b :: \text{cat}\,(a :: \langle\,\rangle, \langle\,\rangle) \\ &= b :: \text{cat}\,(\langle a\rangle, \langle\,\rangle) \\ &= b :: \langle a\rangle \\ &= \langle b, a\rangle\,.\end{aligned}$$

end example

The definitions for the inorder and postorder traversals of a binary tree are similar to the preorder traversal. The only difference is when the root is visited during the traversal.

Inorder Traversal

The *inorder* traversal of a binary tree starts with an inorder traversal of the left subtree. Then the root is visited. Lastly, there is an inorder traversal of the right subtree.

For example, the inorder listing of the tree in Figure 3.6 is

$$b\ a\ d\ c\ e.$$

Postorder Traversal

The *postorder* traversal of a binary tree starts with a postorder traversal of the left subtree and is followed by a postorder traversal of the right subtree. Lastly, the root is visited.

The postorder listing of the tree in Figure 3.6 is

$$b\ d\ e\ c\ a.$$

We'll leave the construction of the inorder and postorder procedures and functions as exercises.

3.2.5 Two More Problems

We'll look at two more problems, each of which requires a little extra thinking on the way to a solution.

The Repeated Element Problem

Suppose we want to remove repeated elements from a list. Depending on how we proceed, there might be different solutions. For example, we can remove the repeated elements from the list $\langle u, g, u, h, u\rangle$ in three ways, depending on which occurrence of u we keep: $\langle u, g, h\rangle$, $\langle g, u, h\rangle$, or $\langle g, h, u\rangle$. We'll solve the problem by always keeping the leftmost occurrence of each element. Let "remove" be the function that takes a list L and returns the list remove(L), which has no repeated elements and contains the leftmost occurrence of each element of L.

To start things off, we can say remove($\langle\ \rangle$) = $\langle\ \rangle$. Now if $L \neq \langle\ \rangle$, then L has the form $L = b :: M$ for some list M. In this case, the head of remove(L) should be b. The tail of remove(L) can be obtained by removing all occurrences of b from M and then removing all repeated elements from the resulting list. So we need a new function to remove all occurrences of an element from a list.

Let removeAll(b, M) denote the list obtained from M by removing all occurrences of b. Now we can write a definition for the remove function as follows:

$$\begin{aligned}&\text{remove}\,(\langle\ \rangle) = \langle\ \rangle,\\&\text{remove}\,(b :: M) = b :: \text{remove}\,(\text{removeAll}\,(b, M))\,.\end{aligned}$$

We can rewrite the solution in if-then-else form as follows:

$$\begin{aligned}\text{remove}\,(L) = {}&\text{if } L = \langle\ \rangle \text{ then } \langle\ \rangle\\&\quad\text{else head}\,(L) :: \text{remove}\,(\text{removeAll}\,(\text{head}\,(L), \text{tail}\,(L)))\,.\end{aligned}$$

To complete the task, we need to define the "removeAll" function. The basis case is removeAll(b, $\langle\ \rangle$) = $\langle\ \rangle$. If $M \neq \langle\ \rangle$, then the value of removeAll(b, M) depends on head(M). If head(M) = b, then throw it away and return the value of removeAll(b, tail(M)). But if head(M) $\neq b$, then it's a keeper. So we should return the value head(M) :: removeAll(b, tail(M)). We can write the definition in if-then-else form as follows:

$$\begin{aligned}\text{removeAll}\,(b, M) = {}&\text{if } M = \langle\ \rangle \text{ then } \langle\ \rangle\\&\quad\text{else if head}\,(M) = b \text{ then}\\&\qquad\text{removeAll}\,(b, \text{tail}\,(M))\\&\quad\text{else}\\&\qquad\text{head}\,(M) :: \text{removeAll}\,(b, \text{tail}\,(M))\,.\end{aligned}$$

We'll evaluate the expression removeAll(b, $\langle a, b, c, b\rangle$):

$$\begin{aligned}\text{removeAll}\,(b, \langle a, b, c, b\rangle) &= a :: \text{removeAll}\,(b, \langle b, c, b\rangle)\\&= a :: \text{removeAll}\,(b, \langle c, b\rangle)\\&= a :: c :: \text{removeAll}\,(b, \langle b\rangle)\\&= a :: c :: \text{removeAll}\,(b, \langle\ \rangle)\\&= a :: c :: \langle\ \rangle\\&= a :: \langle c\rangle\\&= \langle a, c\rangle\,.\end{aligned}$$

Try to write out each unfolding step in the evaluation of the expression remove($\langle b, a, b\rangle$). Be sure to start writing at the left-hand edge of your paper.

The Power Set Problem

Suppose we want to construct the power set of a finite set. One solution uses the fact that power($\{x\} \cup T$) is the union of power(T) and the set obtained from power(T) by adding x to each of its elements. Let's see whether we can discover

a solution technique by considering a small example. Let $S = \{a, b, c\}$. Then we can write power(S) as follows:

$$\begin{aligned}\text{power}(S) &= \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\ &= \{\{\}, \{b\}, \{c\}, \{b,c\}\} \cup \{\{a\}, \{a,b\}, \{a,c\}, \{a,b,c\}\}.\end{aligned}$$

We've written power(S) $= A \cup B$, where B is obtained from A by adding the element a to each set in A. If we represent S as the list $\langle a, b, c\rangle$, then we can restate the definition for power(S) as the concatenation of the following two lists:

$$\langle\langle\,\rangle, \langle b\rangle, \langle c\rangle, \langle b, c\rangle\rangle \quad \text{and} \quad \langle\langle a\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle a, b, c\rangle\rangle.$$

The first of these lists is power($\langle b, c\rangle$). The second list can be obtained from power($\langle b, c\rangle$) by working backward to the answer as follows:

$$\begin{aligned}\langle\langle a\rangle, \langle a,b\rangle, \langle a,c\rangle, \langle a,b,c\rangle\rangle &= \langle a :: \langle\,\rangle, a :: \langle b\rangle, a :: \langle c\rangle, a :: \langle b,c\rangle\rangle \\ &= \text{map}(::, \langle\langle a, \langle\,\rangle\rangle, \langle a, \langle b\rangle\rangle, \langle a, \langle c\rangle\rangle, \langle a, \langle b,c\rangle\rangle\rangle) \\ &= \text{map}(::, \text{dist}(a, \text{power}(\langle b,c\rangle))).\end{aligned}$$

This example is the key to the recursive part of the definition. Using the fact that power($\langle\,\rangle$) $= \langle\langle\,\rangle\rangle$ as the basis case, we can write down the following definition for power:

$$\begin{aligned}\text{power}(\langle\,\rangle) &= \langle\langle\,\rangle\rangle, \\ \text{power}(a :: T) &= \text{cat}(\text{power}(T), \text{map}(::, \text{dist}(a, \text{power}(T)))).\end{aligned}$$

The if-then-else form of the definition can be written as follows:

$$\begin{aligned}\text{power}(L) = {} & \text{if } L = \langle\,\rangle \text{ then } \langle\langle\,\rangle\rangle \text{ else} \\ & \text{cat}(\text{power}(\text{tail}(L)), \text{map}(::, \text{dist}(\text{head}(L), \text{power}(\text{tail}(L))))).\end{aligned}$$

We'll evaluate the expression power($\langle a, b\rangle$). The first step yields the equation

$$\text{power}(\langle a, b\rangle) = \text{cat}(\text{power}(\langle b\rangle), \text{map}(::, \text{dist}(a, \text{power}(\langle b\rangle)))).$$

Now we'll evaluate power($\langle b\rangle$) and substitute it in the preceding equation:

$$\begin{aligned}\text{power}(\langle b\rangle) &= \text{cat}(\text{power}(\langle\,\rangle), \text{map}(::, \text{dist}(b, \text{power}(\langle\,\rangle)))) \\ &= \text{cat}(\langle\langle\,\rangle\rangle, \text{map}(::, \text{dist}(b, \langle\langle\,\rangle\rangle))) \\ &= \text{cat}(\langle\langle\,\rangle\rangle, \text{map}(::, \langle\langle b, \langle\,\rangle\rangle\rangle)) \\ &= \text{cat}(\langle\langle\,\rangle\rangle, \langle b :: \langle\,\rangle\rangle) \\ &= \text{cat}(\langle\langle\,\rangle\rangle, \langle\langle b\rangle\rangle) \\ &= \langle\langle\,\rangle, \langle b\rangle\rangle.\end{aligned}$$

Now we can continue with the evaluation of power($\langle a, b\rangle$):

$$\begin{aligned}\text{power}(\langle a,b\rangle) &= \text{cat}(\text{power}(\langle b\rangle), \text{map}(::, \text{dist}(a, \text{power}(\langle b\rangle)))) \\ &= \text{cat}(\langle\langle\,\rangle, \langle b\rangle\rangle, \text{map}(::, \text{dist}(a, \langle\langle\,\rangle, \langle b\rangle\rangle))) \\ &= \text{cat}(\langle\langle\,\rangle, \langle b\rangle\rangle, \text{map}(::, \langle\langle a, \langle\,\rangle\rangle, \langle a, \langle b\rangle\rangle\rangle)) \\ &= \text{cat}(\langle\langle\,\rangle, \langle b\rangle\rangle, \langle a :: \langle\,\rangle, a :: \langle b\rangle\rangle) \\ &= \text{cat}(\langle\langle\,\rangle, \langle b\rangle\rangle, \langle\langle a\rangle, \langle a, b\rangle\rangle) \\ &= \langle\langle\,\rangle, \langle b\rangle, \langle a\rangle, \langle a, b\rangle\rangle.\end{aligned}$$

3.2.6 Infinite Sequences

Let's see how some infinite sequences can be defined recursively. To illustrate the idea, suppose the function "ints" returns the following infinite sequence for any integer x:

$$\text{ints}(x) = \langle x, x+1, x+2, \ldots\rangle.$$

We'll assume that the list operations of cons, head, and tail work for infinite sequences. For example, the following relationships hold.

$$\begin{aligned}\text{ints}(x) &= x :: \text{ints}(x+1), \\ \text{head}(\text{ints}(x)) &= x, \\ \text{tail}(\text{ints}(x)) &= \text{ints}(x+1).\end{aligned}$$

p. 152

Even though the definition of ints does not conform to (3.6), it is still recursively defined because it is defined in terms of itself. If we executed the definition, an infinite loop would construct the infinite sequence. For example, ints(0) would construct the infinite sequence of natural numbers as follows:

$$\begin{aligned}\text{ints}(0) &= 0 :: \text{ints}(1) \\ &= 0 :: 1 :: \text{ints}(2) \\ &= 0 :: 1 :: 2 :: \text{ints}(3) \\ &= \ldots.\end{aligned}$$

In practice, an infinite sequence is used as an argument and it is evaluated only when some of its values are needed. Once the needed values are computed, the evaluation stops. This is an example of a technique called *lazy evaluation*. For example, the following function returns the nth element of an infinite sequence s.

$$\text{get}(n, s) = \text{if } n = 1 \text{ then head}(s) \text{ else get}(n-1, \text{tail}(s)).$$

example 3.31 Picking Elements

We'll get the third element from the infinite sequence ints(6) by unfolding the expression get(3, ints(6)).

$$\begin{aligned}
\text{get}\,(3, \text{ints}\,(6)) &= \text{get}\,(2, \text{tail}\,(\text{ints}\,(6))) \\
&= \text{get}\,(1, \text{tail}\,(\text{tail}\,(\text{ints}\,(6)))) \\
&= \text{head}\,(\text{tail}\,(\text{tail}\,(\text{ints}\,(6)))) \\
&= \text{head}\,(\text{tail}\,(\text{tail}\,(6 :: \text{ints}\,(7)))) \\
&= \text{head}\,(\text{tail}\,(\text{ints}\,(7))) \\
&= \text{head}\,(\text{tail}\,(7 :: \text{ints}\,(8))) \\
&= \text{head}\,(\text{ints}\,(8)) \\
&= \text{head}\,(8 :: \text{ints}\,(9)) \\
&= 8.
\end{aligned}$$

end example

example 3.32 Summing

Suppose we need a function to sum the first n elements in an infinite sequence s of integers. The following definition does the trick:

$$\text{sum}(n,\ s) = \text{if } n = 0 \text{ then } 0 \text{ else head}(s) + \text{sum}(n - 1,\ \text{tail}(s)).$$

We'll compute the sum of the first three numbers in ints(4):

$$\begin{aligned}
\text{sum}\,(3, \text{ints}\,(4)) &= 4 + \text{sum}\,(2, \text{ints}\,(5)) \\
&= 4 + 5 + \text{sum}\,(1, \text{ints}\,(6)) \\
&= 4 + 5 + 6 + \text{sum}\,(0, \text{ints}\,(7)) \\
&= 4 + 5 + 6 + 0 \\
&= 15.
\end{aligned}$$

end example

example 3.33 The Sieve of Eratosthenes

Suppose we want to study prime numbers. For example, we might want to find the 500th prime, we might want to find the difference between the 500th and 501st primes, and so on. One way to proceed might be to define functions to extract information from the following infinite sequence of all prime numbers.

$$\textit{Primes} = \langle 2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ \ldots \rangle.$$

We'll construct this infinite sequence by the method of Eratosthenes (called *the sieve of Eratosthenes*). The method starts with the infinite sequence ints(2):

$$\text{ints}(2) = \langle 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ \ldots \rangle.$$

The next step removes all multiples of 2 (except 2) to obtain the infinite sequence

$$\langle 2,\ 3,\ 5,\ 7,\ 9,\ 11,\ 13,\ 15,\ \ldots \rangle.$$

The next step removes all multiples of 3 (except 3) to obtain the infinite sequence

$$\langle 2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ \ldots \rangle.$$

The process continues in this way.

We can construct the desired infinite sequence of primes once we have the function to remove multiples of a number from an infinite sequence. If we let remove(n, s) denote the infinite sequence obtained from s by removing all multiples of n, then we can define the sieve process as follows for an infinite sequence s of numbers:

$$\text{sieve}(s) = \text{head}(s) :: \text{sieve}(\text{remove}(\text{head}(s),\ \text{tail}(s))).$$

But we need to define the remove function. Notice that for natural numbers m and n with $n > 0$ that we have the following equivalences:

$$m \text{ is a multiple of } n \text{ iff } n \text{ divides } m \text{ iff } m \bmod n = 0.$$

This allows us to write the following definition for the remove function:

$$\begin{aligned}
\text{remove}\,(n, s) = {} & \text{if head}\,(s) \bmod n = 0 \text{ then remove}\,(n, \text{tail}\,(s)) \\
& \text{else head}\,(s) :: \text{remove}\,(n, \text{tail}\,(s))\,.
\end{aligned}$$

Then our desired sequence of primes is represented by the expression

$$\textit{Primes} = \text{sieve}(\text{ints}(2)).$$

In the exercises we'll evaluate some functions dealing with primes.

end example

Exercises

Evaluating Recursively Defined Functions

1. Given the following definition for the nth Fibonacci number:

$$\begin{aligned}
&\text{fib}\,(0) = 0, \\
&\text{fib}\,(1) = 1, \\
&\text{fib}\,(n) = \text{fib}\,(n-1) + \text{fib}\,(n-2) \quad \text{if } n > 1.
\end{aligned}$$

Write down each step in the evaluation of fib(4).

2. Given the following definition for the length of a list:

$$\text{length}(L) = \text{if } L = \langle\ \rangle \text{ then } 0 \text{ else } 1 + \text{length}(\text{tail}(L)).$$

Write down each step in the evaluation of length($\langle r, s, t, u\rangle$).

3. For each of the two definitions of "makeTree" given by (3.9) and (3.10), write down all steps to evaluate makeTree($\langle\ \rangle$, $\langle 3, 2, 4\rangle$).

Numbers

4. Construct a recursive definition for each of the following functions, where all variables represent natural numbers.

   a. $f(n) = 0 + 2 + 4 + \cdots + 2n$.
   b. $f(n) = \text{floor}(0/2) + \text{floor}(1/2) + \cdots + \text{floor}(n/2)$.
   c. $f(k, n) = \gcd(1, n) + \gcd(2, n) + \cdots + \gcd(k, n)$ for $k > 0$.
   d. $f(n) = (0 \bmod 2) + (1 \bmod 3) + \cdots + (n \bmod (n + 2))$.
   e. $f(n, k) = 0 + k + 2k + \cdots + nk$.
   f. $f(n, k) = k + (k + 1) + (k + 2) + \cdots + (k + n)$.

Strings

5. Construct a recursive definition for each of the following string functions for strings over the alphabet $\{a, b\}$.

   a. $f(x)$ returns the reverse of x.
   b. $f(x) = xy$, where y is the reverse of x.
   c. $f(x, y)$ tests whether x is a prefix of y.
   d. $f(x, y)$ tests whether $x = y$.
   e. $f(x)$ tests whether x is a palindrome.

Lists

6. Construct a recursive definition for each of the following functions that involve lists. Use the infix form of cons in the recursive part of each definition. In other words, write $h :: t$ in place of cons(h, t).

   a. $f(n) = \langle 2n, 2(n-1), \ldots, 2, 0\rangle$, over natural numbers.
   b. $f(z) =$ if $z \geq 0$ then $\langle z, z-1, \ldots, 0\ \rangle$ else $\langle z, z+1, \ldots, 0\ \rangle$, over integers.
   c. max(L) is the maximum value in the nonempty list L of numbers.
   d. $f(x, \langle a_0, \ldots, a_n\rangle) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$.
   e. $f(L)$ is the list of elements x in list L that have property P.

   f. $f(a, \langle x_1, \ldots, x_n\rangle) = \langle x_1 + a, \ldots, x_n + a\rangle$.
   g. $f(a, \langle (x_1, y_1), \ldots, (x_n, y_n)\rangle) = \langle (x_1 + a, y_1), \ldots, (x_n + a, y_n)\rangle$.
   h. $f(g, \langle x_1, \ldots, x_n\rangle) = \langle (x_1, g(x_1)), \ldots, (x_n, g(x_n))\rangle$.
   i. $f(n) = \langle (0, n), (1, n-1), \ldots, (n-1, 1), (n, 0)\rangle$. *Hint*: Use Part (g).
   j. $f(g, h, \langle x_1, \ldots, x_n\rangle) = \langle (g(x_1), h(x_1)), \ldots, (g(x_n), h(x_n))\rangle$.

Using Cat or ConsR

7. Construct a recursive definition for each of the following functions that involve lists. Use the cat operation or consR operation in the recursive part of each definition. (Notice that for any list L and element x we have cat(L, $\langle x\rangle$) = consR(L, x).)

   a. $f(n) = \langle 0, 1, \ldots, n\rangle$.
   b. $f(n) = \langle 0, 2, 4, \ldots, 2n\rangle$.
   c. $f(n) = \langle 1, 3, 5, \ldots, 2n + 1\rangle$.
   d. $f(n, k) = \langle n, n + 1, n + 2, \ldots, n + k\rangle$.
   e. $f(n, k) = \langle 0, k, 2k, 3k, \ldots, nk\rangle$.
   f. $f(g, n) = \langle (0, g(0)), (1, g(1)), \ldots, (n, g(n))\rangle$.
   g. $f(n, m) = \langle n, n + 1, n + 2, \ldots, m - 1, m\rangle$, where $n \leq m$.

8. Let *insert* be a function that extends any binary function so that it evaluates a list of two or more arguments. For example,

$$\text{insert}(+, \langle 1, 4, 2, 9\rangle) = 1 + (4 + (2 + 9)) = 16.$$

Write a recursive definition for insert(f, L), where f is any binary function and L is a list of two or more arguments.

9. Write a recursive definition for the function "eq" to check two lists for equality.

10. Write recursive definitions for the following list functions.

    a. The function "last" that returns the last element of a nonempty list. For example, last($\langle a, b, c\rangle$) = c.
    b. The function "front" that returns the list obtained by removing the last element of a nonempty list. For example, front($\langle a, b, c\rangle$) = $\langle a, b\rangle$.

11. Write down a recursive definition for the function "pal" that tests a list of letters to see whether their concatenations form a palindrome. For example, pal($\langle r, a, d, a, r\rangle$) = true since *radar* is a palindrome. *Hint:* Use the functions of Exercise 10.

12. Solve the repeated element problem with the restriction that we want to keep the rightmost occurrence of each repeated element. *Hint:* Use the functions of Exercise 10.

Binary Trees

13. Given the algebraic expression $a + (b \cdot (d + e))$, draw a picture of the binary tree representation of the expression. Then write down the preorder, inorder, and postorder listings of the tree. Are any of the listings familiar to you?

14. Write down recursive definitions for each of the following procedures to print the nodes of a binary tree.

    a. In: Prints the nodes of a binary tree from an inorder traversal.
    b. Post: Prints the nodes of a binary tree from a postorder traversal.

15. Write down recursive definitions for each of the following functions. Include both the pattern-matching and if-then-else forms for each definition.

    a. leaves: Returns the number of leaf nodes in a binary tree.
    b. inOrd: Returns the inorder listing of nodes in a binary tree.
    c. postOrd: Returns the postorder listing of nodes in a binary tree.

16. Construct a recursive definition for each of the following functions that involve binary trees. Represent binary trees as lists where $\langle\ \rangle$ is the empty tree and any nonempty binary tree has the form $\langle L, r, R\rangle$, where r is the root and L and R are its left and right subtrees. Assume that nodes are numbers.

    a. $f(T)$ is the sum of the nodes of T.
    b. $f(T)$ is the depth of T, where the empty tree has depth -1.
    c. $f(T)$ is the list of nodes in T that have property p.
    d. $f(T)$ is the maximum value of the nodes in the nonempty binary tree T.

Trees and Algebraic Expressions

17. Recall from Section 1.4 that any algebraic expression can be represented as a tree and the tree can be represented as a list whose head is the root and whose tail is the list of operands in the form of trees. For example, the algebraic expression $a \times b + f(c, d, e)$, can be represented by the list

$$\langle +, \langle \times, \langle a\rangle, \langle b\rangle\rangle, \langle f, \langle c\rangle, \langle d\rangle, \langle e\rangle\rangle\rangle.$$

    a. Draw the picture of the tree for the given algebraic expression.
    b. Construct a recursive definition for the function "post" that takes an algebraic expression written in the form of a list, as above, and returns the list of nodes in the algebraic expression tree in postfix notation. For example,

$$\text{post}(\langle +, \langle \times, \langle a\rangle, \langle b\rangle\rangle, \langle f, \langle c\rangle, \langle d\rangle, \langle e\rangle\rangle\rangle) = \langle a, b, \times, c, d, e, f, +\rangle.$$

Relations as Lists of Tuples

18. Construct a recursive definition for each of the following functions that involve lists of tuples. If x is an n-tuple, then x_k represents the kth component of x.

    a. $f(k, L)$ is the list of kth components x_k of tuples x in the list L.
    b. $\text{sel}(k, a, L)$ is the list of tuples x in the list L such that $x_k = a$.

Sets Represented as Lists

19. Write a recursive definition for each of the following functions, in which the input arguments are sets represented as lists. Use the primitive operations of cons, head, and tail to build your functions (along with functions already defined):

    a. isMember. For example, $\text{isMember}(a, \langle b, a, c\rangle)$ is true.
    b. isSubset. For example, $\text{isSubset}(\langle a, b\rangle, \langle b, c, a\rangle)$ is true.
    c. areEqual. For example, $\text{areEqual}(\langle a, b\rangle, \langle b, a\rangle)$ is true.
    d. union. For example, $\text{union}(\langle a, b\rangle, \langle c, a\rangle) = \langle a, b, c\rangle$.
    e. intersect. For example, $\text{intersect}(\langle a, b\rangle, \langle c, a\rangle) = \langle a\rangle$.
    f. difference. For example, $\text{difference}(\langle a, b, c\rangle, \langle b, d\rangle) = \langle a, c\rangle$.

Challenges

20. Conway's challenge sequence is defined recursively as follows:

$$\textit{Basis: } f(1) = f(2) = 1.$$
$$\textit{Recursion: } f(n) = f(f(n-1)) + f(n - f(n-1)) \quad \text{for } n > 2.$$

    Calculate the first 17 elements $f(1), f(2), \ldots, f(17)$. The article by Mallows [1991] contains an account of this sequence.

21. Let $\text{fib}(k)$ denote the kth Fibonacci number, and let

$$\text{sum}(k) = 1 + 2 + \cdots + k.$$

    Write a recursive definition for the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = \text{sum}(\text{fib}(n))$. *Hint:* Write down several examples, such as $f(0)$, $f(1)$, $f(2)$, $f(3)$, $f(4)$, ... Then try to find a way to write $f(4)$ in terms of $f(3)$. This might help you discover a pattern.

22. Write a function in if-then-else form to produce the Cartesian product set of two finite sets. You may assume that the sets are represented as lists.

23. We can approximate the square root of a number by using the Newton–Raphson method, which gives an infinite sequence of approximations to the

square root of x by starting with an initial guess g. We can define the sequence with the following function:

$$\text{sqrt}(x, g) = g :: \text{sqrt}(x, (0.5)(g + (x/g))).$$

Find the first three numbers in each of the following infinite sequences, and compare the values with the square root obtained with a calculator.

a. sqrt(4, 1). b. sqrt(4, 2). **c.** sqrt(4, 3).
d. sqrt(2, 1). **e.** sqrt(9, 1). f. sqrt(9, 5).

24. Find a definition for each of the following infinite sequence functions.

a. Square: Squares each element in a sequence of numbers.
b. Diff: Finds the difference of the nth and mth numbers of a sequence.
c. Prod: Finds the product of the first n numbers of a sequence.
d. Add: Adds corresponding elements of two numeric sequences.
e. $\text{Skip}(x, k) = \langle x, x + k, x + 2k, x + 3k, \ldots \rangle$.
f. Map: Applies a function to each element of a sequence.
g. ListOf: Finds the list of the first n elements of a sequence.

25. Evaluate each of the following expressions by unfolding the definitions for *Primes* and remove from Example 3.33.

a. head(*Primes*).
b. tail(*Primes*) until reaching the value sieve(remove (2, ints (3))).
c. remove(2, ints (0)) until reaching the value 1 :: 2 :: remove(2, ints (4)).

26. Suppose we define the function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \text{if } x > 10 \text{ then } x - 10 \text{ else } f(f(x + 11)).$$

This function is recursively defined even though it is not defined by (3.6). Give a simple definition of the function.

3.3 Grammars

Informally, a grammar is a set of rules used to define the structure of the strings in a language. Grammars are important in computer science not only for defining programming languages, but also for defining data sets for programs. Typical applications try to build algorithms that test whether or not an arbitrary string belongs to some language. In this section we'll see that grammars provide a convenient and useful way to describe languages in a fashion similar to an inductive definition, which we discussed in Section 3.1. We'll also see that grammars provide a technique to test whether a string belongs to a language in a fashion similar to the calculation of a recursively defined function, which we described in Section 3.2. So let's get to it.

3.3.1 Recalling English Grammar

We can think of an English sentence as a string of characters if we agree to let the alphabet consist of the usual letters together with the blank character, period, comma, and so on. To *parse* a sentence means to break it up into parts that conform to a given grammar.

For example, if an English sentence consists of a subject followed by a predicate, then the sentence

"The big dog chased the cat."

would be broken up into two parts, a subject and a predicate, as follows:

subject = The big dog,
predicate = chased the cat.

To denote the fact that a sentence consists of a subject followed by a predicate we'll write the following *grammar rule*:

sentence → subject predicate.

If we agree that a subject can be an article followed by either a noun or an adjective followed by a noun, then we can break up "The big dog" into smaller parts. The corresponding grammar rule can be written as follows:

subject → article adjective noun.

Similarly, if we agree that a predicate is a verb followed by an object, then we can break up "chased the cat" into smaller parts. The corresponding grammar rule can be written as follows:

predicate → verb object.

This is the kind of activity that can be used to detect whether or not a sentence is grammatically correct.

A parsed sentence is often represented as a tree, called the *parse tree* or *derivation tree*. The parse tree for "The big dog chased the cat" is pictured in Figure 3.7.

3.3.2 Structure of Grammars

Now that we've recalled a bit of English grammar, let's describe the general structure of grammars for arbitrary languages. If L is a language over an alphabet A, then a grammar for L consists of a set of *grammar rules* of the form

$$\alpha \to \beta,$$

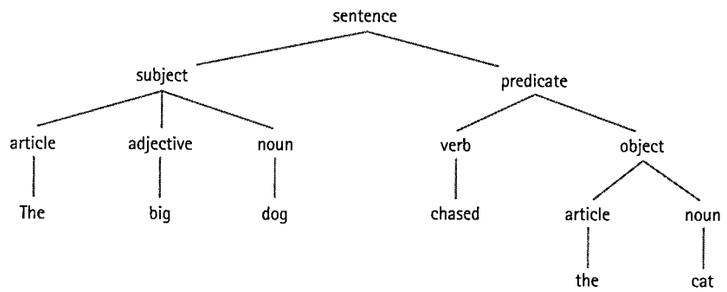


Figure 3.7 Parse tree.

where α and β denote strings of symbols taken from A and from a set of grammar symbols disjoint from A.

The grammar rule $\alpha \to \beta$ is often called a *production*, and it can be read in several different ways as

replace α by β,
α produces β,
α rewrites to β,
α reduces to β.

Every grammar has a special grammar symbol called the *start symbol*, and there must be at least one production with the left side consisting of only the start symbol. For example, if S is the start symbol for a grammar, then there must be at least one production of the form

$$S \to \beta.$$

A Beginning Example

Let's give an example of a grammar for a language and then discuss the process of deriving strings from the productions. Let $A = \{a, b, c\}$. Then a grammar for the language A^* can be described by the following four productions:

$$\begin{aligned} S &\to \Lambda \\ S &\to aS \\ S &\to bS \\ S &\to cS. \end{aligned} \tag{3.11}$$

How do we know that this grammar describes the language A^*? We must be able to describe each string of the language in terms of the grammar rules. For example, let's see how we can use the productions (3.11) to show that the string $aacb$ is in A^*. We'll begin with the start symbol S. Next we'll replace S

by the right side of production $S \to aS$. We chose production $S \to aS$ because $aacb$ matches the right-hand side of $S \to aS$ by letting $S = acb$. The process of replacing S by aS is called a *derivation step*, which we denote by writing

$$S \Rightarrow aS.$$

A *derivation* is a sequence of derivation steps. The right-hand side of this derivation contains the symbol S. So we again replace S by aS using the production $S \to aS$ a second time. This results in the derivation

$$S \Rightarrow aS \Rightarrow aaS.$$

The right-hand side of this derivation contains S. In this case we'll replace S by the right side of $S \to cS$. This gives the derivation

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS.$$

Continuing, we replace S by the right side of $S \to bS$. This gives the derivation

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS.$$

Since we want this derivation to produce the string $aacb$, we now replace S by the right side of $S \to \Lambda$. This gives the desired derivation of the string $aacb$:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\Lambda = aacb.$$

Each step in a derivation corresponds to attaching a new subtree to the parse tree whose root is the start symbol. For example, the parse trees corresponding to the first three steps of our example are shown in Figure 3.8. The completed derivation and parse tree are shown in Figure 3.9.

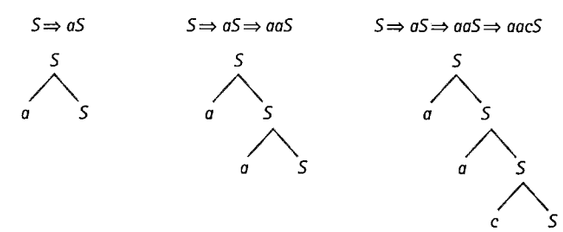


Figure 3.8 Partial derivations and parse trees.

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\Lambda = aacb$

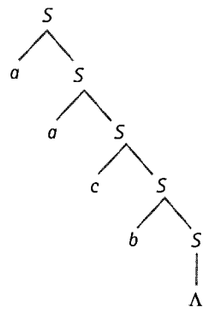


Figure 3.9 Derivation and parse tree.

Definition of a Grammar

Now that we've introduced the idea of a grammar, let's take a minute to describe the four main ingredients of any grammar.

The Four Parts of a Grammar (3.12)

1. An alphabet N of grammar symbols called *nonterminals*.
2. An alphabet T of symbols called *terminals*. The terminals are distinct from the nonterminals.
3. A specific nonterminal S, called the *start* symbol.
4. A finite set of productions of the form $\alpha \to \beta$, where α and β are strings over the alphabet $N \cup T$ with the restriction that α is not the empty string. There is at least one production with only the start symbol S on its left side. Each nonterminal must appear on the left side of some production.

Assumption: In this chapter, all grammar productions will have a single nonterminal on the left side. In Chapter 14 we'll see examples of grammars that allow productions to have strings of more than one symbol on the left side.

When two or more productions have the same left side, we can simplify the notation by writing one production with alternate right sides separated by the vertical line |. For example, the four productions (3.11) can be written in the following shorthand form:

$$S \to \Lambda \mid aS \mid bS \mid cS,$$

and we say, "S can be replaced by either Λ , or aS, or bS, or cS."

We can represent a grammar G as a 4-tuple $G = (N, T, S, P)$, where P is the set of productions. For example, if P is the set of productions (3.11), then the grammar can be represented by the 4-tuple

$$(\{S\}, \{a, b, c\}, S, P).$$

The 4-tuple notation is useful for discussing general properties of grammars. But for a particular grammar it's common practice to write down only the productions of the grammar, where the nonterminals are uppercase letters and the first production listed contains the start symbol on its left side. For example, suppose we're given the following grammar:

$$\begin{aligned} S &\to AB \\ A &\to \Lambda \mid aA \\ B &\to \Lambda \mid bB. \end{aligned} \tag{3.13}$$

We can deduce that the nonterminals are S, A, and B, the start symbol is S, and the terminals are a and b.

3.3.3 Derivations

To discuss grammars further, we need to formalize things a bit. Suppose we're given some grammar. A string made up of terminals and/or nonterminals is called a *sentential form*. Now we can formalize the idea of a derivation.

Definition of Derivation

If x and y are sentential forms and $\alpha \to \beta$ is a production, then the replacement of α by β in $x\alpha y$ is called a *derivation step*, which we denote by writing

$$x\alpha y \Rightarrow x\beta y.$$

A *derivation* is a sequence of derivation steps.

The following three symbols with their associated meanings are used quite often in discussing derivations:

$\Rightarrow$	derives in one step,
$\Rightarrow^+$	derives in one or more steps,
$\Rightarrow^*$	derives in zero or more steps.

For example, let's consider the previous grammar (3.13) and the string aab.

The statement $S \Rightarrow^+ aab$ means that there exists a derivation of aab that takes one or more steps. For example, we have the derivation

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aab.$$

In some grammars it may be possible to find several different derivations of the same string. Two kinds of derivations are worthy of note. A derivation is called a *leftmost derivation* if at each step the leftmost nonterminal of the sentential form is reduced by some production. Similarly, a derivation is called a *rightmost derivation* if at each step the rightmost nonterminal of the sentential form is reduced by some production. For example, the preceding derivation of aab is a leftmost derivation. Here's a rightmost derivation of aab:

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow Ab \Rightarrow aAb \Rightarrow aaAb \Rightarrow aab.$$

The Language of a Grammar

Sometimes it can be quite difficult, or impossible, to write down a grammar for a given language. So we had better nail down the idea of the language that is associated with a grammar. If G is a grammar, then the *language of* G is the set of terminal strings derived from the start symbol of G. The language of G is denoted by

$$L(G).$$

We can also describe $L(G)$ more formally.

The Language of a Grammar **(3.14)**
If G is a grammar with start symbol S and set of terminals T, then the language of G is the set

$$L(G) = \{s \mid s \in T^* \text{ and } S \Rightarrow^+ s\}.$$

When we're trying to write a grammar for a language, we should at least check to see whether the language is finite or infinite. If the language is finite, then a grammar can consist of all productions of the form $S \to w$ for each string w in the language. For example, the language $\{a, ab\}$ can be described by the grammar $S \to a \mid ab$.

If the language is infinite, then some production or sequence of productions must be used repeatedly to construct the derivations. To see this, notice that there is no bound on the length of strings in an infinite language. Therefore, there is no bound on the number of derivation steps used to derive the strings. If the grammar has n productions, then any derivation consisting of $n + 1$ steps must use some production twice (by the pigeonhole principle).

For example, the infinite language $\{a^n b \mid n \geq 0\}$ can be described by the grammar

$$S \to b \mid aS.$$

To derive the string $a^n b$, we would use the production $S \to aS$ repeatedly—n times to be exact—and then stop the derivation by using the production $S \to b$. The situation is similar to the way we make inductive definitions for sets. For example, the production $S \to aS$ allows us to make the informal statement "If S derives w, then it also derives aw."

Recursive Productions

A production is called *recursive* if its left side occurs on its right side. For example, the production $S \to aS$ is recursive. A production $A \to \alpha$ is *indirectly recursive* if A derives a sentential form that contains A. For example, suppose we have the following grammar:

$$S \to b \mid aA$$
$$A \to c \mid bS.$$

The productions $S \to aA$ and $A \to bS$ are both indirectly recursive because of the following derivations:

$$S \Rightarrow aA \Rightarrow abS,$$
$$A \Rightarrow bS \Rightarrow baA.$$

A grammar is *recursive* if it contains either a recursive production or an indirectly recursive production. So we can make the following more precise statement about grammars for infinite languages:

A grammar for an infinite language must be recursive.

Now let's look at the opposite problem of describing the language of a grammar. We know—by definition—that the language of a grammar is the set of all strings derived from the grammar. But we can also make another interesting observation about any language defined by a grammar:

Any language defined by a grammar is an inductively defined set.

Let's see why this is the case for any grammar G. The following inductive definition does the job, where S denotes the start symbol of G. To simplify the description, we'll say that a derivation is *recursive* if some nonterminal occurs twice due to a recursive production or due to a series of indirectly recursive productions.

Constructing an Inductive Definition for $L(G)$ (3.15)

Basis: If $S \Rightarrow^+ w$ without using a recursive derivation, then put w in $L(G)$.

Induction: Let $w \in L(G)$ with derivation $S \Rightarrow^+ w$. If the derivation contains a nonterminal A from a recursive production $A \to \alpha A \beta$ or from an indirectly recursive production corresponding to $A \Rightarrow^+ \alpha A \beta$, then modify the derivation by using either $A \to \alpha A \beta$ or $A \Rightarrow^+ \alpha A \beta$ to obtain a new derivation $S \Rightarrow^+ x$ and put x in $L(G)$.

Proof: Let G be a grammar and let M be the inductive set defined by (3.15). We need to show that $M = L(G)$. It's clear that $M \subseteq L(G)$ because all strings in M are derived from the start symbol of G. Assume, by way of contradiction, that $M \neq L(G)$. In other words, we have $L(G) - M \neq \varnothing$. Since S derives all the elements of $L(G) - M$, there must be some string $w \in L(G) - M$ that has the shortest leftmost derivation among elements of $L(G) - M$. We can assume that this derivation is recursive. Otherwise, the basis case of (3.15) would force us to put $w \in M$, contrary to our assumption that $w \in L(G) - M$. So the leftmost derivation of w must have the following form, where s and t are terminal strings and α, β, and γ are sentential forms that don't include B:

$$S \Rightarrow^+ sB\gamma \Rightarrow^+ stB\beta\gamma \Rightarrow st\alpha\beta\gamma \Rightarrow^* w.$$

We can replace $sB\gamma \Rightarrow^+ stB\beta\gamma$ in this derivation with $sB\gamma \Rightarrow s\alpha\gamma$ to obtain the following derivation of a string u of terminals:

$$S \Rightarrow^+ sB\gamma \Rightarrow s\alpha\gamma \Rightarrow^* u.$$

This derivation is shorter than the derivation of w. So we must conclude that $u \in M$. Now we can apply the induction part of (3.15) to this latter derivation of u to obtain the derivation of w. This tells us that $w \in M$, contrary to our assumption that $w \notin M$. The only thing left for us to conclude is that our assumption that $M \neq L(G)$ was wrong. Therefore $M = L(G)$. QED.

example 3.34 From Grammar to Inductive Definition

Let's do a simple example to illustrate the use of (3.15). Suppose we're given the following grammar G:

$$S \to \Lambda \mid aB$$
$$B \to b \mid bB.$$

We'll give an inductive definition for $L(G)$. There are two derivations that don't contain recursive productions: $S \Rightarrow \Lambda$ and $S \Rightarrow aB \Rightarrow ab$. This gives us the basis part of the definition for $L(G)$.

Basis: Λ, $ab \in L(G)$.

Now let's find the induction part of the definition. The only recursive production of G is $B \to bB$. So any element of $L(G)$ whose derivation contains an occurrence of B must have the general form $S \Rightarrow aB \Rightarrow^+ ay$ for some string y. So we can use the production $B \to bB$ to add one more step to the derivation as follows:

$$S \Rightarrow aB \Rightarrow abB \Rightarrow^+ aby.$$

This gives us the induction step in the definition of $L(G)$.

Induction: If $ay \in L(G)$, then put aby in $L(G)$.

For example, the basis case tells us that $ab \in L(G)$ and the derivation $S \Rightarrow aB \Rightarrow ab$ contains an occurrence of B. So we add one more step to the derivation using the production $B \to bB$ to obtain the derivation

$$S \Rightarrow aB \Rightarrow abB \Rightarrow abb.$$

So $ab \in L(G)$ implies that $abb \in L(G)$, which in turn implies $ab^3 \in L(G)$, and so on. Thus we can conjecture with some confidence that $L(G)$ is the language $\{\Lambda\} \cup \{ab^n \mid n \in \mathbb{N}\}$.

end example

3.3.4 Constructing Grammars

Now let's get down to business and construct some grammars. We'll start with a few simple examples, and then we'll give some techniques for combining simple grammars. We should note that a language might have more than one grammar. So we shouldn't be surprised when two people come up with two different grammars for the same language.

example 3.35 Three Simple Grammars

We'll write a grammar for each of three simple languages. In each case we'll include a sample derivation of a string in the language. Test each grammar by constructing a few more derivations for strings.

1. $\{\Lambda, a, aa, \ldots, a^n, \ldots\} = \{a^n \mid n \in \mathbb{N}\}$.

   Notice that any string in this language is either Λ or of the form ax for some string x in the language. The following grammar will derive any of these strings:

$$S \to \Lambda \mid aS.$$

For example, we'll derive the string aaa:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa.$$

2. $\{\Lambda, ab, aabb, \ldots, a^n b^n, \ldots\} = \{a^n b^n \mid n \in \mathbb{N}\}$.

Notice that any string in this language is either Λ or of the form axb for some string x in the language. The following grammar will derive any of these strings:

$$S \to \Lambda \mid aSb.$$

For example, we'll derive the string $aaabbb$:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

3. $\{\Lambda, ab, abab, \ldots, (ab)^n, \ldots\} = \{(ab)^n \mid n \in \mathbb{N}\}$.

Notice that any string in this language is either Λ or of the form abx for some string x in the language. The following grammar will derive any of these strings:

$$S \to \Lambda \mid abS.$$

For example, we'll derive the string $ababab$:

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow ababab.$$

end example

Combining Grammars

Sometimes a language can be written in terms of simpler languages, and a grammar can be constructed for the language in terms of the grammars for the simpler languages. We'll concentrate here on the operations of union, product, and closure.

Combining Grammars (3.16)

Suppose M and N are languages whose grammars have disjoint sets of nonterminals. (Rename them if necessary.) Suppose also that the start symbols for the grammars of M and N are A and B, respectively. Then we have the following new languages and grammars:

Union Rule: The language $M \cup N$ starts with the two productions

$$S \to A \mid B.$$

Continued ➡

Product Rule: The language MN starts with the production

$$S \to AB.$$

Closure Rule: The language M^* starts with the two productions

$$S \to AS \mid \Lambda.$$

example 3.36 Using the Union Rule

Let's write a grammar for the following language:

$$L = \{\Lambda, a, b, aa, bb, \ldots, a^n, b^n, \ldots\}.$$

After some thinking we notice that L can be written as a union $L = M \cup N$, where $M = \{a^n \mid n \in \mathbb{N}\}$ and $N = \{b^n \mid n \in \mathbb{N}\}$. Thus we can write the following grammar for L.

$S \to A \mid B$	union rule,
$A \to \Lambda \mid aA$	grammar for M,
$B \to \Lambda \mid bB$	grammar for N.

end example

example 3.37 Using the Product Rule

We'll write a grammar for the following language:

$$L = \{a^m b^n \mid m, n \in \mathbb{N}\}.$$

After a little thinking we notice that L can be written as a product $L = MN$, where $M = \{a^m \mid m \in \mathbb{N}\}$ and $N = \{b^n \mid n \in \mathbb{N}\}$. Thus we can write the following grammar for L:

$S \to AB$	product rule,
$A \to \Lambda \mid aA$	grammar for M,
$B \to \Lambda \mid bB$	grammar for N.

end example

example 3.38 Using the Closure Rule

We'll construct a grammar for the language $L = \{aa, bb\}^*$. If we let $M = \{aa, bb\}$, then $L = M^*$. Thus we can write the following grammar for L.

$$S \to AS \mid \Lambda \quad \text{closure rule,}$$
$$A \to aa \mid bb \quad \text{grammar for } M.$$

We can simplify this grammar by substituting for A to obtain the following grammar:

$$S \to aaS \mid bbS \mid \Lambda.$$

end example

example 3.39 Decimal Numerals

We can find a grammar for the language of decimal numerals by observing that a decimal numeral is either a digit or a digit followed by a decimal numeral. The following grammar rules reflect this idea:

$$S \to D \mid DS$$
$$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

We can say that S is replaced by either D or DS, and D can be replaced by any decimal digit. A derivation of the numeral 7801 can be written as follows:

$$S \Rightarrow DS \Rightarrow 7S \Rightarrow 7DS \Rightarrow 7DDS \Rightarrow 78DS \Rightarrow 780S \Rightarrow 780D \Rightarrow 7801.$$

This derivation is not unique. For example, another derivation of 7801 can be written as follows:

$$S \Rightarrow DS \Rightarrow DDS \Rightarrow D8S \Rightarrow D8DS \Rightarrow D80S \Rightarrow D80D \Rightarrow D801 \Rightarrow 7801.$$

end example

example 3.40 Even Decimal Numerals

We can find a grammar for the language of decimal numerals for the even natural numbers by observing that each numeral must have an even digit on its right side. In other words, either it's an even digit or it's a decimal numeral followed by an even digit. The following grammar will do the job:

$$S \to E \mid NE$$
$$N \to D \mid DN$$
$$E \to 0 \mid 2 \mid 4 \mid 6 \mid 8$$
$$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

For example, the even numeral 136 has the derivation

$$S \Rightarrow NE \Rightarrow N6 \Rightarrow DN6 \Rightarrow DD6 \Rightarrow D36 \Rightarrow 136.$$

end example

example 3.41 Identifiers

Most programming languages have identifiers for names of things. Suppose we want to describe a grammar for the set of identifiers that start with a letter of the alphabet followed by zero or more letters or digits. Let S be the start symbol. Then the grammar can be described by the following productions:

$$S \to L \mid LA$$
$$A \to LA \mid DA \mid \Lambda$$
$$L \to a \mid b \mid \ldots \mid z$$
$$D \to 0 \mid 1 \mid \ldots \mid 9.$$

We'll give a derivation of the string $a2b$ to show that it is an identifier.

$$S \Rightarrow LA \Rightarrow aA \Rightarrow aDA \Rightarrow a2A \Rightarrow a2LA \Rightarrow a2bA \Rightarrow a2b.$$

end example

example 3.42 Some Rational Numerals

Let's find a grammar for those rational numbers that have a finite decimal representation. In other words, we want to describe a grammar for the language of strings having the form $m.n$ or $-m.n$, where m and n are decimal numerals. For example, 0.0 represents the number 0. Let S be the start symbol. We can start the grammar with the two productions

$$S \to N.N \mid -N.N.$$

To finish the job, we need to write some productions that allow N to derive a decimal numeral. We can use the following renamed productions from Example 3.39:

$$N \to D \mid DN$$
$$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

end example

example 3.43 Palindromes

We can write a grammar for the set of all palindromes over an alphabet A. Recall that a palindrome is a string that is the same when written in reverse order. For example, let $A = \{a, b, c\}$. Let P be the start symbol. Then the language of palindromes over the alphabet A has the grammar

$$P \rightarrow aPa \mid bPb \mid cPc \mid a \mid b \mid c \mid \Lambda.$$

For example, the palindrome *abcba* can be derived as follows:

$$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abcba.$$

end example

3.3.5 Meaning and Ambiguity

Most of the time we attach meanings to the strings in our lives. For example, the string 3+4 means 7 to most people. The string 3–4–2 may have two distinct meanings to two different people. One person may think that

$$3\text{–}4\text{–}2 = (3\text{–}4)\text{–}2 = -3,$$

while another person might think that

$$3\text{–}4\text{–}2 = 3\text{–}(4\text{–}2) = 1.$$

If we have a grammar, then we can define the *meaning* of any string in the grammar's language to be the parse tree produced by a derivation. We can often write a grammar so that each string in the grammar's language has exactly one meaning (i.e., one parse tree). When this is not the case, we have an ambiguous grammar. Here's the formal definition.

Definition of Ambiguous Grammar

A grammar is said to be *ambiguous* if its language contains some string that has two different parse trees. This is equivalent to saying that some string has two distinct leftmost derivations or that some string has two distinct rightmost derivations.

To illustrate the ideas, we'll look at some grammars for simple arithmetic expressions. For example, suppose we define a set of arithmetic expressions by the grammar

$$E \rightarrow a \mid b \mid E\text{–}E.$$

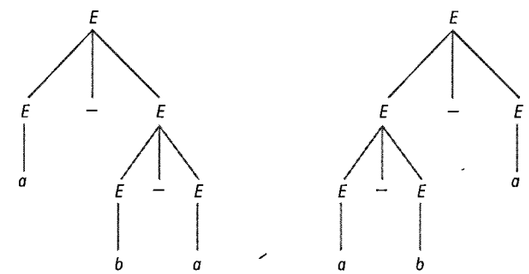


Figure 3.10 Parse trees for an ambiguous string.

The language of this grammar contains strings like a, b, b–a, a–b–a, and b–b–a–b. This grammar is ambiguous because it has a string, namely, a–b–a, that has two distinct parse trees as shown in Figure 3.10.

Since having two distinct parse trees means the same thing as having two distinct leftmost derivations, it's no problem to find the following two distinct leftmost derivations of a–b–a.

$$E \Rightarrow E - E \Rightarrow a - E \Rightarrow a - E - E \Rightarrow a - b - E \Rightarrow a - b - a.$$
$$E \Rightarrow E - E \Rightarrow E - E - E \Rightarrow a - E - E \Rightarrow a - b - E \Rightarrow a - b - a.$$

The two trees in Figure 3.10 reflect the two ways we could choose to evaluate a–b–a. The first tree indicates the meaning

$$a\text{–}b\text{–}a = a\text{–}(b\text{–}a),$$

while the second tree indicates

$$a\text{–}b\text{–}a = (a\text{–}b)\text{–}a.$$

How can we make sure there is only one parse tree for every string in the language? We can try to find a different grammar for the same set of strings. For example, suppose we want a–b–a to mean $(a$–$b)$–a. In other words, we want the first minus sign to be evaluated before the second minus sign. We can give the first minus sign higher precedence than the second by introducing a new nonterminal as shown in the following grammar:

$$E \rightarrow E - T \mid T$$
$$T \rightarrow a \mid b.$$

Notice that T can be replaced in a derivation only by either a or b. Therefore, every derivation of a–b–a produces a unique parse tree.

Exercises

Derivations

1. Given the following grammar:

$$S \to D \mid DS$$
$$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

a. Find the production used in each step of the following derivation.

$$S \Rightarrow DS \Rightarrow 7S \Rightarrow 7DS \Rightarrow 7DDS \Rightarrow 78DS \Rightarrow 780S \Rightarrow 780D \Rightarrow 7801.$$

b. Find a leftmost derivation of the string 7801.
c. Find a rightmost derivation of the string 7801.

2. Given the following grammar:

$$S \to S[S] \mid \Lambda.$$

For each of the following strings, construct a leftmost derivation, a rightmost derivation, and a parse tree.

a. [ ]. b. [[ ]]. **c.** [ ] [ ]. d. [[ ] [[ ]]].

Constructing Grammars

3. Find a grammar for each of the following languages.

a. $\{bb, bbbb, bbbbbb, \ldots\} = \left\{(bb)^{n+1} \mid n \in \mathbb{N}\right\}$.
b. $\{a, ba, bba, bbba, \ldots\} = \{b^n a \mid n \in \mathbb{N}\}$.
c. $\{\Lambda, ab, abab, ababab, \ldots\} = \{(ab)^n \mid n \in \mathbb{N}\}$.
d. $\{bb, bab, baab, baaab, \ldots\} = \{ba^n b \mid n \in \mathbb{N}\}$.
e. $\left\{ab, abab, \ldots, (ab)^{n+1}, \ldots\right\} = \left\{(ab)^{n+1} \mid n \in \mathbb{N}\right\}$.
f. $\{ab, aabb, \ldots, a^n b^n, \ldots\} = \{a^{n+1} b^{n+1} \mid n \in \mathbb{N}\}$.
g. $\{b, bbb, \ldots, b^{2n+1}, \ldots\} = \{b^{2n+1} \mid n \in \mathbb{N}\}$.
h. $\{b, abc, aabcc, \ldots, a^n bc^n, \ldots\} = \{a^n bc^n \mid n \in \mathbb{N}\}$.
i. $\{ac, abc, abbc, \ldots, ab^n c, \ldots\} = \{ab^n c \mid n \in \mathbb{N}\}$.
j. $\{\Lambda, aa, aaaa, \ldots, a^{2n}, \ldots\} = \{a^{2n} \mid n \in \mathbb{N}\}$.

4. Find a grammar for each language.

a. $\{a^m b^n \mid m, n \in \mathbb{N}\}$.
b. $\{a^m bc^n \mid m, n \in \mathbb{N}\}$.
c. $\{a^m b^n \mid m, n \in \mathbb{N}$, where $m > 0\}$.
d. $\{a^m b^n \mid m, n \in \mathbb{N}$, where $n > 0\}$.
e. $\{a^m b^n \mid m, n \in \mathbb{N}$, where $m > 0$ and $n > 0\}$.

5. Find a grammar for each language.

a. The even palindromes over $\{a, b, c\}$.
b. The odd palindromes over $\{a, b, c\}$.
c. $\{a^{2n} \mid n \in \mathbb{N}\} \cup \{b^{2n+1} \mid n \in \mathbb{N}\}$.
d. $\{a^n bc^n \mid n \in \mathbb{N}\} \cup \{b^m a^n \mid m, n \in \mathbb{N}\}$
e. $\{a^m b^n \mid m, n \in \mathbb{N}$, where $m > 0$ or $n > 0\}$.

Mathematical Expressions

6. Find a grammar for each of the following languages.

a. The set of binary numerals that represent odd natural numbers.
b. The set of binary numerals that represent even natural numbers.
c. The set of decimal numerals that represent odd natural numbers.

7. Find a grammar for each of the following languages.

a. The set of arithmetic expressions that are constructed from decimal numerals, +, and parentheses. Examples: 17, 2+3, (3+(4+5)), and 5+9+20.
b. The set of arithmetic expressions that are constructed from decimal numerals, – (subtraction), and parentheses, with the property that each expression has only one meaning. For example, 9–34–10 is not allowed.

8. Let the letters a, b, and c be constants; let the letters x, y, and z be variables; and let the letters f and g be functions of arity 1. We can define the set of terms over these symbols by saying that any constant or variable is a term and if t is a term, then so are $f(t)$ and $g(t)$.

a. Find a grammar for the set of terms.
b. Find a derivation for the expression $f(g(f(x)))$.

9. Let the letters a, b, and c be constants; let the letters x, y, and z be variables; and let the letters f and g be functions of arity 1 and 2, respectively. We can define the set of terms over these symbols by saying that any constant or variable is a term and if s and t are terms, then so are $f(t)$ and $g(s, t)$.

a. Find a grammar for the set of terms.
b. Find a derivation for the expression $f(g(x, f(b)))$.

10. Find a grammar to capture the precedence $\cdot$ over $+$ in the absence of parentheses. For example, the meaning of $a + b \cdot c$ should be $a + (b \cdot c)$.

Ambiguity

11. Show that each of the following grammars is ambiguous. In other words, find a string that has two different parse trees (equivalently, two different leftmost derivations or two different rightmost derivations).

    a. $S \to a \mid SbS$.
    b. $S \to abB \mid AB$ and $A \to \Lambda \mid Aa$ and $B \to \Lambda \mid bB$.
    c. $S \to aS \mid Sa \mid a$.
    d. $S \to aS \mid Sa \mid b$.
    e. $S \to S[S]S \mid \Lambda$.
    f. $S \to Ab \mid A$ and $A \to b \mid bA$.

Challenges

12. Find a grammar for the language of all strings over $\{a, b\}$ that have the same number of a's and b's.

13. For each grammar, try to find an equivalent grammar that is not ambiguous.

    a. $S \to a \mid SbS$.
    b. $S \to abB \mid AB$ and $A \to \Lambda \mid Aa$ and $B \to \Lambda \mid bB$.
    c. $S \to a \mid aS \mid Sa$.
    d. $S \to b \mid aS \mid Sa$.
    e. $S \to S[S]S \mid \Lambda$.
    f. $S \to Ab \mid A$ and $A \to b \mid bA$.

14. For each grammar, find an equivalent grammar that has no occurrence of Λ on the right side of any rule.

    a. $S \to AB$
       $A \to Aa \mid a$
       $B \to Bb \mid \Lambda$.

    b. $S \to AcAB$
       $A \to aA \mid \Lambda$
       $B \to bB \mid b$.

15. For each grammar G, use (3.15) to find an inductive definition for $L(G)$.

    a. $S \to \Lambda \mid aaS$.
    b. $S \to a \mid aBc$ and $B \to b \mid bB$.

3.4 Chapter Summary

This chapter covered some basic construction techniques that apply to many objects of importance to computer science.

Inductively defined sets are characterized by a basis step, an induction step, and a closure clause that is always assumed without comment. The constructors of an inductively defined set are the elements listed in the basis step and the rules specified in the induction step. Many sets of objects used in computer science can be defined inductively—numbers, strings, lists, binary trees, and Cartesian products of sets.

A recursively defined function is defined in terms of itself. Most recursively defined functions have domains that are inductively defined sets. These functions are normally defined by a basis case and a recursive case. The situation is similar for recursively defined procedures. Some infinite sequence functions can be defined recursively. Recursively defined functions and procedures yield powerful programs that are simply stated.

Grammars provide useful ways to describe languages. Grammar productions are used to derive the strings of a language. Any grammar for an infinite language must contain at least one production that is recursive or indirectly recursive. Grammars for different languages can be combined to form new grammars for unions, products, and closures of the languages. Some grammars are ambiguous.