7 Recursive Data Types

Recursive data types play a central role in programming, and induction is really all about them.

Recursive data types are specified by *recursive definitions*, which say how to construct new data elements from previous ones. Along with each recursive data type there are recursive definitions of properties or functions on the data type. Most importantly, based on a recursive definition, there is a *structural induction* method for proving that all data of the given type have some property.

This chapter examines a few examples of recursive data types and recursively defined functions on them:

- strings of characters,
- "balanced" strings of brackets,
- the nonnegative integers, and
- arithmetic expressions.
- two-player games with perfect information.

7.1 Recursive Definitions and Structural Induction

We'll start off illustrating recursive definitions and proofs using the example of character strings. Normally we'd take strings of characters for granted, but it's informative to treat them as a recursive data type. In particular, strings are a nice first example because you will see recursive definitions of things that are easy to understand, or that you already know, so you can focus on how the definitions work without having to figure out what they are supposed to mean.

Definitions of recursive data types have two parts:

- **Base case(s)** specifying that some known mathematical elements are in the data type, and
- **Constructor case(s)** that specify how to construct new data elements from previously constructed elements or from base elements.

The definition of strings over a given character set A follows this pattern:

Definition 7.1.1. Let A be a nonempty set called an *alphabet*, whose elements are referred to as *characters* (also called *letters*, *symbols*, or *digits*). The recursive data type A^* of strings over alphabet A is defined as follows:

- **Base case**: the empty string λ is in A^*.
- **Constructor case**: If $a \in A$ and $s \in A^*$, then the pair $\langle a, s\rangle \in A^*$.

So $\{0,1\}^*$ are the binary strings.

The usual way to treat binary strings is as sequences of 0's and 1's. For example, we have identified the length-4 binary string 1011 as a sequence of bits, the 4-tuple $(1,0,1,1)$. But according to the recursive Definition 7.1.1, this string would be represented by nested pairs, namely

$$\langle 1, \langle 0, \langle 1, \langle 1, \lambda\rangle\rangle\rangle\rangle\,.$$

These nested pairs are definitely cumbersome and may also seem bizarre, but they actually reflect the way that such lists of characters would be represented in programming languages like Scheme or Python, where $\langle a, s\rangle$ would correspond to $\text{cons}(a, s)$.

Notice that we haven't said exactly how the empty string is represented. It really doesn't matter, as long as we can recognize the empty string and not confuse it with any nonempty string.

Continuing the recursive approach, let's define the length of a string.

Definition 7.1.2. The length $|s|$ of a string s is defined recursively based on Definition 7.1.1.

Base case: $|\lambda| ::= 0$.

Constructor case: $|\langle a, s\rangle| ::= 1 + |s|$.

This definition of length follows a standard pattern: functions on recursive data types can be defined recursively using the same cases as the data type definition. Specifically, to define a function f on a recursive data type, define the value of f for the base cases of the data type definition, then define the value of f in each constructor case in terms of the values of f on the component data items.

Let's do another example: the *concatenation* $s \cdot t$ of the strings s and t is the string consisting of the letters of s followed by the letters of t. This is a perfectly clear mathematical definition of concatenation (except maybe for what to do with the empty string), and in terms of Scheme/Python lists, $s \cdot t$ would be the list $\text{append}(s, t)$. Here's a recursive definition of concatenation.

Definition 7.1.3. The *concatenation* $s \cdot t$ of the strings $s, t \in A^*$ is defined recursively based on Definition 7.1.1:

Base case:

$$\lambda \cdot t ::= t.$$

Constructor case:

$$\langle a, s\rangle \cdot t ::= \langle a, s \cdot t\rangle .$$

7.1.1 Structural Induction

Structural induction is a method for proving that all the elements of a recursively defined data type have some property. A structural induction proof has two parts corresponding to the recursive definition:

- Prove that each base case element has the property.
- Prove that each constructor case element has the property, when the constructor is applied to elements that have the property.

For example, in the base case of the definition of concatenation 7.1.3, we *defined* concatenation so the empty string was a "left identity," namely, $\lambda \cdot s ::= s$. We intend the empty string also to be a "right identity," namely, $s \cdot \lambda = s$. Being a right identity is not part of Definition 7.1.3, but we can prove it easily by structural induction:

Lemma 7.1.4.

$$s \cdot \lambda = s$$

for all $s \in A^*$.

Proof. The proof is by structural induction on the recursive definition 7.1.3 of concatenation. The induction hypothesis will be

$$P(s) ::= [s \cdot \lambda = s].$$

Base case: $(s = \lambda)$.

$$\begin{aligned} s \cdot \lambda &= \lambda \cdot \lambda \\ &= \lambda && (\lambda \text{ is a left identity by Def 7.1.3}) \\ &= s. \end{aligned}$$

Constructor case: $(s = \langle a, t\rangle)$.

$$\begin{aligned} s \cdot \lambda &= \langle a, t\rangle \cdot \lambda \\ &::= \langle a, t \cdot \lambda\rangle\,. && \text{(Constructor case of Def 7.1.3)} \\ &= \langle a, t\rangle && \text{by induction hypothesis } P(t) \\ &= s. \end{aligned}$$

So $P(s)$ holds. This completes the proof of the constructor case, and we conclude by structural induction that equation (7.1.4) holds for all $s \in A^*$. ■

We can also verify properties of recursive functions by structural induction on their definitions. For example, let's verify the familiar fact that the length of the concatenation of two strings is the sum of their lengths:

Lemma.

$$|s \cdot t| = |s| + |t|$$

for all $s, t \in A^*$.

Proof. By structural induction on the definition of $s \in A^*$. The induction hypothesis is

$$P(s) ::= \forall t \in A^*.\, |s \cdot t| = |s| + |t|.$$

Base case $(s = \lambda)$:

$$\begin{aligned} |s \cdot t| &= |\lambda \cdot t| \\ &= |t| && \text{(base case of Def 7.1.3 of concatenation)} \\ &= 0 + |t| \\ &= |s| + |t| && \text{(Def of } |\lambda|). \end{aligned}$$

Constructor case: $(s ::= \langle a, r\rangle)$.

$$\begin{aligned} |s \cdot t| &= |\,\langle a, r\rangle \cdot t\,| \\ &= |\,\langle a, r \cdot t\rangle\,| && \text{(constructor case of Def of concat)} \\ &= 1 + |r \cdot t| && \text{(constructor case of def length)} \\ &= 1 + (|r| + |t|) && \text{(ind. hyp. } P(r)) \\ &= (1 + |r|) + |t| \\ &= |\,\langle a, r\rangle\,| + |t| && \text{(constructor case, def of length)} \\ &= |s| + |t|. \end{aligned}$$

This proves that $P(s)$ holds, completing the constructor case. By structural induction, we conclude that $P(s)$ holds for all strings $s \in A^*$. ■

These proofs illustrate the general principle:

The Principle of Structural Induction.

Let P be a predicate on a recursively defined data type R. If

- $P(b)$ is true for each base case element $b \in R$, and
- for all two-argument constructors $\mathbf{c}$,

$$[P(r) \text{ AND } P(s)] \text{ IMPLIES } P(\mathbf{c}(r,s))$$

for all $r, s \in R$,
and likewise for all constructors taking other numbers of arguments,

then

$P(r)$ is true for all $r \in R$.

7.2 Strings of Matched Brackets

Let $\{\texttt{]}, \texttt{[}\}^*$ be the set of all strings of square brackets. For example, the following two strings are in $\{\texttt{]}, \texttt{[}\}^*$:

$$\texttt{]][[[[]]} \quad \text{and} \quad \texttt{[[]][]][]} \tag{7.1}$$

A string $s \in \{\texttt{]}, \texttt{[}\}^*$ is called a *matched string* if its brackets "match up" in the usual way. For example, the left-hand string above is not matched because its second right bracket does not have a matching left bracket. The string on the right is matched.

We're going to examine several different ways to define and prove properties of matched strings using recursively defined sets and functions. These properties are pretty straightforward, and you might wonder whether they have any particular relevance in computer science. The honest answer is "not much relevance *any more*." The reason for this is one of the great successes of computer science, as explained in the text box below.

Expression Parsing

During the early development of computer science in the 1950's and 60's, creation of effective programming language compilers was a central concern. A key aspect in processing a program for compilation was expression parsing. One significant problem was to take an expression like

$$x + y * z^2 \div y + 7$$

and *put in* the brackets that determined how it should be evaluated—should it be

$$[[x + y] * z^2 \div y] + 7, \text{ or,}$$
$$x + [y * z^2 \div [y + 7]], \text{ or,}$$
$$[x + [y * z^2]] \div [y + 7], \text{ or} \ldots?$$

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert W. Floyd, for, among other things, discovering simple procedures that would insert the brackets properly.

In the 70's and 80's, this parsing technology was packaged into high-level compiler-compilers that automatically generated parsers from expression grammars. This automation of parsing was so effective that the subject no longer demanded attention. It had largely disappeared from the computer science curriculum by the 1990's.

The matched strings can be nicely characterized as a recursive data type:

Definition 7.2.1. Recursively define the set RecMatch of strings as follows:

- **Base case**: $\lambda \in \text{RecMatch}$.
- **Constructor case**: If $s, t \in \text{RecMatch}$, then

$$[\, s \,]\, t \in \text{RecMatch}.$$

Here $[\, s \,]\, t$ refers to the concatenation of strings which would be written in full as

$$[\, \cdot (s \cdot (]\, \cdot t)).$$

From now on, we'll usually omit the "·'s."

Using this definition, $\lambda \in \text{RecMatch}$ by the base case, so letting $s = t = \lambda$ in the constructor case implies

$$[\, \lambda]\, \lambda = [\,] \in \text{RecMatch}.$$

Now,

$$\begin{aligned} [\lambda][\,] = [\,][\,] &\in \text{RecMatch} && (\text{letting } s = \lambda, t = [\,]) \\ [[\,]]\lambda = [[\,]] &\in \text{RecMatch} && (\text{letting } s = [\,], t = \lambda) \\ [[\,]][\,] &\in \text{RecMatch} && (\text{letting } s = [\,], t = [\,]) \end{aligned}$$

are also strings in RecMatch by repeated applications of the constructor case; and so on.

It's pretty obvious that in order for brackets to match, there had better be an equal number of left and right ones. For further practice, let's carefully prove this from the recursive definitions, beginning with a recursive definition of the number $\#_c(s)$ of occurrences of the character $c \in A$ in a string s:

Definition 7.2.2.

Base case: $\#_c(\lambda) ::= 0$.

Constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

The following Lemma follows directly by structural induction on Definition 7.2.2. We'll leave the proof for practice (Problem 7.9).

Lemma 7.2.3.

$$\#_c(s \cdot t) = \#_c(s) + \#_c(t).$$

Lemma. *Every string in RecMatch has an equal number of left and right brackets.*

Proof. The proof is by structural induction with induction hypothesis

$$P(s) ::= \left[\#_{[}(s) = \#_{]}(s)\right].$$

Base case: $P(\lambda)$ holds because

$$\#_{[}(\lambda) = 0 = \#_{]}(\lambda)$$

by the base case of Definition 7.2.2 of $\#_c()$.

Constructor case: By structural induction hypothesis, we assume $P(s)$ and $P(t)$ and must show $P([\,s\,]\,t)$:

$$\begin{aligned}
\#_{[}([\,s\,]\,t) &= \#_{[}([\,) + \#_{[}(s) + \#_{[}(]\,) + \#_{[}(t) && \text{(Lemma 7.2.3)}\\
&= 1 + \#_{[}(s) + 0 + \#_{[}(t) && \text{(def } \#_{[}\text{())}\\
&= 1 + \#_{]}(s) + 0 + \#_{]}(t) && \text{(by } P(s) \text{ and } P(t))\\
&= 0 + \#_{]}(s) + 1 + \#_{]}(t)\\
&= \#_{]}([\,) + \#_{]}(s) + \#_{]}(]\,) + \#_{]}(t) && \text{(def } \#_{]}\text{())}\\
&= \#_{]}([\,s\,]\,t) && \text{(Lemma 7.2.3)}
\end{aligned}$$

This completes the proof of the constructor case. We conclude by structural induction that $P(s)$ holds for all $s \in$ RecMatch. ■

Warning: When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on its definition would always be well-defined. Recursively defining a function on an ambiguous data type definition usually will not work. To illustrate the problem, here's another definition of the matched strings.

Definition 7.2.4. Define the set, AmbRecMatch $\subseteq \{]\,,[\,\}^*$ recursively as follows:

- **Base case**: $\lambda \in$ AmbRecMatch,
- **Constructor cases**: if $s, t \in$ AmbRecMatch, then the strings $[\,s\,]$ and st are also in AmbRecMatch.

It's pretty easy to see that the definition of AmbRecMatch is just another way to define RecMatch, that is AmbRecMatch = RecMatch (see Problem 7.20). The definition of AmbRecMatch is arguably easier to understand, but we didn't use it because it's ambiguous, while the trickier definition of RecMatch is unambiguous. Here's an example that illustrates why this matters. Let's define the number of operations $f(s)$ to construct a matched string s recursively on the definition of $s \in$ AmbRecMatch:

$$\begin{aligned}
f(\lambda) &::= 0, && (f \text{ base case})\\
f([\,s\,]\,) &::= 1 + f(s),\\
f(st) &::= 1 + f(s) + f(t). && (f \text{ concat case})
\end{aligned}$$

This definition may seem ok, but it isn't: $f(\lambda)$ winds up with two values, and consequently:

$$\begin{aligned} 0 &= f(\lambda) && (f \text{ base case})) \\ &= f(\lambda \cdot \lambda) && (\text{concat def, base case}) \\ &= 1 + f(\lambda) + f(\lambda) && (f \text{ concat case}), \\ &= 1 + 0 + 0 = 1 && (f \text{ base case}). \end{aligned}$$

This is definitely not a situation we want to be in!

7.3 Recursive Functions on Nonnegative Integers

The nonnegative integers can be understood as a recursive data type.

Definition 7.3.1. The set $\mathbb{N}$ is a data type defined recursively as:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then the *successor* $n + 1$ of n is in $\mathbb{N}$.

The point here is to make it clear that ordinary induction is simply the special case of structural induction on the recursive Definition 7.3.1. This also justifies the familiar recursive definitions of functions on the nonnegative integers.

7.3.1 Some Standard Recursive Functions on $\mathbb{N}$

Example 7.3.2. *The factorial function.* This function is often written "$n!$." You will see a lot of it in later chapters. Here, we'll use the notation $\text{fac}(n)$:

- $\text{fac}(0) ::= 1$.
- $\text{fac}(n+1) ::= (n+1) \cdot \text{fac}(n)$ for $n \geq 0$.

Example 7.3.3. *Summation notation.* Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define $S(n)$ with the rules

- $S(0) ::= 0$.
- $S(n+1) ::= f(n+1) + S(n)$ for $n \geq 0$.

7.3.2 Ill-formed Function Definitions

There are some other blunders to watch out for when defining functions recursively. The main problems come when recursive definitions don't follow the recursive definition of the underlying data type. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but really aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{7.2}$$

This "definition" has no base case. If some function f_1 satisfied (7.2), so would a function obtained by adding a constant to the value of f_1. So equation (7.2) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{7.3}$$

This "definition" has a base case, but still doesn't uniquely determine f_2. Any function that is 0 at 0 and constant everywhere else would satisfy the specification, so (7.3) also does not uniquely define anything.

In a typical programming language, evaluation of $f_2(1)$ would begin with a recursive call of $f_2(2)$, which would lead to a recursive call of $f_2(3)$, ... with recursive calls continuing without end. This "operational" approach interprets (7.3) as defining a *partial* function f_2 that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{7.4}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (7.4) doesn't define anything.

Mathematicians have been wondering about this function specification, known as the Collatz conjecture for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{7.5}$$

For example, $f_4(3) = 1$ because

$$f_4(3) ::= f_4(10) ::= f_4(5) ::= f_4(16) ::= f_4(8) ::= f_4(4) ::= f_4(2) ::= f_4(1) ::= 1.$$

The constant function equal to 1 will satisfy (7.5), but it's not known if another function does as well. The problem is that the third case specifies $f_4(n)$ in terms of f_4 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_4 satisfying (7.5) equals 1 for all n up to over 10^{18}.

A final example is the Ackermann function, which is an extremely fast-growing function of two nonnegative arguments. Its inverse is correspondingly slow-growing—it grows slower than $\log n$, $\log\log n$, $\log\log\log n$, ..., but it does grow unboundly. This inverse actually comes up analyzing a useful, highly efficient procedure known as the *Union-Find algorithm*. This algorithm was conjectured to run in a number of steps that grew linearly in the size of its input, but turned out to be "linear" but with a slow growing coefficient nearly equal to the inverse Ackermann function. This means that pragmatically, *Union-Find* is linear, since the theoretically growing coefficient is less than 5 for any input that could conceivably come up.

The Ackermann function can be defined recursively as the function A given by the following rules:

$$A(m,n) = 2n \qquad \text{if } m = 0 \text{ or } n \leq 1, \tag{7.6}$$

$$A(m,n) = A(m-1, A(m,n-1)) \qquad \text{otherwise.} \tag{7.7}$$

Now these rules are unusual because the definition of $A(m,n)$ involves an evaluation of A at arguments that may be a lot bigger than m and n. The definitions of f_2 above showed how definitions of function values at small argument values in terms of larger one can easily lead to nonterminating evaluations. The definition of the Ackermann function is actually ok, but proving this takes some ingenuity (see Problem 7.26).

7.4 Arithmetic Expressions

Expression evaluation is a key feature of programming languages, and recognition of expressions as a recursive data type is a key to understanding how they can be processed.

To illustrate this approach we'll work with a toy example: arithmetic expressions like $3x^2 + 2x + 1$ involving only one variable, "x." We'll refer to the data type of such expressions as Aexp. Here is its definition:

Definition 7.4.1.

- **Base cases**:

- The variable x is in Aexp.
- The arabic numeral $\mathtt{k}$ for any nonnegative integer k is in Aexp.

- **Constructor cases**: If $e, f \in$ Aexp, then
  - $[\, e + f \,] \in$ Aexp. The expression $[\, e + f \,]$ is called a *sum*. The Aexp's e and f are called the *components* of the sum; they're also called the *summands*.
  - $[\, e * f \,] \in$ Aexp. The expression $[\, e * f \,]$ is called a *product*. The Aexp's e and f are called the *components* of the product; they're also called the *multiplier* and *multiplicand*.
  - $-[\, e \,] \in$ Aexp. The expression $-[\, e \,]$ is called a *negative*.

Notice that Aexp's are fully bracketed, and exponents aren't allowed. So the Aexp version of the polynomial expression $3x^2+2x+1$ would officially be written as

$$[[\,3 * [\,x * x\,]\,] + [[\,2 * x\,] + 1\,]]\,. \tag{7.8}$$

These brackets and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2+2x+1$" instead of (7.8). But it's important to recognize that $3x^2+2x+1$ is not an Aexp; it's an *abbreviation* for an Aexp.

7.4.1 Evaluation and Substitution with Aexp's

Evaluating Aexp's

Since the only variable in an Aexp is x, the value of an Aexp is determined by the value of x. For example, if the value of x is 3, then the value of $3x^2 + 2x + 1$ is 34. In general, given any Aexp e and an integer value n for the variable x we can evaluate e to finds its value eval(e, n). It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 7.4.2. The *evaluation function*, eval : Aexp $\times\, \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions $e \in$ Aexp as follows. Let n be any integer.

- **Base cases**:

$$\text{eval}(x, n) ::= n \qquad \text{(value of variable } x \text{ is } n\text{)}, \tag{7.9}$$

$$\text{eval}(\mathtt{k}, n) ::= k \qquad \text{(value of numeral } \mathtt{k} \text{ is } k\text{, regardless of } x\text{.)} \tag{7.10}$$

- **Constructor cases**:

$$\text{eval}([\,e_1 + e_2\,], n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n), \tag{7.11}$$
$$\text{eval}([\,e_1 * e_2\,], n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n), \tag{7.12}$$
$$\text{eval}(-[\,e_1\,], n) ::= -\text{eval}(e_1, n). \tag{7.13}$$

For example, here's how the recursive definition of eval would arrive at the value of $3 + x^2$ when x is 2:

$$\begin{aligned}
\text{eval}([\,3 + [\,x * x\,]\,], 2) &= \text{eval}(3, 2) + \text{eval}([\,x * x\,], 2) && \text{(by Def 7.4.2.7.11)}\\
&= 3 + \text{eval}([\,x * x\,], 2) && \text{(by Def 7.4.2.7.10)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 7.4.2.7.12)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 7.4.2.7.9)}\\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard operation used by compilers and algebra systems. For example, the result of substituting the expression $3x$ for x in the expression $x(x-1)$ would be $3x(3x-1)$. We'll use the general notation $\text{subst}(f, e)$ for the result of substituting an Aexp f for each of the x's in an Aexp e. So as we just explained,

$$\text{subst}(3x, x(x-1)) = 3x(3x-1).$$

This substitution function has a simple recursive definition:

Definition 7.4.3. The *substitution function* from Aexp × Aexp to Aexp is defined recursively on expressions $e \in$ Aexp as follows. Let f be any Aexp.

- **Base cases**:

$$\text{subst}(f, x) ::= f \quad \text{(subbing } f \text{ for variable } x \text{ just gives } f\text{,)} \tag{7.14}$$
$$\text{subst}(f, \mathtt{k}) ::= \mathtt{k} \quad \text{(subbing into a numeral does nothing.)} \tag{7.15}$$

- **Constructor cases**:

$$\text{subst}(f, [\, e_1 + e_2 \,]) ::= [\, \text{subst}(f, e_1) + \text{subst}(f, e_2)] \tag{7.16}$$

$$\text{subst}(f, [\, e_1 * e_2 \,]) ::= [\, \text{subst}(f, e_1) * \text{subst}(f, e_2)] \tag{7.17}$$

$$\text{subst}(f, -[\, e_1 \,]) ::= -[\, \text{subst}(f, e_1)] \,. \tag{7.18}$$

Here's how the recursive definition of the substitution function would find the result of substituting $3x$ for x in the expression $x(x-1)$:

$$\begin{aligned}
&\text{subst}(3x, x(x-1)) \\
&= \text{subst}([\, 3 * x \,], [\, x * [\, x + -[\, 1]\,]\,]) && \text{(unabbreviating)} \\
&= [\, \text{subst}([\, 3 * x \,], x) * \\
&\qquad \text{subst}([\, 3 * x \,], [\, x + -[\, 1]\,])] && \text{(by Def 7.4.3 7.17)} \\
&= [\,[\, 3 * x \,] * \text{subst}([\, 3 * x \,], [\, x + -[\, 1]\,])] && \text{(by Def 7.4.3 7.14)} \\
&= [\,[\, 3 * x \,] * [\, \text{subst}([\, 3 * x \,], x) \\
&\qquad + \text{subst}([\, 3 * x \,], -[\, 1])]\,] && \text{(by Def 7.4.3 7.16)} \\
&= [\,[\, 3 * x \,] * [\,[\, 3 * x \,] + -[\, \text{subst}([\, 3 * x \,], 1)]\,]\,] && \text{(by Def 7.4.3 7.14 \& 7.18)} \\
&= [\,[\, 3 * x \,] * [\,[\, 3 * x \,] + -[\, 1]\,]\,] && \text{(by Def 7.4.3 7.15)} \\
&= 3x(3x-1) && \text{(abbreviation)}
\end{aligned}$$

Now suppose we have to find the value of $\text{subst}(3x, x(x-1))$ when $x = 2$. There are two approaches. First, we could actually do the substitution above to get $3x(3x-1)$, and then we could evaluate $3x(3x-1)$ when $x = 2$, that is, we could recursively calculate $\text{eval}(3x(3x-1), 2)$ to get the final value 30. This approach is described by the expression

$$\text{eval}(\text{subst}(3x, x(x-1)), 2). \tag{7.19}$$

In programming jargon, this would be called evaluation using the *Substitution Model*. With this approach, the formula $3x$ appears twice after substitution, so the multiplication $3 \cdot 2$ that computes its value gets performed twice.

The second approach is called evaluation using the *Environment Model*. Here, to compute the value of (7.19), we evaluate $3x$ when $x = 2$ using just 1 multiplication to get the value 6. Then we evaluate $x(x-1)$ when x has this value 6 to arrive at the value $6 \cdot 5 = 30$. This approach is described by the expression

$$\text{eval}(x(x-1), \text{eval}(3x, 2)). \tag{7.20}$$

The Environment Model only computes the value of $3x$ once, and so it requires one fewer multiplication than the Substitution model to compute (7.20).

This is a good place to stop and work this example out yourself (Problem 7.27).

The fact that the final integer values of (7.19) and (7.20) agree is no surprise. The substitution model and environment models will *always* produce the same final. We can prove this by structural induction directly following the definitions of the two approaches. More precisely, what we want to prove is

Theorem 7.4.4. *For all expressions e, $f \in Aexp$ and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{7.21}$$

Proof. The proof is by structural induction on e.[1]

Base cases:

- Case[x]

  The left-hand side of equation (7.21) equals $\text{eval}(f, n)$ by this base case in Definition 7.4.3 of the substitution function; the right-hand side also equals $\text{eval}(f, n)$ by this base case in Definition 7.4.2 of eval.

- Case[k].

  The left-hand side of equation (7.21) equals k by this base case in Definitions 7.4.3 and 7.4.2 of the substitution and evaluation functions. Likewise, the right-hand side equals k by two applications of this base case in the Definition 7.4.2 of eval.

Constructor cases:

- Case[$[\, e_1 + e_2 \,]$]

  By the structural induction hypothesis (7.21), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

  $$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{7.22}$$

  for $i = 1, 2$. We wish to prove that

  $$\text{eval}(\text{subst}(f, [\, e_1 + e_2 \,]), n) = \text{eval}([\, e_1 + e_2 \,], \text{eval}(f, n)). \tag{7.23}$$

  The left-hand side of (7.23) equals

  $$\text{eval}([\ \text{subst}(f, e_1) + \text{subst}(f, e_2)\,], n)$$

[1] This is an example of why it's useful to notify the reader what the induction variable is—in this case it isn't n.

by Definition 7.4.3.7.16 of substitution into a sum expression. But this equals

$$\text{eval}(\text{subst}(f, e_1), n) + \text{eval}(\text{subst}(f, e_2), n)$$

by Definition 7.4.2.(7.11) of eval for a sum expression. By induction hypothesis (7.22), this in turn equals

$$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}(f, n)).$$

Finally, this last expression equals the right-hand side of (7.23) by Definition 7.4.2.(7.11) of eval for a sum expression. This proves (7.23) in this case.

- Case[$[\, e_1 * e_2 \,]$] Similar.
- Case[$-[\, e_1 \,]$] Even easier.

This covers all the constructor cases, and so completes the proof by structural induction.

■

7.5 Games as a Recursive Data Type

Chess, Checkers, Go, and Nim are examples of *two-person games of perfect information*. These are games where two players, Player-1 and Player-2, alternate moves, and "perfect information" means that the situation at any point in the game is completely visible to both players. In Chess, for example, the visible positions of the pieces on the chess board completely determine how the rest of the game can be played by each player. By contrast, most card games are *not* games of perfect information because neither player can see the other's hand.

In the section we'll examine the *win-lose* two-person games of perfect information, WinLoseGm. We will define WinLoseGm as a recursive data type, and then we will prove, by structural induction, a fundamental theorem about winning strategies for these games. The idea behind the recursive definition is to recognize that the situation at any point during game play can itself be treated as the start of a new game. This is clearest for the game of Nim.

A Nim game starts with several piles of stones. A move in the game consists of removing some positive number of stones from a single pile. Player-1 and player-2 alternate making moves, and whoever takes the last stone wins. So if there is only one pile, then the first player to move wins by taking the whole pile.

A Nim game is defined by the sizes of the piles of stones at the start.

For example, let $\text{Nim}_{\langle 3,2\rangle}$ be the Nim game that starts with one size-three pile and one size-two pile. In this game, Player-1 has $2 + 3 = 5$ possible moves corresponding to removing one to three stones from the size-three pile, or one to two stones from the size-two pile. These five moves lead respectively to five new Nim games in which Player-2 will move first:;

$$\begin{array}{ll} \text{remove 1 from the size-three pile:} & \text{Nim}_{\langle 2,2\rangle}, \\ \text{remove 2 from the size-three pile:} & \text{Nim}_{\langle 2,1\rangle}, \\ \text{remove 3 from the size-three pile:} & \text{Nim}_{\langle 2\rangle}, \\ \text{remove 1 from the size-two pile:} & \text{Nim}_{\langle 1,3\rangle}, \\ \text{remove 2 from the size-two pile:} & \text{Nim}_{\langle 3\rangle}. \end{array}$$

Let's call these the *first-move* games $\text{Fmv}_{\langle 3,2\rangle}$ of $\text{Nim}_{\langle 3,2\rangle}$:

$$\text{Fmv}_{\langle 3,2\rangle} = \{\text{Nim}_{\langle 2,2\rangle}, \text{Nim}_{\langle 2,1\rangle}, \text{Nim}_{\langle 2\rangle}, \text{Nim}_{\langle 1,3\rangle}, \text{Nim}_{\langle 3\rangle}\}. \tag{7.24}$$

Now instead of saying Player-1's move is to pick some number of stones to remove from some pile, we can say that Player-1's move is to *select one of these first-move Nim games in which Player-2 will move first*. Abstractly we can *define* $\text{Nim}_{\langle 3,2\rangle}$ to be its set of first moves:

$$\text{Nim}_{\langle 3,2\rangle} ::= \text{Fmv}_{\langle 3,2\rangle}.$$

This is the big idea: we can *recursively define a Nim game to be a set of Nim games*. The base case will be the empty set of Nim games, corresponding to the Nim game with no piles; this is the game in which the player who moves first immediately loses.

Let's see how this works for $\text{Nim}_{\langle 3,2\rangle}$. Each of the first-move games listed in (7.24) abstractly is the same as its own set of first-move games. To take the simplest case, in the first-move game $\text{Nim}_{\langle 2\rangle}$, player-2 has two possible moves: he can remove one stone from the pile leaving $\text{Nim}_{\langle 1\rangle}$, or he can remove the whole pile leaving the game $\text{Nim}_{\langle\emptyset\rangle}$ with no piles.

$$\text{Fmv}_{\langle 2\rangle} = \{\text{Nim}_{\langle 1\rangle}, \text{Nim}_{\langle\emptyset\rangle}\}.$$

Since taking the whole pile leaves Player-1 with no move to make, that is,

$$\text{Nim}_{\langle\emptyset\rangle} ::= \text{Fmv}_{\langle\emptyset\rangle} = \emptyset,$$

this is a winning move for player-2.

But suppose player-2 chooses to remove only one stone. This move leaves the one-pile one-stone game $\text{Nim}_{\langle 1\rangle}$. Now player-1 has no choice but to choose the single first move in $\text{Fmv}_{\langle 1\rangle}$:

$$\text{Fmv}_{\langle 1\rangle} ::= \{\text{Nim}_{\langle\emptyset\rangle}\} = \{\emptyset\}.$$

That is, player-1 must choose the empty set and then wins, since player-2 is left with no move to make.

So abstractly, $\text{Nim}_{\langle 2\rangle}$ is just a *pure set made out of nothing*:

$$\text{Nim}_{\langle 2\rangle} = \{\{\emptyset\}, \emptyset\}.$$

Likewise, the whole $\text{Nim}_{\langle 3,2\rangle}$ game abstractly is another pure set:

$$\begin{array}{l}\{\\ \{\emptyset,\ \{\emptyset\}\},\\ \{\{\emptyset,\{\emptyset\}\},\ \{\{\emptyset\},\{\{\emptyset\}\},\{\emptyset,\{\emptyset\}\}\}\\ \{\emptyset,\ \{\emptyset\},\ \{\emptyset,\{\emptyset\}\}\}\\ \{\{\emptyset\},\ \{\{\emptyset\}\},\ \{\emptyset\{\emptyset\}\}\}\\ \{\{\emptyset\},\ \{\{\emptyset\}\},\ \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\},\ \{\{\emptyset\},\{\{\emptyset\}\},\{\emptyset,\{\emptyset\}\}\}\}\\ \}\end{array}$$

In fact, we could keep expanding any Nim game until all the numbers disappeared leaving only a set of sets of sets of... sets bottoming out at the empty set. This example hints at some of the connections between Nim games and pure set theory,[2] which we will explore a bit further in Section 8.3.3, Problem 8.42.

By the way, the winning move for player-1 in $\text{Nim}_{\langle 3,2\rangle}$ is to remove one stone from the size-three pile, leaving player-2 the first move in $\text{Nim}_{\langle 2,2\rangle}$. This is a winning move because the player who moves second in any game of two same-size piles can guarantee a win simply by mimicking the first player. For example, if the first player removes one stone from one pile, then the second player removes one stone from the other pile. It's worth thinking for a moment about why **the mimicking strategy guarantees a win** for the second player. More generally, there is an elegant procedure to figure out who has a winning strategy in any Nim game (Problem 7.36).

Nim games are special in that the two players are racing to the same goal, namely, leaving their opponent with no move. More generally, a two-person game can end in different ways, some of which count as wins for player-1 and others as losses—that is, wins for player-2. With this idea in mind, we now give the formal definition of win-lose games.

Definition 7.5.1. The class WinLoseGm of two-person win-lose games of perfect information is defined recursively as follows:

Base case: **P1Win** and **P2Win** are WinLoseGm's.

[2] Insert ref to Conway's *nimbers*.

Constructor case: If G is a nonempty set of WinLoseGm's, then G is a WinLoseGm game. Each game $M \in G$ is called a possible *first move* of G.

A *play* of a WinLoseGm game is a sequence of moves that ends with a win or loss for the first player, or goes on forever without arriving at an outcome.[3] More formally:

Definition. A *play* of a WinLoseGm game G and its outcome is defined recursively on the definition of WinLoseGm:

Base case: ($G = \mathbf{P1Win}$). The sequence $\langle \mathbf{P1Win} \rangle$ of length one is a *play* of G. Its *outcome* is a *win for player-1*.

Base case: ($G = \mathbf{P2Win}$). The sequence $\langle \mathbf{P2Win} \rangle$ of length one is a *play* of G. Its *outcome* is a *loss for player-1*, or equivalently, a *win for player-2*.

Constructor case: (G is a nonempty set of WinLoseGm's). A *play* of G is a sequence that starts with G followed by a play P_M of some game $M \in G$. The *outcome* of the play, if any, is the outcome of P_M.

The basic rules of some games do allow plays that go on forever. In Chess for example, a player might just keep moving the same piece back and forth, and if his opponent did the same, the play could go on forever.[4] But the recursive definition of WinLoseGm games actually rules out the possibility of infinite play.

Lemma 7.5.2. *Every play of a game $G \in$ WinLoseGm has an outcome.*

Proof. We prove Lemma 7.5.2 by structural induction, using the statement of the Lemma as the induction hypothesis.

Base case: ($G = \mathbf{P1Win}$). There is only one play of G, namely the length one play $\langle \mathbf{P1Win} \rangle$, whose outcome is a win for player-1.

Base case: ($G = \mathbf{P2Win}$). Likewise with the outcome being a win for player-2.

Constructor case: (G is a nonempty set of WinLoseGm's). A play of G by definition consists G followed by a play P_M for some $M \in G$. By structural induction, P_M must be a sequence of some finite length n that ends with an outcome. So this play of G is a length $n + 1$ sequence that finishes with the same outcome. ■

[3] In English, "Nim game" might refer to the rules that define the game, but it might also refer to a particular play of the game—as in the once famous third game in the 1961 movie *Last Year at Marienbad*. It's usually easy to figure out which way the phrase is being used, and we won't worry about it.

[4] Real chess tournaments rule this out by setting an advance limit on the number of moves, or by forbidding repetitions of the same position more than twice.

Among the games of Checkers, Chess, Go and Nim, only Nim is genuinely a win-lose game. The other games might end in a tie/draw/stalemate/jigo rather than a win or loss. But the results about win-lose games will apply to games with ties if we simply treat a tie as a loss for player-1.

In all the games like this that we're familiar with, there are only a finite number of possible first moves and a bound on the possible length of play. It's worth noting that the definition of WinLoseGm does not require this. In fact, since finiteness is not needed to prove any of the results below, it would arguably be misleading to assume it.

This observation that games might be infinite illustrates the surprising power of structural induction. The proof shows that even in infinite games, every play must have an outcome. This is *not* the same as saying that the length of play has a *fixed bound*. For example, there is a game whose first move allows, for every nonempty integer n, selection of a game that can played for exactly n steps. Even though every play in such a game has an outcome, there are plays of every finite length.

Infinite games are a little far-fetched for Computer Science applications, but they actually play an important role in the study of set theory and logic. A lovely feature of structural induction is that it implicitly applies to infinite structures even if you didn't notice it.

7.5.1 Game Strategies

A *strategy* for a player is a rule that tells the player which move to make whenever it is their turn. More precisely, a strategy s is a function from games to games with the property that $s(G) \in G$ for all games G. A pair of strategies for the two players determines exactly which moves the players choose, and so it determines a unique play of the game, depending on who moves first.

A key question about a game is what strategy will ensure that a player will win. Each Player wants a strategy whose outcome is guaranteed to be a win for themselves.

7.5.2 Fundamental Theorem for Win-Lose Games

The Fundamental Theorem for WinLoseGm games says that one of the players always has a fixed "winning" strategy that guarantees a win against *every possible* opponent strategy.

Thinking about Chess for instance, this seems surprising. Serious chess players are typically secretive about their intended play strategies, believing that an opponent could take advantage of knowing their strategy. Their concern seems to be that for any strategy they choose, their opponent could tailor a strategy to beat it.

But the Fundamental Theorem says otherwise. In theory, in any win-lose-tie

game like Chess or Checkers, each of the players will have a strategy that guarantees a win or a stalemate, even if the strategy is known to their opponent. That is,

- there is winning strategy for one of the players, or
- both players have strategies that guarantee them at worst a draw.

Even though the Fundamental Theorem reveals a profound fact about games, it has a very simple proof by structural induction.

Theorem 7.5.3. *[Fundamental Theorem for Win-Lose Games] For any WinLoseGm game G, one of the players has a winning strategy.*

Proof. The proof is by structural induction on the definition of a $G \in$ WinLoseGm. The induction hypothesis is that one of the players has a winning strategy for G.

Base case: ($G =$ **P1Win** or **P2Win**). Then there is only one possible strategy for each player, namely, do nothing and finish with outcome G.

Constructor case: (G is a nonempty set of WinLoseGm's). By structural induction we may assume that for each $M \in G$ one of the players has a winning strategy. Notice that since players alternate moves, the first player in G becomes the second player in M.

Now if there is a move $M_0 \in G$ where the second player in M_0 has a winning strategy, then the first player in G has a simple winning strategy: pick M_0 as the first move, and then follow the second player's winning strategy for M_0.

On the other hand, if no $M \in G$ has a winning strategy for the second player in M, then we can conclude by induction that every $M \in G$ has a winning strategy for the first player in M. Now the second player in G has a simple winning strategy, namely if the first player in G makes the move M, then the second player in G should follow the follow the winning strategy for the first player in M. ■

Notice that although Theorem 7.5.3 guarantees a winning strategy, its proof gives no clue which player has it. For the Subset Takeaway Game of Problem 4.7 and most familiar 2PerGm's like Chess, Go, ..., no one knows which player has a winning strategy.[5]

[5] Checkers used to be in this list, but it's now been shown that each player has a strategy that forces a tie (reference TBA).

Infinite Games [Optional]

Suppose we play a tournament of n chess games for some positive integer n. This tournament will be a WinLoseGm if we agree on a rule for combining the payoffs of the n individual chess games into a final payoff for the whole tournament.

An n game tournament still has a bound on the length of play, namely, n times the maximum length of one game. But now suppose we define a *meta-chess-tournament*, whose first move is a choice of any positive integer n, after which we play an n-game tournament. Now the meta-chess-tournament is an infinite game—it has an infinite number of first moves.

Only the first move in the meta-chess-tournament is infinite. If we set up a tournament consisting of n meta-chess-tournaments, this would be a game with n infinite sets of possible second moves. Then we oould go to meta-meta tournaments,.... Weird as these meta games may seem, the Fundamental Theorem still applies: in each of these games, one of the players will have winning strategy.

7.6 Search Trees

Searching through data may be the most common of all computations, and *search trees* are a widely used data structure that supports efficient search.

The basic idea behind search trees is simple. Assuming the data to be searched have some kind of order relation on them—like numerical order for numbers or alphabetical order for words—the data is laid out in a kind of branching "tree" structure illustrated in Figure 7.1. In this example, there are two branches at each branch point, with all the values in the left branch less than the value at the branch point, and all the values in the right branch greater than the value at the branch point. This makes it easy to search for a given value in the tree: starting at the topmost branchpoint, compare the given value to the value at the branchpoint. If the given value—call it g—is less than the value at the branchpoint—call it b—then continue searching in the left branch; if g is greater than b continue searching in the right branch; if $g = b$ then the given value has been found, and the search ends successfully. Finally, if there are no more branches to search, then g is not there, so the search fails.

For example, to search for 3.7 in the tree in Figure 7.1, we compare 3.7 to the value 6.8 at the topmost branch. Since $3.7 < 6.8$, we go left and compare 3.7 to the new top value π. Since $3.7 > \pi$, go right comparing 3.7 to 4.9, go left again, and arrive at 3.7. So the search ends successfully.

Organizing the data in this way means you can search for a value without having

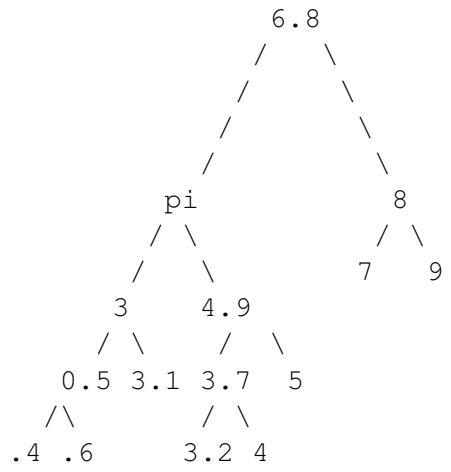


Figure 7.1 A binary search tree.

to scan through all the values in the data set. In fact, if the all the top-to-bottom paths through the tree don't differ too much in length, then with maximum path length n, the search tree will hold up to 2^n items. Since each search simply follows some path, this means that searching for a value will take *exponentially less time* than it would to search through all the data. That's why search trees are a big win.

Of course it is possible that the tree has paths that are not short. An extreme example is illustrated in Figure 7.2 where there is a path containing half the values. For this tree, the average search for a value requires searching halfway down the path, and the exponential savings evaporates.

By treating search trees as a recursive data type, we will be able to describe simple recursive procedures for tree management, and we also will have a powerful inductive proof method for proving properties of these trees and procedures. This will allow concise formal definitions of search trees whose top-to-bottom paths are close to the same length, and precise definitions of procedures for maintaining this property as the data evolves through adding and removing values.

Although we will end up with an elegant and efficient algorithm for tree search and insertion, the importance of this section is its illustration of simple recursive definitions for a basic data type and functions on the data. So there are a lot of mathematical definitions in this section, because you can't have a mathematical proof about things that haven't been mathematically defined. Once these definitions are in place, we can proceed to prove data properties and procedure correctness using the basic method for recursive data, namely, structural induction.

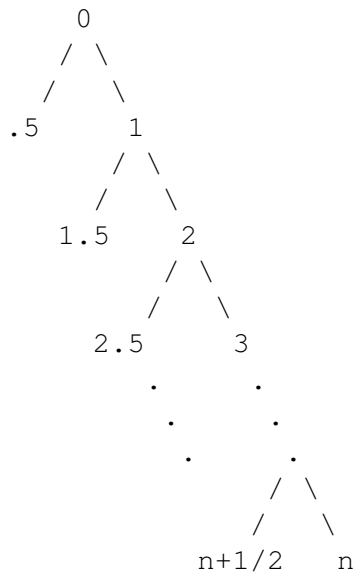


Figure 7.2 A very unbalanced search tree.

7.6.1 Binary Branching

The trees in the examples above have either a single "leaf" with no branching, or a binary (two-way) branch to the left and right. Allowing more than two branches is important in some applications, but binary branching is most common and is enough to illustrate all the important ideas.

There are lots of ways to represent trees as a computer data structure—or a mathematical set—but we will take an abstract approach that will enable us, largely, to understand binary trees without worrying about how they are represented.

We can begin by assuming only that we have some set BBTr of things that represent *binary branching trees*. A crucial property of a binary branching tree is whether or not it is a leaf. In any representation, there has to be way to test this. So there has to be a *leaf predicate* leaf? that does the job. That is, a binary branching tree $T \in \text{BBTr}$ is a leaf when leaf?(T) is true. Let Leaves be the set of leaves:

$$\text{Leaves} ::= \{T \in \text{BBTr} \mid \text{leaf?}(T)\}.$$

The non-leaf trees are supposed to be the trees that actually have left and right branches. In any representation of trees there has to be a way to select these branches, that is, there have to be *selector functions* left and right that produce the left and right branches of non-leaf elements of BBTr. We formalize this by defining

Branching ::= BBTr − Leaves and stipulating that that there are total functions

$$\text{left} : \text{Branching} \to \text{BBTr},$$
$$\text{right} : \text{Branching} \to \text{BBTr}.$$

So by definition, the left branch of T is the tree left(T) and the right branch is right(T).

Subtrees

In the example search trees we started with, if you start at the top and follow some path down, you arrive at a subtree that is also a search tree. Any such path can be described by the sequence of left and right branches you select. For example in the tree given in Figure 7.1, the sequence

$$(\text{right}, \text{left}, \text{left}). \tag{7.25}$$

describes a length-three path that starts at the top tree $T_{6.8}$ with label 6.8. The selectors are applied in right-to-left order, so we first go to left$T_{6.8}$ which is the subtree T_π labelled π. Then we take the left branch to the subtree leftT_π which is the subtree T_3 labelled 3, and take a final right branch to end with the subtree rightT_3 labelled 3.1. That is, the subtree at the end of the path given by the sequence (7.25) is

$$\text{right}(\text{left}(\text{left}T_{6.8})) = T_{3.1}.$$

More generally, let $\vec{P}$ be a finite sequence of selector functions and suppose $T \in$ BBTr. The notation for the subtree at the end of the path described by $\vec{P}$ is subtree$_T(\vec{P})$. For example, we just saw that

$$\text{subtree}_{T_{6.8}}((\text{right}, \text{left}, \text{left})) = T_{3.1}.$$

Notice that if the path $\vec{P}$ tries to continue past a leaf of T, there won't be any subtree at the end. For example, subtree$_{T_{6.8}}$(left, right, right) is undefined since subtree$_{T_{6.8}}$(right, right) is the leaf labelled with 9 that has no left branch.

There's a technicality worth noting here that can be confusing. When people talk about trees they normally mean the whole tree structure of the kind shown in Figures 7.1 and 7.2 above. In this section, that's *not* what "tree" means. Instead, a tree is just a single point or node, namely the "root" node at the top. The rest of the nodes below the root are by themselves trees that are called the *subtrees* of the tree.

To formalize this, we will give a definition of subtree$_T(\vec{P})$ by induction on path length. We'll use the notation

$$f \cdot \vec{P}$$

for the sequence that starts with f followed by the successive elements of $\vec{P}$. For example, if $\vec{P}$ is the length three sequence in (7.25), then

$$\text{right} \cdot \vec{P}$$

is the length four sequence

$$(\text{right}, \text{right}, \text{left}, \text{left}).$$

Likewise for $\vec{P} \cdot f$, so for example

$$\vec{P} \cdot \text{right}$$

would be the length four sequence

$$(\text{right}, \text{left}, \text{left}, \text{right}).$$

Definition 7.6.1. The tree $\text{subtree}_T(\vec{P}) \in \text{BBTr}$ will be defined by induction on the length of the sequence $\vec{P}$:

Base case: ($\vec{P} = \lambda$). Then

$$\text{subtree}_T(\vec{P}) ::= T.$$

Constructor case: ($\vec{P} = f \cdot \vec{Q}$). If $\text{subtree}_T(\vec{Q})$ is defined and equals a tree in Branching, then

$$\text{subtree}_T(\vec{P}) ::= f(\text{subtree}_T(\vec{Q})).$$

Otherwise $\text{subtree}_T(\vec{P})$ is undefined.[6]

Let $\text{Subtrs}(T)$ be all the trees you can get to by following paths starting at T. More precisely,

Definition 7.6.2.

$$\text{Subtrs}(T) ::= \{S \in \text{BBTr} \mid S = \text{subtree}_T(\vec{P}) \text{ for some selector sequence } \vec{P}\}.$$

[6] An alternative Constructor case could have been

Constructor case: ($\vec{P} = \vec{Q} \cdot f$). If $T \in$ Branching, then

$$\text{subtree}_T(\vec{P}) ::= \text{subtree}_{f(T)}(\vec{Q}).$$

The *proper subtrees* PropSubtrs(T) are the subtrees that are actually "below" T.

$$\text{PropSubtrs}(T) ::= \{S \in \text{BBTr} \mid S = \text{subtree}_T(\vec{P}) \text{ for some selector sequence } \vec{P} \neq \lambda\}.$$

So we have by definition that

$$\text{Subtrs}(T) = \text{PropSubtrs}(T) \cup \{T\}. \tag{7.26}$$

There are a couple of obvious facts about paths and subtrees that we might just take for granted but are worth saying explicitly. The first is that a proper subtree of T must be a subtree of its left subtree or its right subtree:

Corollary 7.6.3. *For $T \in$ Leaves,*

$$\mathit{PropSubtrs}(T) = \emptyset.$$

For $T \in$ Branching,

$$\mathit{PropSubtrs}(T) = \mathit{Subtrs}(\mathit{left}(T)) \cup \mathit{Subtrs}(\mathit{right}(T)).$$

Therefore,

$$\mathit{Subtrs}(T) = \{T\} \cup \mathit{Subtrs}(\mathit{left}(T)) \cup \mathit{Subtrs}(\mathit{right}(T)). \tag{7.27}$$

Corollary 7.6.3 follows from the fact when the first step in a path goes to a subtree, the rest of the path is in that subtree.

The second obvious fact is that a subtree of a subtree is a subtree:

Corollary 7.6.4. *If $S \in \mathit{PropSubtrs}(T)$ and $R \in \mathit{Subtrs}(S)$, then $R \in \mathit{PropSubtrs}(T)$.*

Corollary 7.6.4 follows from the fact that if $\vec{P}$ is a path in T to a subtree S of T, and $\vec{Q}$ is a path in S to a subtree R of S, as in Figure 7.3, then the concatenation $\vec{Q} \cdot \vec{P}$ is a path from T to R.[7]

7.6.2 Binary Trees

Although we've called the elements of BBTr binary branching "trees," the abstract way we defined BBTr allows some weird stuff to happen. One weirdness is that nothing forbids "circular" trees that are proper subtrees of themselves. And even if we forbid circular trees, there are still infinite trees. For example, suppose we choose BBTr to be the nonnegative integers and define $\text{left}(n) ::= 2n + 1$ and $\text{right}(n) ::= 2n + 2$. Now we get the infinite binary tree indicated in Figure 7.4.

[7] Path sequences are applied right-to-left, so $\vec{Q} \cdot \vec{P}$ is the path that starts with $\vec{P}$ and then follows $\vec{Q}$.

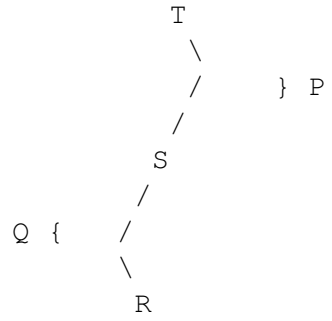


Figure 7.3 A subbranch S of T and a subbranch R of S.

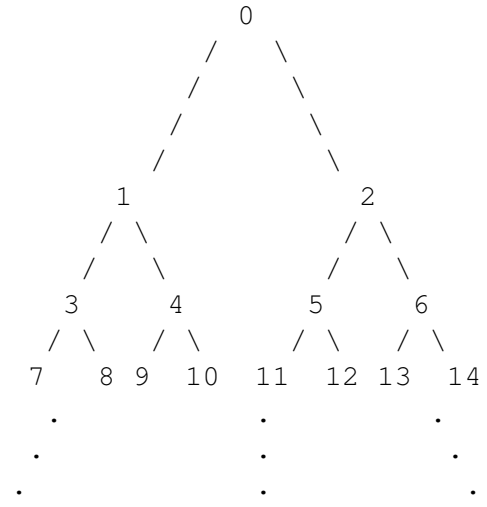


Figure 7.4 An infinite tree with no leaves.

The problem with both circular structures and infinite ones is that they will have infinite paths that you might have to keep searching down without ever finishing. By forbidding infinite paths we ensure that searches will end, and we also stop all the circular and infinite weirdness.

Paths that result in subtrees are by definition finite, so we should be clear about the definition of an infinite path.

Definition 7.6.5. A infinite path in $T \in \text{BBTr}$ is an infinite sequence

$$\ldots, f_n, \ldots, f_1, f_0$$

of selector functions such that

$$\text{subtree}_T(\vec{P}) \in \text{Branching}$$

for every finite suffix

$$\vec{P} = (f_n, \ldots, f_1, f_0).$$

But there's still a problem. Even if they have no infinite paths, a tree in BBTr may have some unexpected *sharing* of subtrees. For example, it is possible to have a tree in BBTr whose left and right subtrees are the same.

Definition 7.6.6. A tree $T \in \text{BBTr}$ *shares subtrees* if there are two finite sequences $\vec{P} \neq \vec{Q}$ of selector functions such that $\text{subtree}_T(\vec{P})$ and $\text{subtree}_T(\vec{Q})$ are both defined, and

$$\text{subtree}_T(\vec{P}) = \text{subtree}_T(\vec{Q}).$$

Sharing of subtrees can be very useful in getting a compact representation of a tree, but it seriously complicates adding or removing values from search trees, so we will forbid sharing as well. This leads us to FinTr, the familiar trees that underlie search trees.

Definition 7.6.7.

$$\text{FinTr} ::= \{T \in \text{BBTr} \mid T \text{ has no infinite path and does not share subtrees}\}.$$

There is also a simple way to define trees recursively.

Definition 7.6.8. The set RecTr of *recursive trees* is defined recursively. For $T \in \text{BBTr}$,

Base case: ($T \in \text{Leaves}$). $T \in \text{RecTr}$.

Constructor case: ($T \in \text{Branching}$). If $\text{left}(T), \text{right}(T) \in \text{RecTr}$, and these two trees have no subtree in common, that is,

$$\text{Subtrs}(\text{left}(T)) \cap \text{Subtrs}(\text{right}(T)) = \emptyset,$$

then T is in RecTr.

Let's verify first of all that recursive trees are finite.

Definition 7.6.9. The *size* of a tree $T \in \text{BBTr}$ is the number of subtrees it has:

$$\text{size}(T) ::= |\text{Subtrs}(T)|. \tag{7.28}$$

Corollary 7.6.10. *Every recursive tree has finite size.*

Proof. The proof is by structural induction on the definition of $T \in \text{RecTr}$.

Base case: ($T \in$ Leaves). The only subtree of T is itself, so

$$\text{size}(T) = 1 < \infty.$$

Constructor case: ($T \in$ Branching). In this case, $L ::= \text{left}(T)$ and $R ::= \text{right}(T)$ are defined.

By induction hypothesis, we may assume that the sizes of L and R are finite, so by (7.27),

$$\text{size}(T) \leq 1 + \text{size}(L) + \text{size}(R) \tag{7.29}$$

is also finite.[8] ■

Now it is an initially surprising, very useful, and—we think—very cool fact that the trees with no infinite paths and no shared subtrees are actually the *same* as the recursively defined trees. It takes a bit of ingenuity to prove this.

Theorem 7.6.11. *(Fundamental Theorem for Recursive Trees)*

$$\mathit{FinTr} = \mathit{RecTr}.$$

We first prove

$$\text{RecTr} \subseteq \text{FinTr}, \tag{7.30}$$

that is, no recursive tree T has an infinite path or a shared subtree.

Proof. The proof follows by structural induction on the definition of RecTr:

Base case: ($T \in$ Leaves). T has no infinite path or shared subtree because it has no proper subtrees at all.

Constructor case: ($T \in$ Branching). In this case, left(T) and right(T) are defined and in RecTr. By induction hypothesis, neither left(T) nor right(T) has an infinite path nor sharing of subtrees.

[8] The inequality (7.29) is actually an equality, see (7.34) below.

But if T had an infinite path

$$\ldots, f_n, \ldots, f_2, f_1, f_0,$$

then

$$\ldots, f_n, \ldots, f_2, f_1 \tag{7.31}$$

would be an infinite path in $f_0(T)$, contradicting the induction hypothesis. So T cannot have an infinite path.

Next we show that T has no shared subtree.

Since T has no infinite path, T cannot be a subtree of left(T) or right(T), and so will not itself be a shared subtree. Since there are no shared subtrees within left(T) or right(T), the only remaining possibility would be a subtree that was in both left(T) and right(T), which is not possible by definition of RecTr. ■

Second, we prove

$$\text{FinTr} \subseteq \text{RecTr}. \tag{7.32}$$

Taking the contrapositive, if $T \in$ BBTr is *not* a recursive tree, we want to prove the claim that it has an infinite path or it has a shared subtree.

Proof. If $T \notin$ RecTr, then T cannot be a leaf. Now if left(T) and right(T) have a subtree in common, then this subtree is shared in T, proving the claim. So we may assume that left(T) and right(T) have no subtree in common. In this case, if both left(T) and right(T) were in RecTr, then T would be in RecTr by definition, contradicting the hypothesis that $T \notin$ RecTr. Therefore left(T) and right(T) cannot both be in RecTr.

Let f_0 be a selector such that $f_0(T) \notin$ RecTr. Now if left($f_0(T)$) and right($f_0(T)$) have a shared subtree, then, since a subtree of a subtree is a subtree (Corollary 7.6.4), this will also be a shared subtree of T, proving the claim. Therefore, left($f_0(T)$) and right($f_0(T)$) cannot both be in RecTr. Let f_1 be a selector such that $f_1(f_0(T)) \notin$ RecTr. Similarly, if there is no sharing or infinite path in $f_1(f_0(T))$, then there must be a selector f_2 such that $f_2(f_1(f_0(T))) \notin$ RecTr. Continuing in this way, we either find a shared subtree of T or we find an infinite path

$$\ldots, f_n, \ldots, f_2, f_1, f_0$$

in T. ■

7.6.3 Properties of Recursive Trees

With the recursive definition of trees, we can define some important basic functions on trees recursively. We start with a recursive definition of size:

Definition 7.6.12. We define a function recsize : RecTr $\to \mathbb{Z}^+$ by induction on the definition of RecTr:

Base case: ($T \in$ Leaves).

$$\text{recsize}(T) ::= 1.$$

Constructor case: ($T \in$ Branching).

$$\text{recsize}(T) ::= 1 + \text{recsize}(\text{left}(T)) + \text{recsize}(\text{right}(T)).$$

A simple proof by structural induction confirms that the definition of recsize is correct, namely:

Lemma 7.6.13.

$$\mathit{recsize}(T) = \text{size}(T)$$

for all $T \in$ RecTr.

Proof. The proof is by structural induction on the definition of RecTr.

Base case: ($T \in$ Leaves). By Corollary 7.6.3, $\text{Subtrs}(T) = \{T\}$, so

$$\text{recsize}(T) ::= 1 = |\text{Subtrs}(T)| = \text{size}(T).$$

Constructor case: ($T \in$ Branching).

We have by induction hypothesis that

$$\text{size}(T) = \text{recsize}(f(T)), \qquad \text{for } f \in \{\text{left}, \text{right}\}. \tag{7.33}$$

We have from 7.27 that

$$\text{Subtrs}(T) = \{T\} \cup \text{Subtrs}(\text{left}(T)) \cup \text{Subtrs}(\text{right}(T)),$$

for $T \in$ BBTr. Now if $T \in$ RecTr, then these three sets are disjoint, so

$$\text{size}(T) = 1 + \text{size}(T) + \text{size}(T). \tag{7.34}$$

Now we have

$$\begin{aligned} \text{size}(T) &= 1 + \text{recsize}(\text{left}(T)) + \text{recsize}(\text{right}(T)) && \text{(by (7.33) and (7.34))} \\ &= \text{recsize}(T) && \text{(def. of recsize).} \end{aligned}$$

■

Similarly, the *depth* of a recursive tree is the length of its longest path. The formal definition is:

$$\text{depth}(T) ::= \max\{|\vec{P}| \mid \text{subtree}_T(\vec{P}) \text{ is a leaf}\}. \tag{7.35}$$

Here is the recursive definition:

Definition 7.6.14. Base case: ($T \in$ Leaves).

$$\text{recdepth}(T) ::= 0.$$

Constructor case: ($T \in$ Branching).

$$\text{recdepth}(T) ::= 1 + \max\{\text{recdepth}(\text{left}(T)), \text{recdepth}(\text{right}(T))\}.$$

Again, it follows directly by structural induction (Problem 7.44) that

Lemma 7.6.15.

$$\textit{recdepth}(T) = \text{depth}(T). \tag{7.36}$$

7.6.4 Depth versus Size

Arranging values in a search tree gives an efficient way to do repeated searches. In this case, the longest search will be the length of the longest path to a leaf, that is, the depth of the tree. So a first question to address is how short can we guarantee searches will be compared to the size of the set to be searched? The simple answer is that if the set to be searched has n values in it, then some searches will unavoidably take $\log_2 n$ value comparisons; moreover, there is a way to arrange the values so that no search takes any longer than this minimum amount. The following two Theorems confirm this.

We first prove a basic relation between size and depth of recursive trees:

Theorem 7.6.16. *For all $T \in$ RecTr,*

$$\text{size}(T) \leq 2^{\text{depth}(T)+1} - 1. \tag{7.37}$$

There is an easy way to understand and prove this inequality: if there were paths of different lengths that ended at leaves, then there would be a bigger tree with the same depth—just attach two new leaves to the leaf at the end of the shorter path. So the biggest tree with depth d will be the *complete tree* T in which all paths have depth d. In this tree there are two paths of length one that branch into four paths of length two, continuing into 2^d paths of length d. So

$$\text{size}(T) = 2^0 + 2^1 + 2^2 + \cdots + 2^d = 2^{d+1} - 1.$$

There is also a proof by structural induction. The structural induction provides little insight into what's going on and involves some tedious manipulation of exponents. But it goes through routinely with no need for ingenuity, and it can be checked by a proof checker—a person or a computer program—with no insight into trees.

Proof. It is enough to prove Theorem 7.6.16 for recsize and recdepth. The proof is by structural induction on the definition of RecTr:

Base case: ($T \in$ Leaves).

$$\text{recsize}(T) ::= 1 = 2^{0+1} - 1 = 2^{\text{recdepth}(T)+1} - 1.$$

Constructor case: ($T \in$ Branching). By definition,

$$\text{recsize}(T) = 1 + \text{recsize}(\text{left}(T)) + \text{recsize}(\text{right}(T)). \tag{7.38}$$

Let

$$\begin{aligned} d_l &::= \text{recdepth}(\text{left}(T)), \\ d_r &::= \text{recdepth}(\text{right}(T)). \end{aligned}$$

Then by induction hypothesis, the right hand side of (7.38) is

$$\begin{aligned} &\leq 1 + (2^{d_l+1} - 1) + (2^{d_r+1} - 1) \\ &\leq 1 + (2^{\max\{d_l,d_r\}+1} - 1) + (2^{\max\{d_l,d_r\}+1} - 1) \\ &= 2 \cdot 2^{\max\{d_l,d_r\}+1} - 1 \\ &= 2 \cdot 2^{\text{recdepth}(T)} - 1 && \text{(def of recdepth}(T)) \\ &= 2^{\text{recdepth}(T)+1} - 1. \end{aligned}$$

■

Taking $\log_2$ of both sides of the inequality (7.37) confirms that the length of searches has to be at least one less than the log of the size of the data set to be searched:

Corollary 7.6.17. *For any recursive tree $T \in$ RecTr,*

$$\text{depth}(T) \geq \log_2(\text{size}(T)) - 1. \tag{7.39}$$

On the other hand, searches need not be any longer than $\log_2$ of the size of the set of values. Namely, define a tree to be *fully balanced* when all the lengths of paths that end at leaves differ by at most one.

Theorem 7.6.18. *If $B \in RecTr$ is fully balanced with depth at least one, then*

$$\text{size}(B) \geq 2^{\text{depth}(B)} + 1.$$

Proof. The smallest fully balanced tree with depth $d \geq 1$ will be a complete tree of depth $d-1$ along with two leaves of depth d, so its size will be $(2^d - 1) + 2 = 2^d + 1$. ■

Corollary 7.6.19. *For any odd positive integer n, there is a recursive binary tree—namely a fully balanced tree of size n—that has depth at most $\log_2 n$.*

7.6.5 AVL Trees

As a set of values evolves by the insertion or deletion of values, maintaining a fully balanced tree may require changing the branches of all the subtrees—an effort that would swamp the time saved by having short searches. A "have your cake and eat it too" approach is to loosen the fully balanced requirement while still maintaining logarithmic search length.

In *AVL trees* the fully balanced property is relaxed so that left and right branch depths are allowed to differ by one everywhere.

Definition 7.6.20. A tree $T \in \text{RecTr}$ is an *AVL tree* when

$$|\text{depth}(\text{left}(S)) - \text{depth}(\text{right}(S))| \leq 1$$

for all nonleaf subtrees $S \in \text{Subtrs}(T)$.

Lemma 7.6.21. *If T is AVL, then*

$$\text{size}(T) \geq \phi^{\text{depth}(T)}, \tag{7.40}$$

where ϕ is the "golden ratio"

$$\frac{1+\sqrt{5}}{2} \geq 1.6.$$

Proof. Let's prove by strong induction on depth d the induction hypothesis

$$P(d) ::= [T \in \text{RecTr AND recdepth}(T) = d] \text{ IMPLIES size}(T) \geq a^d,$$

for some $a \in \mathbb{R}$ which we will derive in the course of the proof.

Base case: ($d = 0$). In this case $T \in \text{Leaves}$, so

$$\text{recsize}(T) ::= 1 = a^0,$$

proving $P(0)$.

Inductive step: Suppose T is an AVL tree of depth $d+1$, and let L and R be the left and right subtrees of T. Then the depth of at least one of L and R must be d and the depth of the other must be at least $d-1$, by definition of AVL tree. By strong induction hypothesis, the size of one of L and R must be least a^d and the size of the other at least a^{d-1}. So

$$\text{recsize}(T) = 1 + \text{recsize}(L) + \text{recsize}(R) > a^d + a^{d-1}.$$

Now $P(d+1)$ will follow providing

$$a^d + a^{d-1} \geq a^{d+1},$$

which means

$$1 + a \geq a^2. \tag{7.41}$$

Using the quadratic formula to solve (7.41) for equality, we find

$$a = \frac{-(-1) + \sqrt{(-1)^2 - 4 \cdot 1 \cdot (-1)}}{2} = \frac{1+\sqrt{5}}{2} = \phi$$

satisfies (7.41), completing the proof. ■

Taking $\log_2$ of both sides of (7.40), we conclude

Corollary 7.6.22. *If T is an AVL tree, then*

$$\text{depth}(T) \leq \frac{\log_2(\text{size}(T))}{(\log_2 \phi)} \leq 1.44 \log_2(\text{size}(T)).$$

In other words, searches in AVL trees at worst will take 1.44 times the optimal search time, and they will still be exponentially smaller than the size of the data set.

7.6.6 Number Labels

We described the basic idea of search trees at the beginning of this section and used it to explain why the connection between tree size and depth is important. Now we'll define Search trees formally as a recursive data type. This will allow us to define search functions recursively and verify their properties by structural induction.

Search trees have a numerical label on each subtree. Abstractly, this means there is a total function

$$\text{num} : \text{BBTr} \to \mathbb{R}.$$

Let $\min(T)$ be the minimum label in T and likewise for $\max(T)$. More formally,

Definition 7.6.23.

$$\text{nums}(T) ::= \{\text{num}(S) \mid S \in \text{Subtrs}(T)\}, \tag{7.42}$$

$$\min(T) ::= \min \text{nums}(T), \tag{7.43}$$

$$\max(T) ::= \max \text{nums}(T). \tag{7.44}$$

The min and max will exist for any finite tree T.

Definition 7.6.24. The Search trees $T \in \text{BBTr}$ are defined recursively as follows:

Base case: ($T \in$ Leaves). T is a Search tree.

Constructor case: ($T \in$ Branching). If $\text{left}(T)$ and $\text{right}(T)$ are both Search trees, and

$$\max(\text{left}(T)) < \text{num}(T) < \min(\text{right}(T)),$$

then T is a Search tree.

Since Search trees by definition can't have subtrees with the same label, they have no shared subtrees. So every Search tree is a recursive tree in RecTr.

The simple procedure for finding a number in a Search tree explained at the beginning of this section can now be described as a recursive function srch. For any Search tree T and number r, the value of $\text{srch}(T, r)$ will be the path in T that leads to r, assuming r appears in T. Otherwise the value will be a path to a leaf followed by **fail**.

Definition 7.6.25. The function

$$\text{srch} : ((\text{Search trees}) \times \mathbb{R}) \to \left(\{\text{left}, \text{right}\}^* \cup (\{\textbf{fail}\} \cdot \{\text{left}, \text{right}\}^*)\right)$$

is defined as follows:

If $r = \text{num}(T)$, then

$$\text{srch}(T, r) ::= \lambda.$$

If $r \neq \text{num}(T)$, then $\text{srch}(T, r)$ is defined recursively on Definition 7.6.24 of Search tree:

Base case: ($T \in$ Leaves).

$$\text{srch}(T, r) ::= \textbf{fail}.$$

Constructor case: ($T \in$ Branching).

$$\text{srch}(T, r) ::= \begin{cases} \text{srch}(\text{left}(T), r) \cdot \text{left} & \text{if } r < \text{num}(T), \\ \text{srch}(\text{right}(T), r) \cdot \text{right} & \text{if } r > \text{num}(T). \end{cases}$$

We can now rigorously prove that the srch procedure always gives the correct answer.

Theorem 7.6.26. *If $r \in nums(T)$, then $srch(T, r)$ is a sequence of selectors and*

$$num(subtree_T(srch(T, r))) = r.$$

*Otherwise, $srch(T, r)$ will be a sequence of the form **fail**$\cdot \vec{P}$ for some sequence $\vec{P}$.*

Theorem 7.6.26 can be proved by a straightforward structural induction on the recursive Definition 7.6.24 of Search Tree. We'll leave it as an exercise.[9]

7.6.7 Insertions into AVL trees

The procedure for inserting a value r into an AVL tree T can be defined recursively on the definition of AVL tree in a way that parallels the search procedure. If r is already a value in T, the search will discover this, and the insert procedure doesn't have to do anything. If r is not a value in T, then it has to be inserted recursively into one of the branches of T depending on how it compares to num(T). More precisely, suppose A and B are the branches of T, that is, $\{A, B\} = \{\text{left}(T), \text{right}(T)\}$, and A is the subtree into which r will be inserted. The result of inserting r into A will be a new AVL tree S with the same labels as A along with r. That is,

$$\text{nums}(S) = \{r\} \cup \text{nums}(A).$$

Moreover, the depth of S will be at most one larger and no smaller that the depth of A.

Now we let N be a new tree the same label as T, with one branch of N equal to S and the other branch of N equal to the B as shown in Figure 7.5. This makes N a Search tree whose branches are AVL trees and whose labels are the labels of T along with r. Now if the depth of S is at most one greater than the depth of B, then N will be the desired AVL tree.

The only problem is that the depth of S may now be two greater than the depth of the B. The fix is to apply a "rotation operation" that rearranges the top few subtrees to balance depths.

To explain rotation, we need to identify some subtrees of S and that will get rearranged. Let $d ::= \text{depth}(B)$. We know that $\text{depth}(S) = d + 2$. This means S has branches R, U whose depths are either d or $d+1$ with at least one of them with depth $d + 1$. Let U be the branch furthest from the middle, as shown in Figure 7.5.

A simple case of rotation is when $\text{depth}(U) = d+1$. Here we define the rotation of N to be the tree X shown in Figure 7.6.

[9] At this early stage of practicing structural induction, the full proof really belongs in the text. We'll include it when we get a chance.

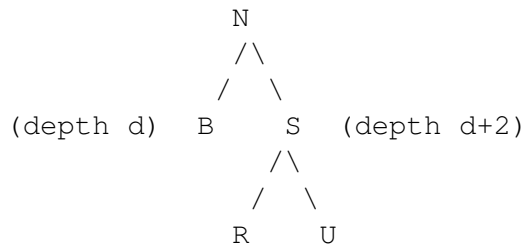


Figure 7.5 The Tree N before rotation.

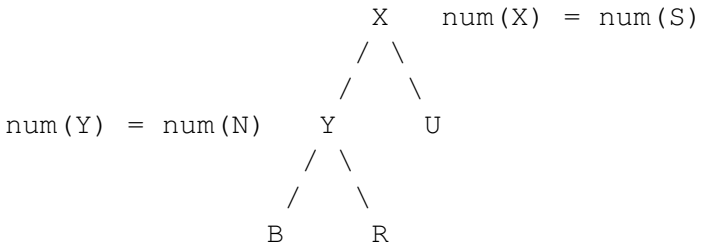


Figure 7.6 X is the rotation of tree N in Figure 7.5.

Here $X \in \text{RecTr}$ is a "new" tree—that is, $X \notin \text{Subtrs}(N)$—with the same label as S, and Y is a new tree with the same label as N. So $\text{nums}(X) = \text{nums}(N)$. Also, depending on whether $\text{depth}(R)$ is d or $d+1$, $\text{depth}(Y)$ is $d+1$ or $d+2$, which is within one of $\text{depth}(U)$, and $\text{depth}(X)$ is $d+2$ or $d+3$ which is within one of $\text{depth}(T)$. So the depths of subtrees of X and Y are properly balanced according to the AVL rule. It is easy to check that X is also a search tree, which makes it an AVL search tree. Moreover, X has very nearly the same subtrees as N. In particular,

$$\text{Subtrs}(X) = \{X, Y\} \cup (\text{Subtrs}(N) - \{N, S\}).$$

There is another rearrangement of subtrees that handles the case when $\text{depth}(U) = d$ which we leave as an exercise (Problem 7.40). As a consequence, we have:

Lemma 7.6.27. *(Rotation) There is a function*

$$\textit{rotate} : (\textit{(AVL Trees)} \times \mathbb{R} \times \textit{(AVL Trees)}) \to \textit{(AVL Trees)},$$

such that if B and S are AVL trees, $r \in \mathbb{R}$, $\text{depth}(S) = \text{depth}(B) + 2$, and either

$$\max(B) < r < \min(S), \quad \textit{or} \quad \max(S) < r < \min(B),$$

then

$$nums(rotate(B, r, S)) = nums(B) \cup \{r\} \cup nums(S),$$

$$\text{depth}\,(S) \leq \text{depth}\,(rotate(B, r, S)) \leq \text{depth}\,(S) + 1,$$

and there are two size-three sets S_{new}, S_{old} of AVL trees, such that

$$Subtrs(rotate(B, r, S)) = (Subtrs(B) \cup Subtrs(S) \cup S_{new}) - S_{old}.$$

In defining the insert function, it turns out to be crucial to allow *leaves* to be labelled with one *or two* values, with both values obeying the labelling rules for search trees. (This could be viewed as a relaxation of binary branching at the leaf level, so the "father" of a leaf need have only one proper subtree instead of two.) Having this extra label at the leaves serves as a buffer that allows comprehensive relabelling to be deferred. Without such a buffer, a single insertion and deletion could force a relabelling of every subtree (Problem 7.42).

Definition 7.6.28. The function

$$\text{insert} : (\text{AVL trees}) \times \mathbb{R} \to (\text{AVL trees})$$

is defined as follows:

If $r = \text{num}(T)$, or if $T \in$ Leaves with two labels and $r \in \text{num}(T)$, then

$$\text{insert}(T, r) ::= T,$$

Otherwise $\text{insert}(T, r)$ is defined recursively on Definition 7.6.24 of Search tree:

Base case: ($T \in$ Leaves).

subcase: ($T \in$ Leaves with one label). Then $\text{insert}(T, r)$ will be a leaf labelled with the set of two values $\{r, \text{num}(T)\}$.

subcase: ($T \in$ Leaves with two labels) Then $\text{insert}(T, r)$ is defined to be a depth one Search tree whose labels are the three values $\{r\} \cup \text{num}(T)$.

Constructor case: ($T \in$ Branching).

Then $\text{insert}(T, r)$ is defined by cases.

subcase: ($r > \text{num}(T)$). Let $S ::= \text{insert}(\text{right}(T), r)$ and $B ::= \text{left}(T)$.

Let N be a "new" tree, that is,

$$N \notin \text{Subtrs}(S) \cup \text{Subtrs}(B),$$

chosen to satisfy

$$\text{num}(N) = \text{num}(T), \tag{7.45}$$
$$\text{right}(N) = S, \tag{7.46}$$
$$\text{left}(N) = B. \tag{7.47}$$

(See Figure 7.5.)

subsubcase: $(\text{depth}(S) \leq \text{depth}(B) + 1.)$

$$\text{insert}(T, r) ::= N.$$

subsubcase: $(\text{depth}(S) = \text{depth}(B) + 2.)$

$$\text{insert}(T, r) ::= \text{rotate}(B, r, S).$$

subcase: $(r < \text{num}(T))$. Same as the case $(r > \text{num}(T))$ with left and right reversed.

Here is the formal theorem that shows that $\text{insert}(T, r)$ is correct. Moreover, the subtrees of T and $\text{insert}(T, r)$ are the same except for at most three times the search length subtrees, reflecting the fact that the time for an insertion, like the time for a search, is exponentially smaller than the size of the data set.

Theorem 7.6.29. *Suppose T is an AVL tree and $r \in \mathbb{R}$. Then $insert(T, r)$ is an AVL tree, and*

$$nums(insert(T, r)) = \{r\} \cup nums(T),$$

$$\text{depth}(T) \leq \text{depth}(insert(T, r)) \leq \text{depth}(T) + 1,$$

and there are two size-$(3 \cdot 1.44 \log_2 \text{size}(T))$ sets S_{new}, S_{old} of AVL trees, such that

$$Subtrs(insert(T)) = (Subtrs(T) \cup S_{new}) - S_{old}.$$

The same approach leads to a procedure for deleting a value from an AVL tree (Problem 7.43).

7.7 Induction in Computer Science

Induction is a powerful and widely applicable proof technique, which is why we've devoted two entire chapters to it. Strong induction and its special case of ordinary

induction are applicable to any kind of thing with nonnegative integer sizes—which is an awful lot of things, including all step-by-step computational processes.

Structural induction then goes beyond number counting, and offers a simple, natural approach to proving things about recursive data types and recursive computation.

In many cases, a nonnegative integer size can be defined for a recursively defined datum, such as the length of a string, or the number of subtrees in a tree. This allows proofs by ordinary induction. But assigning sizes to recursively defined data often produces more cumbersome proofs than structural induction.

In fact, structural induction is theoretically more powerful than ordinary induction. However, it's only more powerful when it comes to reasoning about infinite data types—like infinite trees, for example—so this greater power doesn't matter in practice. What does matter is that for recursively defined data types, structural induction is a simple and natural approach. This makes it a technique every computer scientist should embrace.

Problems for Section 7.1

Practice Problems

Problem 7.1.
The set OBT of *Ordered Binary Trees* is defined recursively as follows:

Base case: $\langle$**leaf**$\rangle$ is an OBT, and

Constructor case: if R and S are OBT's, then $\langle$**node**, $R, S\rangle$ is an OBT.

If T is an OBT, let n_T be the number of **node** labels in T and l_T be the number of **leaf** labels in T.

Prove by structural induction that for all $T \in$ OBT,

$$l_T = n_T + 1.$$

Class Problems

Problem 7.2.
Prove by structural induction on the recursive definition(7.1.1) of A^* that concatenation is *associative*:

$$(r \cdot s) \cdot t = r \cdot (s \cdot t) \tag{7.48}$$

for all strings $r, s, t \in A^*$.

Problem 7.3.
The *reversal* of a string is the string written backwards, for example, $\text{rev}(abcde) = edcba$.

(a) Give a simple recursive definition of $\text{rev}(s)$ based on the recursive definitions 7.1.1 of $s \in A^*$ and of the concatenation operation 7.1.3.

(b) Prove that

$$\text{rev}(s \cdot t) = \text{rev}(t) \cdot \text{rev}(s), \tag{7.49}$$

for all strings $s, t \in A^*$. You may assume that concatenation is associative:

$$(r \cdot s) \cdot t = r \cdot (s \cdot t)$$

for all strings $r, s, t \in A^*$ (Problem 7.2).

Problem 7.4.
The Elementary 18.01 Functions (F18's) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function $\text{id}(x) ::= x$ is an F18,
- any constant function is an F18,
- the sine function is an F18,

Constructor cases:
If f, g are F18's, then so are

1. $f + g$, fg, 2^g,
2. the inverse function f^{-1},
3. the composition $f \circ g$.

(a) Prove that the function $1/x$ is an F18.

Warning: Don't confuse $1/x = x^{-1}$ with the inverse id^{-1} of the identity function $\text{id}(x)$. The inverse id^{-1} is equal to id.

(b) Prove by Structural Induction on this definition that the Elementary 18.01 Functions are *closed under taking derivatives*. That is, show that if $f(x)$ is an F18, then so is $f' ::= df/dx$. (Just work out 2 or 3 of the most interesting constructor cases; you may skip the less interesting ones.)

Problem 7.5.

Here is a simple recursive definition of the set E of even integers:

Definition. Base case: $0 \in E$.

Constructor cases: If $n \in E$, then so are $n + 2$ and $-n$.

Provide similar simple recursive definitions of the following sets:

(a) The set $S ::= \{2^k 3^m 5^n \in \mathbb{N} \mid k, m, n \in \mathbb{N}\}$.

(b) The set $T ::= \{2^k 3^{2k+m} 5^{m+n} \in \mathbb{N} \mid k, m, n \in \mathbb{N}\}$.

(c) The set $L ::= \{(a, b) \in \mathbb{Z}^2 \mid (a - b)$ is a multiple of $3\}$.

Let L' be the set defined by the recursive definition you gave for L in the previous part. Now if you did it right, then $L' = L$, but maybe you made a mistake. So let's check that you got the definition right.

(d) Prove by structural induction on your definition of L' that

$$L' \subseteq L.$$

(e) Confirm that you got the definition right by proving that

$$L \subseteq L'.$$

(f) (*Optional*) See if you can give an *unambiguous* recursive definition of L.

Problem 7.6.

Definition. The recursive data type binary-2PG of *binary trees* with leaf labels L is defined recursively as follows:

- **Base case:** $\langle \texttt{leaf}, l \rangle \in$ binary-2PG, for all labels $l \in L$.
- **Constructor case:** If $G_1, G_2 \in$ binary-2PG, then

$$\langle \texttt{bintree}, G_1, G_2 \rangle \in \text{binary-2PG}.$$

The *size* $|G|$ of $G \in$ binary-2PG is defined recursively on this definition by:

- **Base case:**
$$|\langle \texttt{leaf}, l\rangle| ::= 1, \quad \text{for all } l \in L.$$
- **Constructor case:**
$$|\langle \texttt{bintree}, G_1, G_2\rangle| ::= |G_1| + |G_2| + 1.$$

For example, the size of the binary-2PG G pictured in Figure 7.7, is 7.

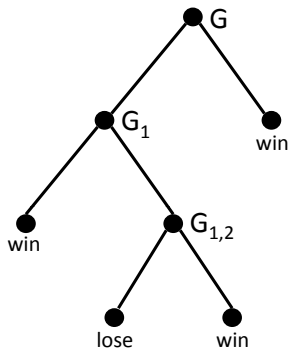


Figure 7.7 A picture of a binary tree G.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PG G pictured in Figure 7.7.

The value of flatten(G) for $G \in$ binary-2PG is the sequence of labels in L of the leaves of G. For example, for the binary-2PG G pictured in Figure 7.7,

$$\text{flatten}(G) = (\texttt{win}, \texttt{lose}, \texttt{win}, \texttt{win}).$$

(b) Give a recursive definition of flatten. (You may use the operation of *concatenation* (append) of two sequences.)

(c) Prove by structural induction on the definitions of flatten and size that

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{7.50}$$

Homework Problems

Problem 7.7.
The string *reversal* function, rev : $A^* \to A^*$ has a simple recursive definition.

Base case: $\text{rev}(\lambda) ::= \lambda$.

Constructor case: $\text{rev}(as) ::= \text{rev}(s)a$ for $s \in A^*$ and $a \in A$.

A string s is a *palindrome* when $\text{rev}(s) = s$. The *palindromes* also have a simple recursive definition as the set RecPal.

Base cases: $\lambda \in \text{RecPal}$ and $a \in \text{RecPal}$ for $a \in A$.

Constructor case: If $s \in \text{RecPal}$, then $asa \in \text{RecPal}$ for $a \in A$.

Verifying that the two definitions agree offers a nice exercise in structural induction and also induction on length of strings. The verification rests on three basic properties of concatenation and reversal proved in separate problems 7.2 and 7.3.

Fact.

$$(rs = uv \text{ AND } |r| = |u|) \text{ IFF } (r = u \text{ AND } s = v) \tag{7.51}$$

$$r \cdot (s \cdot t) = (r \cdot s) \cdot t \tag{7.52}$$

$$\text{rev}(st) = \text{rev}(t)\,\text{rev}(s) \tag{7.53}$$

(a) Prove that $s = \text{rev}(s)$ for all $s \in \text{RecPal}$.

(b) Prove conversely that if $s = \text{rev}(s)$, then $s \in \text{RecPal}$.

Hint: By induction on $n = |s|$.

Problem 7.8.
Let m, n be integers, not both zero. Define a set of integers, $L_{m,n}$, recursively as follows:

- **Base cases**: $m, n \in L_{m,n}$.
- **Constructor cases**: If $j, k \in L_{m,n}$, then
  1. $-j \in L_{m,n}$,
  2. $j + k \in L_{m,n}$.

Let L be an abbreviation for $L_{m,n}$ in the rest of this problem.

(a) Prove *by structural induction* that every common divisor of m and n also divides every member of L.

(b) Prove that any integer multiple of an element of L is also in L.

(c) Show that if $j, k \in L$ and $k \neq 0$, then $\text{rem}(j,\ k) \in L$.

(d) Show that there is a positive integer $g \in L$ that divides every member of L. *Hint:* The least positive integer in L.

(e) Conclude that g from part (d) is $\gcd(m, n)$, the greatest common divisor, of m and n.

Problem 7.9.

Definition. Define the number $\#_c(s)$ of occurrences of the character $c \in A$ in the string s recursively on the definition of $s \in A^*$:

base case: $\#_c(\lambda) ::= 0$.

constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

Prove by structural induction that for all $s, t \in A^*$ and $c \in A$

$$\#_c(s \cdot t) = \#_c(s) + \#_c(t).$$

Problem 7.10.

Fractals are an example of mathematical objects that can be defined recursively. In this problem, we consider the Koch snowflake. Any Koch snowflake can be constructed by the following recursive definition.

- **Base case**: An equilateral triangle with a positive integer side length is a Koch snowflake.
- **Constructor case**: Let K be a Koch snowflake, and let l be a single edge of the snowflake. Remove the middle third of l, and replace it with two line segments of the same length as the middle third, as shown in Figure 7.8

  The resulting figure is also a Koch snowflake.

Figure 7.8 Constructing the Koch Snowflake.

(a) Find a single Koch snowflake that has exactly 9 edges and includes at least 3 different edge lengths.

(b) Prove using structural induction that the area inside any Koch snowflake is of the form $q\sqrt{3}$, where q is a rational number. Be sure to clearly label your induction hypothesis and other necessary assumptions during your proof.

Hint: If you require other facts about Koch snowflakes, be sure to prove those by structural induction too.

Problem 7.11.
The set RBT of *Red-Black Trees* is defined recursively as follows:

Base cases:

- $\langle$**red**$\rangle \in$ RBT, and
- $\langle$**black**$\rangle \in$ RBT.

Constructor cases: A, B are RBT's, then

- if A, B start with **black**, then $\langle$**red**, A, $B\rangle$ is an RBT.
- if A, B start with **red**, then $\langle$**black**, A, $B\rangle$ is an RBT.

For any RBT T, let

- r_T be the number of **red** labels in T,
- b_T be the number of **black** labels in T, and
- $n_T ::= r_T + b_T$ be the total number of labels in T.

Prove that

$$\text{If } T \text{ starts with a } \textbf{red} \text{ label, then } \frac{n_T}{3} \leq r_T \leq \frac{2n_T + 1}{3}, \tag{7.54}$$

Hint:

$$n/3 \leq r \quad \text{IFF} \quad (2/3)n \geq n - r$$

Exam Problems

Problem 7.12.
The Arithmetic Trig Functions (*Atrig*'s) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function $\mathrm{id}(x) ::= x$ is an *Atrig*,
- any constant function is an *Atrig*,
- the sine function is an *Atrig*,

Constructor cases:
If f, g are *Atrig*'s, then so are

1. $f + g$
2. $f \cdot g$
3. the composition $f \circ g$.

Prove by structural induction on this definition that if $f(x)$ is an *Atrig*, then so is $f' ::= df/dx$.

Problem 7.13.

Definition. The set RAF of *rational functions* of one real variable is the set of functions defined recursively as follows:

Base cases:

- The identity function, $\mathrm{id}(r) ::= r$ for $r \in \mathbb{R}$ (the real numbers), is an RAF,
- any constant function on $\mathbb{R}$ is an RAF.

Constructor cases: If f, g are RAF's, then so is $f \circledast g$, where $\circledast$ is one of the operations

1. addition $+$,
2. multiplication $\cdot$ or

3. division /.

(a) Describe how to construct functions $e, f, g \in \text{RAF}$ such that

$$e \circ (f + g) \neq (e \circ f) + (e \circ g). \tag{7.55}$$

(b) Prove that for all real-valued functions e, f, g (not just those in RAF):

$$(e \circledast f) \circ g = (e \circ g) \circledast (f \circ g), \tag{7.56}$$

Hint: $(e \circledast f)(x) ::= e(x) \circledast f(x)$.

(c) Let predicate $P(h)$ be the following predicate on functions $h \in \text{RAF}$:

$$P(h) ::= \forall g \in \text{RAF}.\ h \circ g \in \text{RAF}.$$

Prove by structural induction on the definition of RAF that $P(h)$ holds for all $h \in$ RAF.

Make sure to indicate explicitly

- each of the base cases, and
- each of the constructor cases.

Problem 7.14.

The *2-3-averaged numbers* are a subset, N23, of the real interval $[0, 1]$ defined recursively as follows:

Base cases: $0, 1 \in \text{N23}$.

Constructor case: If a, b are in N23, then so is $L(a, b)$ where

$$L(a, b) ::= \frac{2a + 3b}{5}.$$

(a) Use ordinary induction or the Well-Ordering Principle to prove that

$$\left(\frac{3}{5}\right)^n \in \text{N23}$$

for all nonnegative integers n.

(b) Prove by Structural Induction that the product of two 2-3-averaged numbers is also a 2-3-averaged number.

Hint: Prove by structural induction on c that, if $d \in \text{N23}$, then $cd \in \text{N23}$.

Problem 7.15.
This problem is about binary strings $s \in \{0,1\}^*$.

Let's call a recursive definition of a set of strings cat-OK when all its constructors are defined as concatenations of strings.[10]

For example, the set, One1, of strings with exactly one `1` has the cat-OK definition:

Base case: The length-one string `1` is in One1.

Constructor case: If s is in One1, then so is `0`s and s`0`.

(a) Give a cat-OK definition of the set E of even length strings consisting solely of `0`'s.

(b) Let rev(s) be the reversal of the string s. For example, rev(`001`) = `100`. A *palindrome* is a string s such that s = rev(s). For example, `11011` and `010010` are palindromes.

Give a cat-OK definition of the *palindromes*.

(c) Give a cat-OK definition of the set P of strings consisting solely of `0`'s whose length is a power of two.

Problems for Section 7.2

Practice Problems

Problem 7.16.
Define the sets F_1 and F_2 recursively:

- F_1:
  - $5 \in F_1$,
  - if $n \in F_1$, then $5n \in F_1$.
- F_2:
  - $5 \in F_2$,
  - if $n, m \in F_1$, then $nm \in F_2$.

[10] The concatenation of two strings x and y, written xy, is the string obtained by appending x to the left end of y. For example, the concatenation of 01 and 101 is 01101.

(a) Show that one of these definitions is technically *ambiguous*. (Remember that "ambiguous recursive definition" has a technical mathematical meaning which does not imply that the ambiguous definition is unclear.)

(b) Briefly explain what advantage unambiguous recursive definitions have over ambiguous ones.

(c) A way to prove that $F_1 = F_2$, is to show first that $F_1 \subseteq F_2$ and second that $F_2 \subseteq F_1$. One of these containments follows easily by structural induction. Which one? What would be the induction hypothesis? (You do not need to complete a proof.)

Problem 7.17.

The set RecMatch of strings of matched brackets is defined recursively in Definition 7.2.1, and an alternative definition as the set AmbRecMatch is given in 7.2.4.

Fill in the following parts of a proof by structural induction that

$$\text{RecMatch} \subseteq \text{AmbRecMatch}. \tag{*}$$

(As a matter of fact, AmbRecMatch = RecMatch, though we won't prove this here—see Problem 7.20.)

(a) State an **induction hypothesis** suitable for proving (*) by structural induction.

(b) State and prove the **base case**.

(c) Prove the **inductive step**.

An advantage of the RecMatch definition is that it is *unambiguous*, while the definition of AmbRecMatch is ambiguous.

(d) Give an example of a string whose membership in AmbRecMatch is ambiguously derived.

(e) Briefly explain what advantage unambiguous recursive definitions have over ambiguous ones. (Remember that "ambiguous recursive definition" has a technical mathematical meaning which does not imply that the ambiguous definition is unclear.)

Problem 7.18.

Let RecMatch be the set of strings of matched brackets of Definition 7.2.1

Prove that RecMatch is closed under string concatenation. Namely, if s and t are strings in RecMatch, then $s \cdot t \in$ RecMatch.

Class Problems

Problem 7.19.

Let p be the string [ ]. A string of brackets is said to be *erasable* iff it can be reduced to the empty string by repeatedly erasing occurrences of p. For example, to erase the string

$$[[[]][]][],$$

start by erasing the three occurrences of p to obtain

$$[[]].$$

Then erase the single occurrence of p to obtain,

$$[],$$

which can now be erased to obtain the empty string λ.

On the other hand the string

$$[]][[[[]]] \tag{7.57}$$

is not erasable, because when we try to erase, we get stuck. Namely, start by erasing the two occurrences of p in (7.57) to obtain

$$][[[[].$$

The erase the one remaining occurrence of p to obtain.

$$][[[.$$

At this point we are stuck with no remaining occurrences of p. [11]

Let Erasable be the set of erasable strings of brackets. Let RecMatch be the recursive data type of strings of *matched* brackets given in Definition 7.2.1

(a) Use structural induction to prove that

$$\text{RecMatch} \subseteq \text{Erasable}.$$

(b) Supply the missing parts (labeled by "(*)") of the following proof that

$$\text{Erasable} \subseteq \text{RecMatch}.$$

[11] Notice that there are many ways to erase a string, depending on when and which occurrences of p are chosen to be erased. It turns out that given any initial string, the final string reached after performing all possible erasures will be the same, no matter how erasures are performed. We take this for granted here, although it is not altogether obvious. (See Problem 6.28 for a proof).

Proof. We prove by strong induction that every length n string in Erasable is also in RecMatch. The induction hypothesis is

$$P(n) ::= \forall x \in \text{Erasable}.\ |x| = n \text{ IMPLIES } x \in \text{RecMatch}.$$

Base case:

(*) What is the base case? Prove that P is true in this case.

Inductive step: To prove $P(n+1)$, suppose $|x| = n+1$ and $x \in$ Erasable. We need to show that $x \in$ RecMatch.

Let's say that a string y is an *erase* of a string z iff y is the result of erasing a *single* occurrence of p in z.

Since $x \in$ Erasable and has positive length, there must be an erase, $y \in$ Erasable, of x. So $|y| = n - 1 \geq 0$, and since $y \in$ Erasable, we may assume by induction hypothesis that $y \in$ RecMatch.

Now we argue by cases:

Case (y is the empty string):

(*) Prove that $x \in$ RecMatch in this case.

Case ($y = [\,s\,]\,t$ for some strings $s, t \in$ RecMatch): Now we argue by subcases.

- **Subcase**($x = py$):
  (*) Prove that $x \in$ RecMatch in this subcase.
- **Subcase** (x is of the form $[\,s'\,]\,t$ where s is an erase of s'):
  Since $s \in$ RecMatch, it is erasable by part (b), which implies that $s' \in$ Erasable. But $|s'| < |x|$, so by induction hypothesis, we may assume that $s' \in$ RecMatch. This shows that x is the result of the constructor step of RecMatch, and therefore $x \in$ RecMatch.
- **Subcase** (x is of the form $[\,s\,]\,t'$ where t is an erase of t'):
  (*) Prove that $x \in$ RecMatch in this subcase.

(*) Explain why the above cases are sufficient.

This completes the proof by strong induction on n, so we conclude that $P(n)$ holds for all $n \in \mathbb{N}$. Therefore $x \in$ RecMatch for every string $x \in$ Erasable. That is, Erasable $\subseteq$ RecMatch. Combined with part (a), we conclude that

$$\text{Erasable} = \text{RecMatch}.$$

■

Problem 7.20.
We prove that two recursive definitions of strings of matched brackets, RecMatch of Definition 7.2.1 and AmbRecMatch of Definition 7.2.4 define the same set.

(a) Prove that the set RecMatch is closed under string concatenation. Namely, $\forall s, t \in \text{RecMatch}.\ s \cdot t \in \text{RecMatch}$.

(b) Prove by structural induction that AmbRecMatch $\subseteq$ RecMatch.

(c) Prove that RecMatch $=$ AmbRecMatch.

Homework Problems

Problem 7.21.
One way to determine if a string has matching brackets, that is, if it is in the set, RecMatch, of Definition 7.2.1 is to start with 0 and read the string from left to right, adding 1 to the count for each left bracket and subtracting 1 from the count for each right bracket. For example, here are the counts for two sample strings:

[	]	]	[	[	[	[	[	]	]	]	]
0	1	0	−1	0	1	2	3	4	3	2	1	0

[	[	[	]	]	[	]	]	[	]
0	1	2	3	2	1	2	1	0	1	0

A string has a *good count* if its running count never goes negative and ends with 0. So the second string above has a good count, but the first one does not because its count went negative at the third step. Let

$$\text{GoodCount} ::= \{s \in \{\text{]}, \text{[}\}^* \mid s \text{ has a good count}\}.$$

The empty string has a length 0 running count we'll take as a good count by convention, that is, $\lambda \in \text{GoodCount}$. The matched strings can now be characterized precisely as this set of strings with good counts.

(a) Prove that GoodCount contains RecMatch by structural induction on the definition of RecMatch.

(b) Conversely, prove that RecMatch contains GoodCount.

Hint: By induction on the length of strings in GoodCount. Consider when the running count equals 0 for the second time.

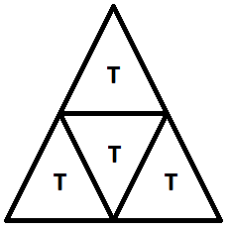


Figure 7.9 DET T' from Four Copies of DET T

Problem 7.22.
Divided Equilateral Triangles (DETs) were defined in Problem 5.10 as follows:

- **Base case:** An equilateral triangle with unit length sides is a DET whose only subtriangle is itself.
- If $T ::=$  is a DET, then the equilateral triangle T' built out of four copies of T as shown in Figure 7.9 is also a DET, and the subtriangles of T' are exactly the subtriangles of each of the copies of T.

Note that "subtriangles" correspond to *unit* subtriangles as illustrated in Figure 7.10. The four large triangles from which a new triangle is made are *not* subtriangles (unless the side length is two).

Properties of DETs were proved earlier by induction on the length of a side of the triangle. Recognizing that the definition of DETs is recursive, we can instead prove properties of DETs by structural induction.

(a) Prove by structural induction that a DET with one of its corner subtriangles removed can be tiled with trapezoids built out of three subtriangles as in Figure 7.11.

(b) Explain why a DET with a middle triangle removed from one side can also be tiled by trapezoids. (If the side length is even, there are two "middle" subtriangles on each side. The middle subtriangles on the left hand edge are colored red in Figure 7.10.)

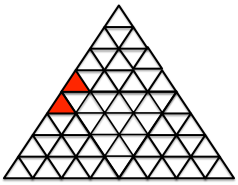


Figure 7.10 Subtriangles and "middle" subtriangles

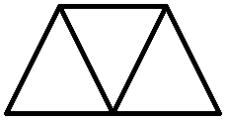


Figure 7.11 Trapezoid from Three Triangles

(c) In tiling a large square using L-shaped blocks as described in Section 5.1.5, there was a tiling with any single subsquare removed. Part (b) indicates that trapezoid-tilings are possible for DETs with a non-corner subtriangle removed, so it's natural to make the mistaken guess that DETs have a corresponding property:

False Claim. *A DET with any single subtriangle removed can be trapezoid-tiled.*

We can try to prove the claim by structural induction as in part (a).

Bogus proof. The claim holds vacuously in the base case of a DET with a single subtriangle.

Now let T' be a DET made of four copies of a DET T, and suppose we remove an arbitrary subtriangle from T'.

The removed subtriangle must be a subtriangle of one of the copies of T. The copies are the same, so for definiteness we assume the subtriangle was removed from copy 1. Then by structural induction hypothesis, copy 1 can be trapezoid-tiled, and then the other three copies of T can be trapezoid-tiled exactly as in the solution to part(a). This yields a complete trapezoid-tiling of T' with the arbitrary subtriangle removed.

We conclude by structural induction that any DET with any subtriangle removed can be trapezoid-tiled. ■

What's wrong with the proof?

Hint: Find a counter-example, and show where the proof breaks down.

(We don't know if there is a simple characterization of exactly which subtriangles can be removed to allow a trapezoid tiling.)

Problem 7.23.

A *binary word* is a finite sequence of `0`'s and `1`'s. In this problem, we'll simply call them "words." For example, $(1, 1, 0)$ and (1) are words of length three and one, respectively. We usually omit the parentheses and commas in the descriptions of words, so the preceding binary words would just be written as `110` and `1`.

The basic operation of placing one word immediately after another is called *concatentation*. For example, the concatentation of `110` and `1` is `1101`, and the concatentation of `110` with itself is `110110`.

We can extend this basic operation on words to an operation on *sets* of words. To emphasize the distinction between a word and a set of words, from now on we'll refer to a set of words as a *language*. Now if R and S are languages, then $R \cdot S$ is the language consisting of all the words you can get by concatenating a word from R with a word from S. That is,

$$R \cdot S ::= \{rs \mid r \in R \text{ AND } s \in S\}.$$

For example,

$$\{0, 00\} \cdot \{00, 000\} = \{000, 0000, 00000\}$$

Another example is $D \cdot D$, abbreviated as D^2, where $D ::= \{1, 0\}$.

$$D^2 = \{00, 01, 10, 11\}.$$

In other words, D^2 is the language consisting of all the length-two words. More generally, D^n will be the language of length-n words.

If S is a language, the language you can get by concatenating any number of copies of words in S is called S^*—pronounced "S star." (By convention, the empty word λ always included in S^*.) For example, $\{0, 11\}^*$ is the language consisting of all the words you can make by stringing together `0`'s and `11`'s. This language could also be described as consisting of the words whose blocks of `1`'s are always of even length. Another example is $(D^2)^*$, which consists of all the even length words. Finally, the language B of *all* binary words is just D^*.

The *Concatenation-Definable* (*C-D*) languages are defined recursively:

- **Base case**: Every finite language is a C-D.
- **Constructor cases**: If L and M are C-D's, then

$$L \cdot M, \quad L \cup M, \quad \text{and } \overline{L}$$

are C-D's.

Note that the *-operation is *not* allowed. For this reason, the C-D languages are also called the "star-free languages," [36].

Lots of interesting languages turn out to be concatenation-definable, but some very simple languages are not. This problem ends with the conclusion that the language $\{00\}^*$ of even length words whose bits are all 0's is not a C-D language.

(a) Show that the set B of all binary words is C-D. *Hint:* The empty set is finite.

Now a more interesting example of a C-D set is the language of all binary words that include three consecutive 1's:

$$B111B.$$

Notice that the proper expression here is "$B \cdot \{111\} \cdot B$." But it causes no confusion and helps readability to omit the dots in concatenations and the curly braces for sets with only one element.

(b) Show that the language consisting of the binary words that start with 0 and end with 1 is C-D.

(c) Show that 0^* is C-D.

(d) Show that if R and S are C-D, then so is $R \cap S$.

(e) Show that $\{01\}^*$ is C-D.

Let's say a language S is 0-*finite* when it includes only a finite number of words whose bits are all 0's, that is, when $S \cap 0^*$ is a finite set of words. A langauge S is 0-*boring*—boring, for short—when either S or $\overline{S}$ is 0-finite.

(f) Explain why $\{00\}^*$ is not boring.

(g) Verify that if R and S are boring, then so is $R \cup S$.

(h) Verify that if R and S are boring, then so is $R \cdot S$.

Hint: By cases: whether R and S are both 0-finite, whether R or S contains no all-0 words at all (including the empty word λ), and whether neither of these cases hold.

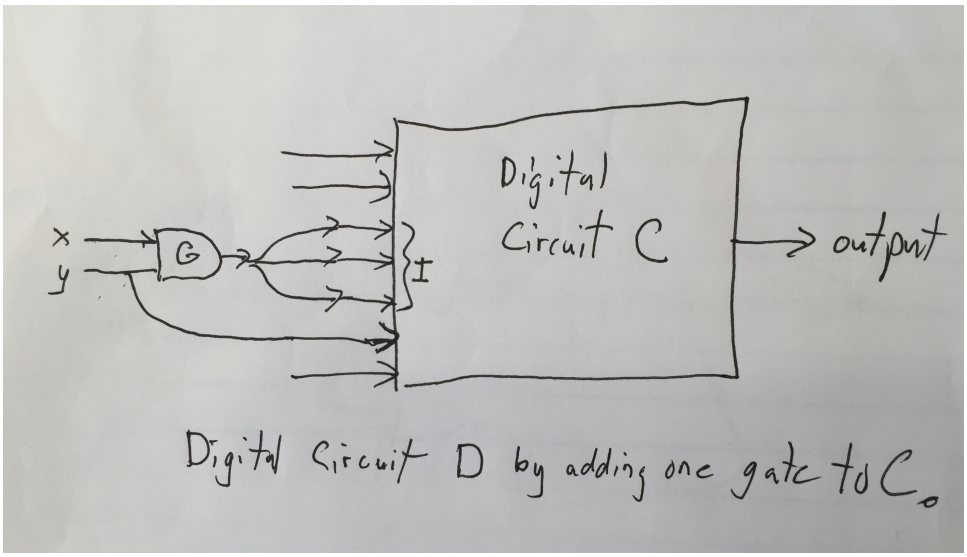


Figure 7.12 Digital Circuit Constructor Step

(i) Conclude by structural induction that all C-D languages are boring.

So we have proved that the set $(00)^*$ of even length all-0 words is not a C-D language.

Problem 7.24.

We can explain in a simple and precise way how digital circuits work, and gain the powerful proof method of structural induction to verify their properties, by defining digital circuits as a recursive data type DigCirc. The definition is a little easier to state if all the gates in the circuit take two inputs, so we will use the two-input NOR gate rather than a one-input NOT, and let the set of gates be

$$\text{Gates} ::= \{\text{NOR}, \text{AND}, \text{OR}, \text{XOR}\}.$$

A digital circuit will be a recursively defined list of *gate connections* of the form (x, y, G, I) where G is a gate, x and y are the input wires, and I is the set of wires that the gate output feeds into as illustrated in Figure 7.12.

Formally, we let W be a set $w_0, w_1, \ldots$ whose elements are called *wires*, and $\mathbf{O} \notin W$ be an object called the *output*.

Definition. The set of digital circuit DigCirc, and their inputs and internal wires, are defined recursively as follows:

Base case: If $x, y \in W$, then $C \in$ DigCirc, where

$$\begin{aligned} C &= \text{list}((\text{x}, \text{y}, \text{G}, \{\mathbf{O}\})) \text{ for some } G \in \text{Gates}, \\ \text{inputs(C)} &::= \{x, y\}, \\ \text{internal(C)} &::= \emptyset. \end{aligned}$$

Constructor cases: If

$$\begin{aligned} C &\in \text{DigCirc}, \\ I &\subseteq \text{inputs(C)}, I \neq \emptyset, \\ x, y &\in W - (I \cup \text{internal(C)}) \end{aligned}$$

then $D \in$ DigCirc, where

$$\begin{aligned} D &= \text{cons}((\text{x}, \text{y}, \text{G}, \text{I}), \text{C}) \text{ for some } G \in \text{Gates}, \\ \text{inputs(D)} &::= \{x, y\} \cup (\text{inputs(C)} - I), \\ \text{internal(D)} &::= \text{internal(C)} \cup I. \end{aligned}$$

For any circuit C define

$$\text{wires(C)} ::= \text{inputs(C)} \cup \text{internal(C)} \cup \{\mathbf{O}\}.$$

A *wire assignment* for C is a function

$$\alpha : \text{wires(C)} \to \{\mathbf{T}, \mathbf{F}\}$$

such that for each gate connection $(x, y, G, I) \in C$,

$$\alpha(i) = (\alpha(x)\ G\ \alpha(y)) \text{ for all } i \in I.$$

(a) Define an *environment* for C to be a function $e : \text{inputs(C)} \to \{\mathbf{T}, \mathbf{F}\}$. Prove that if two wire assignments for C are equal for each wire in inputs(C), then the wire assignments are equal for all wires.

Part (a) implies that for any environment e for C, there is a *unique* wire assignment α_e such that

$$\alpha_e(w) = e(w) \text{ for all } w \in \text{inputs(C)}.$$

So for any input environment e, the circuit computes a unique *output*

$$\text{eval}(C, e) ::= \alpha_e(\mathbf{O}).$$

Now suppose F is a propositional formula whose propositional variables are the input wires of some circuit C. Then C and F are defined to be *equivalent* iff

$$\text{eval}(C, e) = \text{eval}(F, e)$$

for all environments e for C.

(b) Define a function $E(C)$ recursively on the definition of circuit C, such that $E(C)$ is a propositional formula equivalent to C. Then verify the recursive definition by proving the equivalence using structural induction.

(c) Give examples where $E(C)$ is exponentially larger than C.

Exam Problems

Problem 7.25.
Let P be a propositional variable.
(a) Show how to express NOT(P) using P and a selection from among the constant **True**, and the connectives XOR and AND.

The use of the constant **True** above is essential. To prove this, we begin with a recursive definition of XOR-AND formulas that do not use **True**, called the PXA formulas.

Definition. Base case: The propositional variable P is a PXA formula.

Constructor cases If $R, S \in$ PXA, then

- R XOR S,
- R AND S

are PXA's.

For example,

$$(((P \text{ XOR } P) \text{ AND } P) \text{ XOR } (P \text{ AND } P)) \text{ XOR } (P \text{ XOR } P)$$

is a PXA.

(b) Prove by structural induction on the definition of PXA that every PXA formula A is equivalent to P or to **False**.

Problems for Section 7.3

Homework Problems

Problem 7.26.
One version of the the Ackermann function $A : \mathbb{N}^2 \to \mathbb{N}$ is defined recursively by the following rules:

$$A(m,n) ::= 2n \qquad \text{if } m = 0 \text{ or } n \leq 1, \qquad \text{(A-base)}$$
$$A(m,n) ::= A(m-1, A(m,n-1)) \qquad \text{otherwise.} \qquad \text{(AA)}$$

Prove that if $B : \mathbb{N}^2 \to \mathbb{N}$ is a partial function that satisfies this same definition, then B is total and $B = A$.

Problems for Section 7.4

Practice Problems

Problem 7.27. **(a)** Write out the evaluation of

$$\text{eval}(\text{subst}(3x, x(x-1)), 2)$$

according to the Environment Model and the Substitution Model, indicating where the rule for each case of the recursive definitions of eval(,) and [:=] or substitution is first used. Compare the number of arithmetic operations and variable lookups.

(b) Describe an example along the lines of part (a) where the Environment Model would perform 6 fewer multiplications than the Substitution model. You need *not* carry out the evaluations.

(c) Describe an example along the lines of part (a) where the Substitution Model would perform 6 fewer multiplications than the Environment model. You need *not* carry out the evaluations.

Class Problems

Problem 7.28.
In this problem we'll need to be careful about the propositional *operations* that apply to truth values and the corresponding *symbols* that appear in formulas. We'll restrict ourselves to formulas with *symbols* **And** and **Not** that correspond to the operations AND, NOT. We will also allow the constant symbols **True** and **False**.

(a) Give a simple recursive definition of *propositional formula* F and the set $\text{pvar}(F)$ of propositional *variables that appear* in it.

Let V be a set of propositional variables. A *truth environment* e over V assigns truth values to all these variables. In other words, e is a total function,

$$e : V \to \{\mathbf{T}, \mathbf{F}\}.$$

(b) Give a recursive definition of the *truth value*, $\text{eval}(F, e)$, of propositional formula F in an environment e over a set of variables $V \supseteq \text{pvar}(F)$.

Clearly the truth value of a propositional formula only depends on the truth values of the variables in it. How could it be otherwise? But it's good practice to work out a rigorous definition and proof of this assumption.

(c) Give an example of a propositional formula containing the variable P but whose truth value does not depend on P. Now give a rigorous definition of the assertion that "the truth value of propositional formula F does not depend on propositional variable P."

Hint: Let e_1, e_2 be two environments whose values agree on all variables other than P.

(d) Give a rigorous definition of the assertion that "the truth value of a propositional formula only depends on the truth values of the variables that appear in it," and then prove it by structural induction on the definition of propositional formula.

(e) Now we can formally define F being *valid*. Namely, F is valid iff

$$\forall e.\ \text{eval}(F, e) = \mathbf{T}.$$

Give a similar formal definition of formula G being *unsatisfiable*. Then use the definition of eval to prove that a formula F is valid iff $\mathbf{Not}(F)$ is unsatisfiable.

Exam Problems

Problem 7.29.

A class of propositional formulas called the Multivariable AND-OR (MVAO) formulas are defined recursively as follows:

Definition. Base cases: A single propositional variable, and the constants **True** and **False** are MVAO formulas.

Constructor cases: If $G, H \in$ MVAO, then $(G \text{ AND } H)$ and $(G \text{ OR } H)$ are MVAO's.

For example,

$$(((P \text{ OR } Q) \text{ AND } P) \text{ OR } (R \text{ AND } \mathbf{True})) \text{ OR } (Q \text{ OR } \mathbf{False})$$

is a MVAO.

Definition. A propositional formula G is ***False**-decreasing* when substituting the constant **False** for some occurrences of its variables makes the formula "more false." More precisely, if G^f is the result of replacing some occurrences of variables in G by **False**, then any truth assignment that makes G false also makes G^f false.

For example, the formula consisting of a single variable P is **False**-decreasing since P^f is the formula **False**. The formula $G ::= \overline{P}$ is not **False**-decreasing since G^f is the formula $\overline{\mathbf{False}}$ which is true even under a truth assignment where G is false.

Prove by structural induction that every MVAO formula F is **False**-decreasing.

Homework Problems

Problem 7.30. **(a)** Give a recursive definition of a function erase(e) that erases all the symbols in $e \in$ Aexp but the brackets. For example

$$\text{erase}([[3 * [x * x]] + [[2 * x] + 1]]) = [[[]][[2 * x] + 1]].$$

(b) Prove that erase(e) $\in$ RecMatch for all $e \in$ Aexp.

(c) Give an example of a small string $s \in$ RecMatch such that $[s] \neq \text{erase}(e)$ for any $e \in$ Aexp.

Problem 7.31.
Defining predicate formulas as a recursive data type leads to straightforward recursive definitions of free variables, prenex forms, and many other formula properties.

We'll only consider the propositional connectives **And**, **Not**, and **Implies**, since these are enough to illustrate how to handle all the other propositional connectives. For simplicity, the only atomic formulas allowed will be predicate symbols applied to variables. So function symbols and the equality symbol "=" will not appear.

Definition. **Base case**: $P(u, \ldots, v)$ is a predicate formula, where P is a predicate symbol and $u, \ldots, v$ are variable symbols, not necessarily distinct.

Constructor step: If F and G are predicate formulas, then so are

$$\textbf{Not}(F), (F \textbf{ And } G), (F \textbf{ Implies } G), (\exists x.\, F), (\forall x.\, F),$$

where x is a variable.

(a) Give a recursive definition of the set fvar(F) of *free variables* in a predicate formula F. (A variable x is free iff it has an occurrence that is not inside any subformula of the form $\exists x.[\ldots]$ or $\forall x.[\ldots]$.)

A predicate formula is in *prenex form* when all its quantifiers appear at the beginning of the formula; some examples were given in Problem 3.42. More formally, a *prenex formula* is a predicate formula of the form

$$\text{qnt}(F).\ \text{bod}(F),$$

where bod(F) is a quantier-free formula, and qnt F is a (possibly empty) sequence of quantifiers

$$Q_1 x_1.\ Q_2 x_2.\ \ldots\ Q_n x_n.$$

where Q_i is "$\forall$" or "$\exists$," and the x_i's are distinct variables.

It is easy to convert a formula into an equivalent prenex formula if the given formula satisfies the *unique variable convention*: for each variable x, there is at most one occurrence of either a "$\forall x$" or an "$\exists x$," and that if x is a free variable, there is no occurrence of a "$\forall x$" or an "$\exists x$" anywhere else in the formula. (A way of renaming variables to convert a formula into an equivalent one satisfying the unique variable convention was illustrated in Problem 3.41. The general renaming procedure is given in Problem 7.32.)

(b) Let F be a predicate formula satisfying the unique variable convention. Define a recursive procedure based on the recursive definition of predicate formulas that converts F into an equivalent prenex formula whose body is the same as F's with the quantifiers erased. (That is, the propositional connectives of F don't change, in contrast to the method suggested in Problem 3.42.)

It will help to use the notation "$\overline{\text{qnt}(F)}$" for the sequence

$$\overline{Q_1} x_1.\ \overline{Q_2} x_2.\ \ldots\ \overline{Q_n} x_n.$$

where $\overline{\forall} ::= \exists$ and $\overline{\exists} ::= \forall$.

For example, the variable convention ensures that if $(Qx.F)$ is a subformula somewhere, then there are no free occurrences of x anywhere. Therefore,

$$(Qx.F) \textbf{ And } G \quad \text{is equivalent to} \quad Qx.(F \textbf{ And } G),$$

because we can think of G as simply being **True** or **False**. This explains why the following definition of qnt($(F$ **And** $G)$) and bod($(F$ **And** $G)$) preserves equivalence:

$$\text{bod}((F \textbf{ And } G)) ::= (\text{bod}(F) \textbf{ And } \text{bod}(G)),$$
$$\text{qnt}((F \textbf{ And } G)) ::= \text{qnt}(F), \text{qnt}(G).$$

Problem 7.32.
In all modern programming languages, re-using the same procedure names or procedure parameters within different procedure declarations doesn't cause trouble. The programming language's "scoping" rules eliminate any ambiguity about which procedure names or procedure parameters bind to which declarations in any particular context. It's always safe to choose whatever names you like, and your procedure will behave as if your chosen names were unique, and never appeared elsewhere.

The same thing happens in predicate formulas.[12] A variable x may appear at different places bound by different $\forall x$ or $\exists x$ quantifiers, but there are standard rules for figuring out which quantifier binds which occurrence of x. In fact, there is a way to rename variables to make the variable bindings explicit without changing the meaning of the formula or its structure. An example of this was given in Problem 3.41.

In this problem, we give a recursive definition of a general variable renaming function UV that converts any predicate formula F into an *equivalent* formula UV(F) that satisfies the *unique variable convention*:

- For every variable x occurring in UV(F), there is at most one quantified occurrence of x, that is, at most one occurrence of either "$\forall x$" or "$\exists x$," and moreover, "$\forall x$" and "$\exists x$" don't both occur, and
- if there is a subformula of UV(F) of the form $\forall x.G$ or the form $\exists x.G$, then all the occurrences of x that appear anywhere in the whole formula are within the subformula.

From here on we will ignore the distinction between connective symbols and propositional operations, for example, between the symbol **And** and the operation AND. We also restrict ourselves to the propositional connectives AND and NOT.

To define the function UV, we need an operation that mindlessly renames variables (free and bound). We recursively define renm($F, x := y$) to be a mindless renaming of all occurrences of x in F to y:

[12] In fact historically, these issues of variable binding scope first came up in the study of predicate formulas.

Definition. The renaming function, renm, is defined recursively as follows:
For variables x, y, z,

$$\begin{aligned} \text{renm}(x, x := y) &::= y, \\ \text{renm}(z, x := y) &::= z \qquad \text{(for } z \text{ different from } x\text{)}. \end{aligned}$$

Base case: $P(u, \ldots, v)$

$$\text{renm}(P(u, \ldots, v), x := y) ::= P(\text{renm}(u, x := y), \ldots, \text{renm}(v, x := y)).$$

Constructors:

$$\begin{aligned} \text{renm}(\text{NOT}(G), x := y) &::= \text{NOT}(\text{renm}(G, x := y)), \\ \text{renm}((G \text{ AND } H), x := y) &::= (\text{renm}(G, x := y) \text{ AND } \text{renm}(G, x := y)), \\ \text{renm}(\forall u.\, G, x := y) &::= \forall \text{renm}(u, x := y).\, \text{renm}(G, x := y), \\ \text{renm}(\exists u.\, G, x := y) &::= \exists \text{renm}(u, x := y).\, \text{renm}(G, x := y). \end{aligned}$$

To define the Unique Variable Convention transform, we will actually need an auxiliary function UV_2 that takes two arguments, the formula F to be converted, and a set V of already-used variables that have to be avoided. Moreover, $\text{UV}_2(F, V)$ will return a *pair* of results: $\text{left}(\text{UV}_2(F, V))$ will be a converted formula satisfying the unique variable convention, and $\text{right}(\text{UV}_2(F, V))$ will be the set V extended with all the variables that appear in $\text{left}(\text{UV}_2(F, V))$. These are the variables which must be avoided when renaming other formulas. Then we will define

$$\text{UV}(F) ::= \text{UV}_2(F, \emptyset).$$

Let newvar be a function with the property that for any finite set V of variables, $\text{newvar}(V)$ is a "new" variable not in V.

Definition. Base case: $(F = P(u, \ldots, v))$:

$$\begin{aligned} \text{left}(\text{UV}_2(F, V)) &::= F, \\ \text{right}(\text{UV}_2(F, V)) &::= V \cup \{u, \ldots, v\}. \end{aligned}$$

Constructors:
Case $(F = \text{NOT}(G))$:

$$\begin{aligned} \text{left}(\text{UV}_2(F, V)) &::= \text{NOT}(\text{left}(\text{UV}_2(G, V))), \\ \text{right}(\text{UV}_2(F, V)) &::= \text{right}(\text{UV}_2(G, V)). \end{aligned}$$

Case $F = (G \text{ AND } H)$: Let $V' ::= \text{right}(\text{UV}_2(G, V))$.

$$\begin{aligned}\text{left}(\text{UV}_2(F, V)) &::= (\ \text{left}(\text{UV}_2(G, V)) \text{ AND } \text{left}(\text{UV}_2(H, V'))\ ),\\ \text{right}(\text{UV}_2(F, V)) &::= \text{right}(\text{UV}_2(H, V')).\end{aligned}$$

Case ($F = Q\, x.\, G$) where $Q = \exists, \forall$:
Let $y ::= \text{newvar}(V)$, $V' ::= V \cup \{y\}$.

$$\begin{aligned}\text{left}(\text{UV}_2(F, V)) &::= Q\ y.\, \text{renm}((\text{left}(\text{UV}_2(G, V'))), x := y),\\ \text{right}(\text{UV}_2(F, V)) &::= \text{right}(\text{UV}_2(G, V')).\end{aligned}$$

(a) A predicate formula defines a property of its free variables, so converting it to an equivalent formula should leave the same set of free variables. Prove that

$$\text{fvar}(F) = \text{fvar}(\text{UV}_2(F, V))$$

for all sets of variables V.

(b) Prove that $\text{UV}_2(F, V)$ contains no occurrence of any variable in V.

(c) Prove that $\text{UV}_2(F, V)$ satisfies the unique variable convention.

(d) When there is no need to avoid a particular set of variables, the UV function will convert a formula back into a standard one: prove that

$$\text{UV}(\text{UV}_2(F, V)) = \text{UV}(F)$$

for all formulas F and sets V of variables.

Problems for Section 7.5

Practice Problems

Problem 7.33.
In the game Tic-Tac-Toe, there are nine first-move games corresponding to the nine boxes that player-1 could mark with an "X".

Each of these nine games will themselves have eight first-move games corresponding to where the second player can mark his "O", for a total of 72 "second-move" games.

Answer the following questions about the subsequent Tic-Tac-Toe games.

(a) How many third-move games are there where player-1 can mark his second "X"?

(b) What is the first level where this simple pattern of calculating how many first moves are possible in each subsequent game stops working?

Problem 7.34. (a) List the set $\text{Fmv}_{\langle 4\rangle}$ of possible Nim games that can result from the making the first move in the Nim game $\text{Nim}_{\langle 4\rangle}$ with one size-four pile.

(b) What is the representation of $\text{Nim}_{\langle 4\rangle}$ as a pure set "built out of nothing?"

(c) What piles of stones are there is the Nim game whose set representation is

$$\{\{\{\{\emptyset\}\}\}\}$$

Homework Problems

Problem 7.35.
We're going to characterize a large category of games as a recursive data type and then prove, by structural induction, a fundamental theorem about game strategies. We are interested in two person games of perfect information that end with a numerical score. Chess and Checkers would count as value games using the values $1, -1, 0$ for a win, loss or draw for the first player. The game of Go really does end with a *score* based on the number of white and black stones that remain at the end.

Here's the formal definition:

Definition. Let V be a nonempty set of real numbers. The class VG of *V-valued two-person deterministic games of perfect information* is defined recursively as follows:

Base case: A value $v \in V$ is a VG known as a *payoff*.

Constructor case: If G is a nonempty set of VG's, then G is a VG. Each game $M \in G$ is called a possible *first move* of G.

A *strategy* for a player is a rule that tells the player which move to make whenever it is their turn. That is, a strategy is a function s from games to games with the property that $s(G) \in G$ for all games G. Given which player has the first move, a pair of strategies for the two players determines exactly which moves the players will choose. So the strategies determine a unique play of the game and a unique payoff.[13]

[13] We take for granted the fact that no VG has an infinite play. The proof of this by structural induction is essentially the same as that for win-lose games given in Lemma 7.5.2.

The *max-player* wants a strategy that guarantees as high a payoff as possible, and the *min-player* wants a strategy that guarantees as low a payoff as possible.

The Fundamental Theorem for deterministic games of perfect information says that in any game, each player has an optimal strategy, and these strategies lead to the same payoff. More precisely,

Theorem (Fundamental Theorem for VG's)**.** *Let V be a finite set of real numbers and G be a V-valued VG. Then there is a value $v \in V$, called a* max-value max_G *for G, such that if the* max-player moves first*,*

- *the max-player has a strategy that will finish with a payoff of* at least max_G*, no matter what strategy the min-player uses, and*
- *the min-player has a strategy that will finish with a payoff of* at most max_G*, no matter what strategy the max-player uses.*

It's worth a moment for the reader to observe that the definition of $\max_G$ implies that if there is one for G, it is unique. So if the max-player has the first move, the Fundamental Theorem means that there's no point in playing the game: the min-player may just as well pay the max-value to the max-player.

(a) Prove the Fundamental Theorem for VG's.

Hint: VG's are a recursively defined data type, so the basic method for proving that all VG's have some property is structural induction on the definition of VG. Since the min-player moves first in whichever game the max-player picks for their first move, the induction hypothesis will need to cover that case as well.

(b) (OPTIONAL). State some reasonable generalization of the Fundamental Theorem to games with an infinite set V of possible payoffs.

Problem 7.36.

Nim is a two-person game that starts with some piles of stones. A player's move consists of removing one or more stones from a single pile. The players alternate making moves, and whoever takes the last stone wins.

It turns out there is a winning strategy for one of the players that is easy to carry out but is not so obvious.

To explain the winning strategy, we need to think of a number in two ways: as a nonnegative integer and as the bit string equal to the binary representation of the number—possibly with leading zeroes.

For example, the XOR of *numbers* $r, s, \ldots$ is defined in terms of their binary representations: combine the corresponding bits of the binary representations of $r, s, \ldots$

using XOR, and then interpret the resulting bit-string as a number. For example,

$$2 \text{ XOR } 7 \text{ XOR } 9 = 12$$

because, taking XOR's down the columns, we have

0	0	1	0	(binary rep of 2)
0	1	1	1	(binary rep of 7)
1	0	0	1	(binary rep of 9)
1	1	0	0	(binary rep of 12)

This is the same as doing binary addition of the numbers, but throwing away the carries (see Problem 3.6).

The XOR of the numbers of stones in the piles is called their *Nim sum*. In this problem we will verify that if the Nim sum is not zero on a player's turn, then the player has a winning strategy. For example, if the game starts with five piles of equal size, then the first player has a winning strategy, but if the game starts with four equal-size piles, then the second player can force a win.

(a) Prove that if the Nim sum of the piles is zero, then any one move will leave a nonzero Nim sum.

(b) Prove that if there is a pile with more stones than the Nim sum of all the other piles, then there is a move that makes the Nim sum equal to zero.

(c) Prove that if the Nim sum is not zero, then one of the piles is bigger than the Nim sum of the all the other piles.

Hint: Notice that the largest pile may not be the one that is bigger than the Nim sum of the others; three piles of sizes 2,2,1 is an example.

(d) Conclude that if the game begins with a nonzero Nim sum, then the first player has a winning strategy.

Hint: Describe a preserved invariant that the first player can maintain.

(e) (Extra credit) Nim is sometimes played with winners and losers reversed, that is, the person who takes the last stone *loses*. This is called the *misère* version of the game. Use ideas from the winning strategy above for regular play to find one for *misère* play.

Problems for Section 7.6

Practice Problems

Problem 7.37.
Suppose every tree $T \in \text{RecTr}$ has a numerical label $\text{num}(T) \in \mathbb{R}$. The *minimum value* in $\min(T)$ is defined to be the smallest label among its subtrees. The function

$$\text{recmin} : \text{RecTr} \to \mathbb{R}$$

is recursively defined as follows:

Base case: (T is a leaf).

$$\text{recmin}(T) ::= \text{num}(T).$$

Constructor case: ($T \in$ Branching)

$$\text{recmin}(T) ::= \min\{\text{num}(T), \text{recmin}(\text{left}(T)), \text{recmin}(\text{right}(T))\}$$

Prove that

$$\text{recmin}(T) = \min(T)$$

for all $T \in \text{RecTr}$.

Problem 7.38.
Verify that every Search tree is a recursive tree.
Hint: Show that Search trees can't have shared subtrees.

Class Problems

Problem 7.39.
For $T \in \text{BBTr}$, define

$$\text{leaves}(T) ::= \{S \in \text{Subtrs}(T) \mid S \in \text{Leaves}\}$$
$$\text{internal}(T) ::= \{S \in \text{Subtrs}(T) \mid S \in \text{Branching}\}.$$

(a) Explain why it follows immediately from the definitions that if $T \in$ Branching,

$$\text{internal}(T) = \{T\} \cup \text{internal}(\text{left}(T)) \cup \text{internal}(\text{right}(T)), \tag{trnlT}$$
$$\text{leaves}(T) = \text{leaves}(\text{left}(T)) \cup \text{leaves}(\text{right}(T)). \tag{lvT}$$

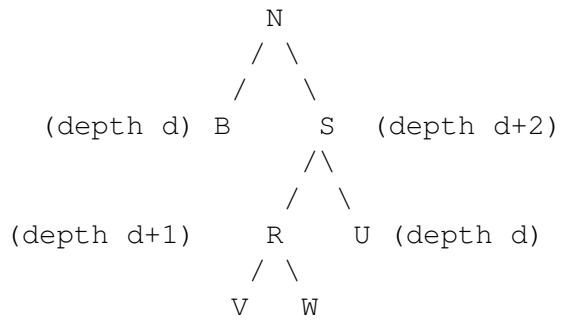


Figure 7.13 The Tree N.

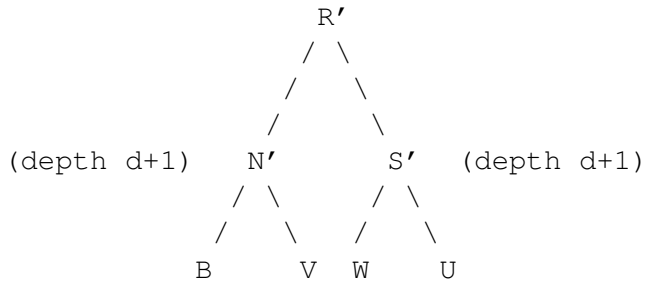


Figure 7.14 R' is the rearrangement of tree N in Figure 7.13.

(b) Prove by structural induction on the definition of RecTr (Definition 7.6.8) that in a recursive tree, there is always one more leaf than there are internal subtrees:
Lemma. *If $T \in RecTr$, then*

$$|leaves(T)| = 1 + |internal(T)|. \tag{lf-vs-in}$$

Problem 7.40.
Suppose B is an AVL tree of depth d, S is an AVL tree of depth $d+2$, and N is a search tree with branches B and S as shown in Figure 7.13. Also, S has as branches an AVL tree R of depth $d+1$ and an AVL tree U with depth d, as shown.

We define R', N', S' to be new trees with

$$\begin{aligned} \text{num}(R') &= \text{num}(R) \\ \text{num}(N') &= \text{num}(N) \\ \text{num}(S') &= \text{num}(S), \end{aligned}$$

with branches as indicated in Figure 7.14.

Verify that R' is an AVL tree and $\text{nums}(R') = \text{nums}(N)$.

Homework Problems

Problem 7.41.

Definition. Define the *sharing binary trees* SharTr recursively:

Base case: ($T \in$ Leaves). $T \in$ SharTr.

Constructor case: ($T \in$ Branching). If $\text{left}(T), \text{right}(T) \in$ SharTr, then T is in SharTr.

(a) Prove size (T) is finite for every $T \in$ SharTr.

(b) Give an example of a finite $T \in$ BBTr that has an infinite path.

(c) Prove that for all $T \in$ BBTr

$$T \in \text{SharTr} \longleftrightarrow T \text{ has no infinite path.}$$

(d) Give an example of a tree $T_3 \in$ BBTr with three branching subtrees and one leaf.

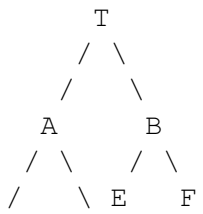
(e) Prove that

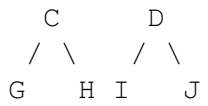
Lemma. *If $T \in$ SharTr, then*

$$|\mathit{leaves}(T)| \leq 1 + |\mathit{internal}(T)|.$$

Hint: Show that for every $T \in$ SharTr, there is a recursive tree $R \in$ RecTr with the same number of internal subtrees and at least as many leaves.

Problem 7.42. (a) Edit the labels in this size 11 tree T so it becomes a search tree for the set of labels [1..11].





Reminder:
Definition. The Search trees $T \in$ BBTr are defined recursively as follows:

Base case: ($T \in$ Leaves). T is a Search tree.

Constructor case: ($T \in$ Branching). If left(T) and right(T) are both Search trees, and

$$\max(\text{left}(T)) < \text{num}(T) < \min(\text{right}(T)),$$

then T is a Search tree.

(b) Let T be a search tree whose labels are the consecutive integers in the interval $[a..b]$ where a, b are odd integers with $b - a = \text{size}(T) - 1$ (for example, $[1..\text{size}(T)]$). Prove by structural induction on Definition 7.6.24 of search tree that the leaves of T are precisely the subtrees whose label is an odd number:

$$\text{leaves}(T) = \{S \in \text{Subtrs}(T) \mid \text{num}(S) \text{ is odd}\}. \qquad \text{(odd-label)}$$

It now follows that a search tree for a given set of values cannot have *any* leaves, and therefore cannot have any subtrees, in common with a search for the given values after just one deletion and one insertion.

For example, suppose T is a search tree for the values $[1..n]$, and U is a search tree for these values after deleting 1 and inserting $n+1$, that is $\text{nums}(U) = [2..(n+1)]$. Then the leaves of T are all odd numbered, and the leaves of U will be even numbered (think why![14]).

This contrasts dramatically with the case of AVL trees, in which leaves are allowed to hold two values. Storing two values at leaves provides enough "slack" to allow incremental updating of the overall tree, so that after an insertion and deletion, the updated tree can share all but proportional to $\log_2$ (tree size) subtrees.

Problem 7.43.
Define and verify correctness of a procedure delete for deleting a value from an AVL tree. The procedure should require a number of label comparisons and new subtrees at worst proportional to log of the size of the AVL tree.

[14] Subtracting one from all the labels of U turns it into a search tree for $[1..n]$ whose leaves must have odd numbered labels. Alternatively, the argument of part (b) could simply be applied to even numbered a, b.

Exam Problems

Problem 7.44.
Prove by structural induction on the definition of recursive binary trees (Def.7.6.8) that

$$\text{recdepth}(T) = \text{depth}(T)$$

for all $T \in \text{RecTr}$.

Problem 7.45.
The *isomorphism* relation between recursive trees $T, U \in \text{RecTr}$ is defined recursively

Base case: ($T \in$ Leaves). T is isomorphic to U iff U is a leaf.

Constructor case: ($T \in$ Branching). T is isomorphic to U iff

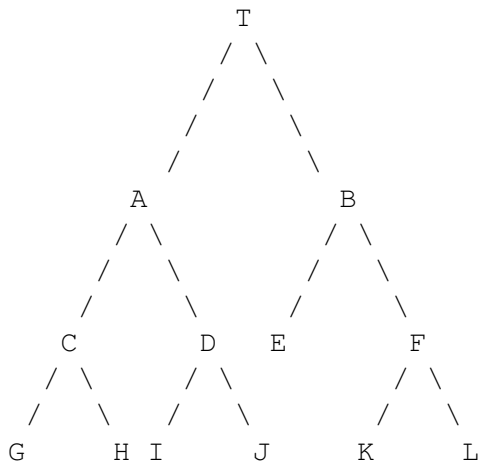
$U \in$ Branching AND left(U) isomorphic to left(T) AND right(U) isomorphic to right(T).

(a) Prove that if T and U are isomorphic, then $\text{size}(T) = \text{size}(U)$.

(b) Show that if T and U are isomorphic search trees for the same set of values, that is, $\text{nums}(T) = \text{nums}(U)$, then $\text{num}(T) = \text{num}(U)$.

Hint: By definition of isomorphism and search tree. No induction needed.

Problem 7.46. (a) Edit the labels in this size 15 tree T so it becomes a search tree for the set of labels [1..15].



```
  /  \
M     N
```

(b) For any recursive tree and set of labels, there is only one way to assign labels to make the tree a search tree. More precisely, let num : RecTr $\to \mathbb{R}$ be a labelling function on the recursive binary trees, and suppose T is a search tree under this labelling. Suppose that num_{alt} is another labelling and that T is also a search tree under num_{alt} for the *same* set of labels. Prove by structural induction on the definition of search tree (Definition 7.6.24) that

$$\text{num}(S) = \text{num}_{\text{alt}}(S) \qquad \text{(same)}$$

for all subtrees $S \in \text{Subtrs}(T)$.

Reminder:
Definition. The Search trees $T \in \text{BBTr}$ are defined recursively as follows:

Base case: ($T \in$ Leaves). T is a Search tree.

Constructor case: ($T \in$ Branching). If left(T) and right(T) are both Search trees, and

$$\max(\text{left}(T)) < \text{num}(T) < \min(\text{right}(T)),$$

then T is a Search tree.