6 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.

6.1 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 for.

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 6.1.1. Let A be a nonempty set called an *alphabet*, whose elements are referred to as *characters*, *letters*, or *symbols*. The recursive data type, A^*, of strings over alphabet, A, are 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 6.1.1, 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 6.1.2. The length, $|s|$, of a string, s, is defined recursively based on the definition of $s \in A^*$:

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

Base case:

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

Constructor case:

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

6.1.1 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, we can verify the familiar fact that the length of the concatenation of two strings is the sum of their lengths using structural induction:

Theorem 6.1.4. *For all* $s, t \in A^*$,

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

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 ($s = \lambda$):

$$\begin{aligned} |s \cdot t| &= |\lambda \cdot t| \\ &= |t| && \text{(def } \cdot\text{, base case)} \\ &= 0 + |t| \\ &= |s| + |t| && \text{(def length, base case)} \end{aligned}$$

Constructor case: Suppose $s ::= \langle a, r\rangle$ and assume the induction hypothesis, $P(r)$. We must show that $P(s)$ holds:

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

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

This proof illustrates 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, **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 $r \in R$.

6.1.2 One More Thing

The number, $\#_c(s)$, of occurrences of the character $c \in A$ in the string s has a simple recursive definition based on the definition of $s \in A^*$:

Definition 6.1.5.

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

Constructor case:

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

We'll need the following lemma in the next section:

Lemma 6.1.6.

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

The easy proof by structural induction is an exercise (Problem 6.7).

6.2 Strings of Matched Brackets

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

$$\texttt{[]][[[[[]]]}\quad \text{and} \quad \texttt{[[[]]][]][]} \tag{6.1}$$

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 6.2.1. 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,

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

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.

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

Proof. The proof is by structural induction with induction hypothesis

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

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

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

by the base case of Definition 6.1.5 of $\#_c()$.

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

$$\begin{aligned} \#_{[}([\,s\,]\,t) &= \#_{[}([\,) + \#_{[}(s) + \#_{[}(\,]) + \#_{[}(t) && \text{(Lemma 6.1.6)} \\ &= 1 + \#_{[}(s) + 0 + \#_{[}(t) && \text{(def } \#_{[}()\text{)} \\ &= 1 + \#_{]}(s) + 0 + \#_{]}(t) && \text{(by } P(s) \text{ and } P(t)\text{)} \\ &= 0 + \#_{]}(s) + 1 + \#_{]}(t) \\ &= \#_{]}([\,) + \#_{]}(s) + \#_{]}(\,]) + \#_{]}(t) && \text{(def } \#_{]}()\text{)} \\ &= \#_{]}([\,s\,]\,t) && \text{(Lemma 6.1.6)} \end{aligned}$$

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 6.2.2. 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 6.15). 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 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!

6.3 Recursive Functions on Nonnegative Integers

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

Definition 6.3.1. 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 6.3.1. This also justifies the familiar recursive definitions of functions on the nonnegative integers.

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

Example 6.3.2. *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 6.3.3. *The Fibonacci numbers.* Fibonacci numbers arose out of an effort 800 years ago to model population growth. They have a continuing fan club of people captivated by their extraordinary properties (see Problems 5.8, 5.21, 5.26). The nth Fibonacci number, fib, can be defined recursively by:

$$\begin{aligned} F(0) &::= 0, \\ F(1) &::= 1, \\ F(n) &::= F(n-1) + F(n-2) \qquad \text{for } n \geq 2. \end{aligned}$$

Here the recursive step starts at $n = 2$ with base cases for 0 and 1. This is needed since the recursion relies on two previous values.

What is $F(4)$? Well, $F(2) = F(1) + F(0) = 1$, $F(3) = F(2) + F(1) = 2$, so $F(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Example 6.3.4. *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 $n \geq 0$.

6.3.2 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.

$$f_1(n) ::= 2 + f_1(n-1). \tag{6.2}$$

This "definition" has no base case. If some function, f_1, satisfied (6.2), so would a function obtained by adding a constant to the value of f_1. So equation (6.2) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{6.3}$$

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 (6.3) 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 (6.3) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{6.4}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (6.4) doesn't define anything.

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

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{6.5}$$

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 (6.5), 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 (6.5) 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:

$$A(m,n) = 2n, \qquad \text{if } m = 0 \text{ or } n \leq 1, \tag{6.6}$$

$$A(m,n) = A(m-1, A(m,n-1)), \qquad \text{otherwise.} \tag{6.7}$$

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 6.17).

6.4 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 6.4.1.

- **Base cases**:
  - The variable, x, is in Aexp.
  - The arabic numeral, `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*.
- $\texttt{-}[e] \in$ Aexp. The expression $\texttt{-}[e]$ 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

$$[[3 * [x * x]] + [[2 * x] + 1]]. \tag{6.8}$$

These brackets and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2+2x+1$" instead of (6.8). But it's important to recognize that $3x^2+2x+1$ is not an Aexp; it's an *abbreviation* for an Aexp.

6.4.1 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, $\text{eval}(e, n)$. It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 6.4.2. The *evaluation function*, $\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in$ Aexp, as follows. Let n be any integer.

- **Base cases**:

$$\text{eval}(x, n) ::= n, \qquad \text{(value of variable } x \text{ is } n.) \tag{6.9}$$

$$\text{eval}(\texttt{k}, n) ::= k, \qquad \text{(value of numeral } \texttt{k} \text{ is } k\text{, regardless of } x.) \tag{6.10}$$

- **Constructor cases**:

$$\text{eval}([e_1 \texttt{+} e_2], n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n), \tag{6.11}$$

$$\text{eval}([e_1 * e_2], n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n), \tag{6.12}$$

$$\text{eval}(\texttt{-}[e_1], n) ::= -\text{eval}(e_1, n). \tag{6.13}$$

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

$$\begin{aligned}
\text{eval}([\,3 + [\,x * x\,]\,], 2) &= \text{eval}(3, 2) + \text{eval}([\,x * x\,], 2) && \text{(by Def 6.4.2.6.11)}\\
&= 3 + \text{eval}([\,x * x\,], 2) && \text{(by Def 6.4.2.6.10)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 6.4.2.6.12)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 6.4.2.6.9)}\\
&= 3 + 4 = 7.
\end{aligned}$$

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 6.4.3. 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**:

$$\text{subst}(f, x) ::= f, \quad \text{(subbing } f \text{ for variable, } x\text{, just gives } f\text{)} \tag{6.14}$$
$$\text{subst}(f, \mathtt{k}) ::= \mathtt{k} \quad \text{(subbing into a numeral does nothing.)} \tag{6.15}$$

- **Constructor cases**:

$$\text{subst}(f, [\,e_1 + e_2\,]) ::= [\,\text{subst}(f, e_1) + \text{subst}(f, e_2)\,] \tag{6.16}$$
$$\text{subst}(f, [\,e_1 * e_2\,]) ::= [\,\text{subst}(f, e_1) * \text{subst}(f, e_2)\,] \tag{6.17}$$
$$\text{subst}(f, -[\,e_1\,]) ::= -[\,\text{subst}(f, e_1)\,]. \tag{6.18}$$

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

$$
\begin{aligned}
&\text{subst}(3x, x(x-1)) \\
&= \text{subst}([\,3 * x\,], [\,x * [\,x + -[\,1]\,]\,]) && \text{(unabbreviating)} \\
&= [\ \text{subst}([\,3 * x\,], x) * \\
&\qquad \text{subst}([\,3 * x\,], [\,x + -[\,1]\,])] && \text{(by Def 6.4.3 6.17)} \\
&= [\,[\,3 * x\,] * \text{subst}([\,3 * x\,], [\,x + -[\,1]\,])] && \text{(by Def 6.4.3 6.14)} \\
&= [\,[\,3 * x\,] * [\ \text{subst}([\,3 * x\,], x) \\
&\qquad + \text{subst}([\,3 * x\,], -[\,1]\,)]\,] && \text{(by Def 6.4.3 6.16)} \\
&= [\,[\,3 * x\,] * [\,[\,3 * x\,] + -[\ \text{subst}([\,3 * x\,], 1)]\,]\,] && \text{(by Def 6.4.3 6.14 \& 6.18)} \\
&= [\,[\,3 * x\,] * [\,[\,3 * x\,] + -[\,1]\,]\,] && \text{(by Def 6.4.3 6.15)} \\
&= 3x(3x-1) && \text{(abbreviation)}
\end{aligned}
$$

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

$$\text{eval}(\text{subst}(3x, x(x-1)), 2) \tag{6.19}$$

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 other approach is called evaluation using the *Environment Model*. Namely, to compute the value of (6.19), 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

$$\text{eval}(x(x-1), \text{eval}(3x, 2)). \tag{6.20}$$

The Environment Model only computes the value of $3x$ once, and so it requires one fewer multiplication than the Substitution model to compute (6.20). This is a good place to stop and work this example out yourself (Problem 6.18).

But how do we know that these final values reached by these two approaches, that is, the final integer values of (6.19) and (6.20), agree? In fact, we can prove pretty easily that these two approaches *always* agree by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 6.4.4. *For all expressions e, $f \in Aexp$ and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{6.21}$$

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

Base cases:

- Case[x]

  The left hand side of equation (6.21) equals eval(f, n) by this base case in Definition 6.4.3 of the substitution function, and the right hand side also equals eval(f, n) by this base case in Definition 6.4.2 of eval.

- Case[k].

  The left hand side of equation (6.21) equals k by this base case in Definitions 6.4.3 and 6.4.2 of the substitution and evaluation functions. Likewise, the right hand side equals k by two applications of this base case in the Definition 6.4.2 of eval.

Constructor cases:

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

  By the structural induction hypothesis (6.21), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

  $$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{6.22}$$

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

  $$\text{eval}(\text{subst}(f, [\, e_1 + e_2 \,]), n) = \text{eval}([\, e_1 + e_2 \,], \text{eval}(f, n)) \tag{6.23}$$

  The left hand side of (6.23) equals

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

  by Definition 6.4.3.6.16 of substitution into a sum expression. But this equals

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

[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 6.4.2.(6.11) of eval for a sum expression. By induction hypothesis (6.22), 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 (6.23) by Definition 6.4.2.(6.11) of eval for a sum expression. This proves (6.23) 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.

■

6.5 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 operations in an Aexp. It is then possible to prove properties of data by ordinary induction on their size. But this approach 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 6.1

Class Problems

Problem 6.1.
Prove that for all strings $r, s, t \in A^*$

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

Problem 6.2.
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 definition 6.1.1 of $s \in A^*$ and using the concatenation operation 6.1.3.

(b) Prove that

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

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

Problem 6.3.
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 6.4.

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

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

Problem 6.5.

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

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

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

The *size*, $|G|$, of $G \in$ binary-2PTG 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-2PTG, G, pictured in Figure 6.1, is 7.

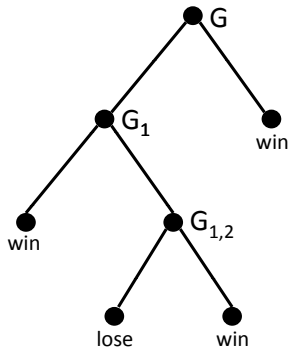

Figure 6.1 A picture of a binary tree G.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 6.1.

The value of flatten(G) for $G \in$ binary-2PTG is the sequence of labels in L of the leaves of G. For example, for the binary-2PTG, G, pictured in Figure 6.1,

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

(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

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{6.24}$$

Homework Problems

Problem 6.6.
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 $g \in L$ which divides every member of L.
Hint: The least positive integer in L.

(e) Conclude that $g = \text{GCD}(m, n)$ for g from part (d).

Problem 6.7.

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 $c \in A$

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

Figure 6.2 Constructing the Koch Snowflake.

Problem 6.8.
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 line segment on the snowflake. Remove the middle third of l, and replace it with two line segments of the same length as is done in Figure 6.2

  The resulting figure is also a Koch snowflake.

Prove by structural induction that the area inside any Koch snowflake is of the form $q\sqrt{3}$, where q is a rational number.

Problem 6.9.
Let L be some convenient set whose elements will be called *labels*. The labeled binary trees, LBT's, are defined recursively as follows:

Definition. Base case: if l is a label, then $\langle l, \texttt{leaf}\rangle$ is an LBT, and

Constructor case: if B and C are LBT's, then $\langle l, B, C\rangle$ is an LBT.

The *leaf-labels* and *internal-labels* of an LBT are defined recursively in the obvious way:

Definition. Base case: The set of leaf-labels of the LBT $\langle l, \texttt{leaf}\rangle$ is $\{l\}$, and its set of internal-labels is the empty set.

Constructor case: The set of leaf labels of the LBT $\langle l, B, C\rangle$ is the union of the leaf-labels of B and of C; the set of internal-labels is the union of $\{l\}$ and the sets of internal-labels of B and of C.

The set of *labels* of an LBT is the union of its leaf- and internal-labels.
The LBT's with *unique* labels are also defined recursively:

Definition. Base case: The LBT $\langle l, \texttt{leaf} \rangle$ has *unique labels*.

Constructor case: If B and C are LBT's with unique labels, no label of B is a label C and vice-versa, and l is not a label of B or C, then $\langle l, B, C \rangle$ has *unique labels*.

If B is an LBT, let n_B be the number of distinct i**n**ternal-labels appearing in B and f_B be the number of distinct lea**f** labels of B. Prove by structural induction that

$$f_B = n_B + 1 \tag{6.25}$$

for all LBT's B with unique labels. This equation can obviously fail if labels are not unique, so your proof had better use uniqueness of labels at some point; be sure to indicate where.

Exam Problems

Problem 6.10.

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 6.11.

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, $\cdot$, and
3. division $/$.

(a) Prove by structural induction that RAF is closed under composition. That is, using the induction hypothesis,

$$P(h) ::= \forall g \in \mathrm{RAF}\,.\, h \circ g \in \mathrm{RAF}, \tag{6.26}$$

prove that $P(h)$ holds for all $h \in \mathrm{RAF}$. Make sure to indicate explicitly

- each of the base cases, and
- each of the constructor cases. *Hint:* One proof in terms of $\circledast$ covers all three cases.

(b) Briefly indicate where a proof would break down using the very similar induction hypothesis

$$Q(g) ::= \forall h \in \mathrm{RAF}\,.\, h \circ g \in \mathrm{RAF}\,.$$

Problems for Section 6.2

Practice Problems

Problem 6.12.
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 $nm \in F_2$.

(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 firat 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 6.13. (a) To prove that the set RecMatch, of matched strings of Definition 6.2.1 equals the set AmbRecMatch of ambiguous matched strings of Definition 6.2.2, you could first prove that

$$\forall r \in \text{RecMatch}.\ r \in \text{AmbRecMatch},$$

and then prove that

$$\forall u \in \text{AmbRecMatch}.\ u \in \text{RecMatch}.$$

Of these two statements, circle the one that would be simpler to prove by structural induction directly from the definitions.

(b) Suppose structural induction was being used to prove that AmbRecMatch $\subseteq$ RecMatch. Circle the one predicate below that would fit the format for a structural induction hypothesis in such a proof.

- $P_0(n) ::= |s| \leq n$ IMPLIES $s \in$ RecMatch.
- $P_1(n) ::= |s| \leq n$ IMPLIES $s \in$ AmbRecMatch.
- $P_2(s) ::= s \in$ RecMatch.
- $P_3(s) ::= s \in$ AmbRecMatch.
- $P_4(s) ::=$ ($s \in$ RecMatch IMPLIES $s \in$ AmbRecMatch).

(c) The recursive definition AmbRecMatch is ambiguous because it allows the $s \cdot t$ constructor to apply when s or t is the empty string. But even fixing that, ambiguity remains. Demonstrate this by giving two different derivations for the string "[][][] according to AmbRecMatch but only using the $s \cdot t$ constructor when $s \neq \lambda$ and $t \neq \lambda$.

Class Problems

Problem 6.14.

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, here's how to erase the string [[[]][]][]:

$$[[[]][]][] \to [[]] \to [] \to \lambda.$$

On the other hand the string [ ]][[[[[]] is not erasable because when we try to erase, we get stuck:][[[:

$$[]][[[[[]] \to][[[[] \to][[[\not\to$$

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 6.2.1

(a) Use structural induction to prove that

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

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

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

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 6.15. **(a)** Prove that the set RecMatch, of matched strings of Definition 6.2.1 is closed under string concatenation. Namely, if $s, t \in$ RecMatch, then $s \cdot t \in$ RecMatch.

(b) Prove AmbRecMatch $\subseteq$ RecMatch, where AmbRecMatch is the set of ambiguous matched strings of Definition 6.2.2.

(c) Prove that RecMatch $=$ AmbRecMatch.

Homework Problems

Problem 6.16.
One way to determine if a string has matching brackets, that is, if it is in the set, RecMatch, of Definition 6.2.1 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.

Problems for Section 6.3

Homework Problems

Problem 6.17.
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 6.4

Practice Problems

Problem 6.18. **(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.

Homework Problems

Problem 6.19. **(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.

v

Problem 6.20.
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. The games we'll consider are known as *deterministic games of perfect information*,

because at each move, the complete game situation is known to the players, and this information completely determines how the rest of the game can be played. Games like chess, checkers, GO, and tic-tac-toe fit this description. In contrast, most card games do not fit, since card players usually do not know exactly what cards belong to the other players. Neither do games involving random features like dice rolls, since a player's move does not uniquely determine what happens next.

Chess counts as a deterministic game of perfect information because at any point of play, both players know whose turn it is to move and the location of every chess piece on the board.[2] At the start of the game, there are 20 possible first moves: the player with the White pieces can move one of his eight pawns forward 1 or 2 squares or one of his two knights forward and left or forward and right. For the second move, the Black player can make one of the 20 corresponding moves of his own pieces. The White player would then make the third move, but now the number of possible third moves depends on what the first two moves happened to be.

A nice way to think of these games is to regard each game situation as a game in its own right. For example, after five moves in a chess game, we think of the players as being at the start of a new "chess" game determined by the current board position and the fact that it is Black's turn to make the next move.

At the end of a chess game, we might assign a score of 1 if the White player won, -1 if White lost, and 0 if the game ended in a stalemate (a tie). Now we can say that White's objective is to maximize the final score and Black's objective is to minimize it. We might also choose to score the game in a more elaborate way, taking into account not only who won, but also how many moves the game took, or the final board configuration.

This leads to an elegant abstraction of this kind of game. We suppose there are two players, called the *max-player* and the *min-player*, whose aim is, respectively, to maximize and minimize the final score. A game will specify its set of possible first moves, each of which will simply be another game. A game with no possible moves is called an *ended game*, and will just have a final score. Strategically, all that matters about an ended game is its score. If a game is not ended, it will have a label `max` or `min` indicating which player is supposed to move first.

This motivates the following formal definition:

Definition. Let V be a nonempty set of real numbers. The class VG of *V-valued deterministic max-min games of perfect information* is defined recursively as fol-

[2] In order to prevent the possibility of an unending game, chess rules specify a limit on the number of moves, or a limit on the number of times a given board postion may repeat. So the number of moves or the number of position repeats would count as part of the game situation known to both players.

lows:

Base case: A value $v \in V$ is a VG, and is called an *ended game*.

Constructor case: If $\{G_0, G_1, \ldots\}$ is a nonempty set of VG's, and a is a label equal to `max` or `min`, then

$$G ::= (a, \{G_0, G_1, \ldots\})$$

is a VG. Each game G_i is called a possible *first move* of G.

In all the games like this that we're familiar with, there are only a finite number of possible first moves. It's worth noting that the definition of VG does not require this. Since finiteness is not needed to prove any of the results below, it would arguably be misleading to assume it. Later, we'll suggest how games with an infinite number of possible first moves might come up.

A *play* of a game is a sequence of legal moves that either goes on forever or finishes with an ended game. More formally:

Definition. A *play* of a game $G \in$ VG is defined recursively on the definition of VG:

Base case: (G is an ended game.) Then the length one sequence (G) is a *play* of G.

Constructor case: (G is not an ended game.) Then a *play* of G is a sequence that starts with a possible first move, G_i, of G and continues with the elements of a play of G_i.

If a play does not go on forever, its *payoff* is defined to be the value it ends with.

Let's first rule out the possibility of playing forever. Namely, every play will have a payoff.

(a) Prove that every play of a $G \in$ VG is a finite sequence that ends with a value in V. *Hint:* By structural induction on the definition of VG.

A *strategy* for a game is a rule that tells a player which move to make when it's his turn. Formally:

Definition. If a is one of the labels `max` or `min`, then an *a-strategy* is a function $s : \text{VG} \to \text{VG}$ such that

$$s(G) \text{ is } \begin{cases} \text{a first move of } G & \text{if } G \text{ has label } a, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Any pair of strategies for the two players determines a unique play of a game, and hence a unique payoff, in an obvious way. Namely, when it is a player's turn to move in a game G, he chooses the move specified by his strategy. A strategy for the max-player is said to *ensure* payoff v when, paired with *any* strategy for the min-player, the resulting payoff is *at least* v. Dually, a strategy for the min-player *caps* payoff at v when, paired with any strategy for the max-player, the resulting payoff is *at most* v.

Assuming for simplicity that the set V of possible values of a game is finite, the WOP (Section 2.4) implies there will be a strategy for the max-player that ensures the largest possible payoff; this is called the *max-ensured-value* of the game. Dually, there will also be a strategy for the min-player that caps the payoff at the smallest possible value, which is called the *min-capped-value* of the game.

The max-ensured-value of course cannot be larger than the min-capped-value. A unique value can be assigned to a game when these two values agree:

Definition. If the max-ensured-value and min-capped-value of a game are equal, their common value is called the *value of the game*.

So if both players play optimally in a game with that has a value, v, then there is actually no point in playing. Since the payoff is ensured to be at least v and is also capped to be at most v, it must be exactly v. So the min-player may as well skip playing and simply pay v to the max-player (a negative payment means the max-player is paying the min-player).

The punch line of our story is that the max-ensured-value and the min-capped-value are *always* equal.

Theorem (Fundamental Theorem for Deterministic Min-Max Games of Perfect Information)**.** *Let V be a finite set of real numbers. Every V-valued deterministic max-min game of perfect information has a value.*

(b) Prove this Fundamental Theorem for VG's by structural induction.

(c) Conclude immediately that in chess, there is a winning strategy for White, or a winning strategy for Black, or both players have strategies that guarantee at least a stalemate. (The only difficulty is that no one knows which case holds.)

So where do we come upon games with an infinite number of first moves? Well, suppose we play a tournament of n chess games for some positive integer n. This tournament will be a VG if we agree on a rule for combining the payoffs of the n individual chess games into a final payoff for the whole tournament.

There still are only a finite number of possible moves at any stage of the n-game chess tournament, but we can 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 has an infinite number of first moves.

Of course only the first move in the meta-chess-tournament is infinite, but then we could set up a tournament consisting of n meta-chess-tournaments. This would be a game with n possible infinite moves. And then we could have a *meta-meta*-chess-tournament whose first move was to choose how many meta-chess-tournaments to play. This meta-meta-chess-tournament will have an infinite number of infinite moves. Then we could move on to meta-meta-meta-chess-tournaments ....

As silly or weird as these meta games may seem, their weirdness doesn't disqualify the Fundamental Theorem: each of these games will still have a value.

(d) State some reasonable generalization of the Fundamental Theorem to games with an infinite set V of possible payoffs. *Optional*: Prove your generalization.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015