Chapter 11

Recursive Data Types

Recursive data types play a central role in programming. From a mathematical point of view, recursive data types are what induction is about. Recursive data types are specified by *recursive definitions* that say how to build something from its parts. These definitions have two parts:

- **Base case(s)** that don't depend on anything else.
- **Constructor case(s)** that depend on previous cases.

11.1 Strings of Brackets

Let `brkts` be the set of all strings of square brackets. For example, the following two strings are in `brkts`:

$$[\,]\,]\,[\,[\,[\,[\,[\,]\,] \quad \text{and} \quad [\,[\,[\,]\,]\,[\,]\,]\,[\,] \tag{11.1}$$

Since we're just starting to study recursive data, just for practice we'll formulate `brkts` as a recursive data type,

Definition 11.1.1. The data type, `brkts`, of strings of brackets is defined recursively:

- **Base case:** The *empty string*, λ, is in `brkts`.
- **Constructor case:** If $s \in$ `brkts`, then $s]$ and $s[$ are in `brkts`.

Here we're writing $s]$ to indicate the string that is the sequence of brackets (if any) in the string s, followed by a right bracket; similarly for $s[$.

A string, $s \in$ `brkts`, 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 straighforward, and you might wonder whether they have any particular relevance in computer scientist —other than as a nonnumerical example of recursion. The honest answer is "not much relevance, *any more*." The reason for this is one of the great successes of computer science.

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. The problem was to take in 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

$$\begin{gathered} [[x + y] * z^2 \div y] + 7, \text{ or,} \\ x + [y * z^2 \div [y + 7]], \text{ or,} \\ [x + [y * z^2]] \div [y + 7], \end{gathered}$$

or . . . ?

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert Floyd, for, among other things, being discoverer of a simple program 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 needed no longer demanded attention. It largely disappeared from the computer science curriculum by the 1990's.

One precise way to determine if a string is matched 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 the two strings above

$$\begin{array}{ccccccccccccc} & [&] &] & [& [& [& [& [&] &] &] &] \\ 0 & 1 & 0 & -1 & 0 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 0 \end{array}$$

$$\begin{array}{ccccccccccc} & [& [& [&] &] & [&] &] & [&] \\ 0 & 1 & 2 & 3 & 2 & 1 & 2 & 1 & 0 & 1 & 0 \end{array}$$

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.

Definition 11.1.2. Let

$$\text{GoodCount} ::= \{s \in \texttt{brkts} \mid s \text{ has a good count}\}.$$

The matched strings can now be characterized precisely as this set of strings with good counts. But it turns out to be really useful to characterize the matched strings in another way as well, namely, as a recursive data type:

Definition 11.1.3. 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 we're writing $[\, s \,]\, t$ to indicate the string that starts with a left bracket, followed by the sequence of brackets (if any) in the string s, followed by a right bracket, and ending with the sequence of brackets in the string t.

Using this definition, we can see that $\lambda \in \text{RecMatch}$ by the Base case, so

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

by the Constructor case. So 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. If you haven't seen this kind of definition before, you should try continuing this example to verify that $[\,[\,[\,]\,]\,[\,]\,]\,[\,] \in \text{RecMatch}$

Given the way this section is set up, you might guess that RecMatch = GoodCount, and you'd be right, but it's not completely obvious. The proof is worked out in Problem 11.6.

11.2 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 11.2.1.

- **Base cases:**

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

- **Constructor cases:** If $e, f \in$ Aexp, then

  3. $(e+f) \in$ 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*.
  4. $(e * f) \in$ 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*.
  5. $-(e) \in$ Aexp. The expression $-(e)$ is called a *negative*.

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

$$((\mathtt{3} * (x * x)) + ((\mathtt{2} * x) + \mathtt{1})). \tag{11.2}$$

These parentheses and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2+2x+1$" instead of (11.2). But it's important to recognize that $3x^2+2x+1$ is not an Aexp; it's an *abbreviation* for an Aexp.

11.3 Structural Induction on Recursive Data Types

Structural induction is a method for proving some property, P, of all the elements of a recursively-defined data type. The proof consists of two steps:

- Prove P for the base cases of the definition.
- Prove P for the constructor cases of the definition, assuming that it is true for the component data items.

A very simple application of structural induction proves that the recursively defined matched strings always have an equal number of left and right brackets. To do this, define a predicate, P, on strings $s \in \mathtt{brkts}$:

$$P(s) ::= \quad s \text{ has an equal number of left and right brackets.}$$

Proof. We'll prove that $P(s)$ holds for all $s \in$ RecMatch by structural induction on the definition that $s \in$ RecMatch, using $P(s)$ as the induction hypothesis.

Base case: $P(\lambda)$ holds because the empty string has zero left and zero right brackets.

Constructor case: For $r = [\ s\ ]\ t$, we must show that $P(r)$ holds, given that $P(s)$ and $P(t)$ holds. So let n_s, n_t be, respectively, the number of left brackets in s and t. So the number of left brackets in r is $1 + n_s + n_t$.

Now from the respective hypotheses $P(s)$ and $P(t)$, we know that the number of right brackets in s is n_s, and likewise, the number of right brackets in t is n_t. So the number of right brackets in r is $1 + n_s + n_t$, which is the same as the number of left brackets. This proves $P(r)$. We conclude by structural induction that $P(s)$ holds for all $s \in$ RecMatch. ■

11.3.1 Functions on Recursively-defined Data Types

Functions on recursively-defined data types can be defined recursively using the same cases as the data type definition. Namely, to define a function, f, on a recursive data type, define the value of f for the base cases of the data type definition, and then define the value of f in each constructor case in terms of the values of f on the component data items.

For example, from the recursive definition of the set, RecMatch, of strings of matched brackets, we define:

Definition 11.3.1. The *depth*, $d(s)$, of a string, $s \in$ RecMatch, is defined recursively by the rules:

- $d(\lambda) ::= 0.$
- $d([\ s\ ]\ t) ::= \max\{d(s) + 1, d(t)\}$

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*. A function defined recursively from an ambiguous definition of a data type will not be well-defined unless the values specified for the different ways of constructing the element agree.

We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on the definition would always be well-defined. As an example of the trouble an ambiguous definition can cause, let's consider yet another definition of the matched strings.

Example 11.3.2. Define the set, $M \subseteq$ `brkts` recursively as follows:

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

Quick Exercise: Give an easy proof by structural induction that $M = \text{RecMatch}$.

Since $M = \text{RecMatch}$, and the definition of M seems more straightforward, why didn't we use it? Because the definition of M is ambiguous, while the trickier definition of RecMatch is unambiguous. Does this ambiguity matter? Yes it does. For suppose we defined

$$\begin{aligned} f(\lambda) &::= 1, \\ f(\,[\,s\,]\,) &::= 1 + f(s), \\ f(st) &::= (f(s)+1)\cdot(f(t)+1) && \text{for } st \neq \lambda. \end{aligned}$$

Let a be the string $[\,[\,]\,] \in M$ built by two successive applications of the first M constructor starting with λ. Next let $b ::= aa$ and $c ::= bb$, each built by successive applications of the second M constructor starting with a.

Alternatively, we can build ba from the second constructor with $s = b$ and $t = a$, and then get to c using the second constructor with $s = ba$ and $t = a$.

Now by these rules, $f(a) = 2$, and $f(b) = (2+1)(2+1) = 9$. This means that $f(c) = f(bb) = (9+1)(9+1) = 100$.

But also $f(ba) = (9+1)(2+1) = 27$, so that $f(c) = f(ba\,a) = (27+1)(2+1) = 84$.

The outcome is that $f(c)$ is defined to be both 100 and 84, which shows that the rules defining f are inconsistent.

On the other hand, structural induction remains a sound proof method even for ambiguous recursive definitions, which is why it was easy to prove that $M = \text{RecMatch}$.

11.3.2 Recursive Functions on Nonnegative Integers

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

Definition 11.3.3. The set, $\mathbb{N}$, is a data type defined recursivly as:

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

This of course makes it clear that ordinary induction is simply the special case of structural induction on the recursive Definition 11.3.3, This also justifies the familiar recursive definitions of functions on the nonnegative integers. Here are some examples.

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

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

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. The nth Fibonacci number, fib, can be defined recursively by:

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

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

Sum-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 $n \geq 0$.

Ill-formed Function Definitions

There are some blunders to watch out for when defining functions recursively. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but they aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{11.3}$$

This "definition" has no base case. If some function, f_1, satisfied (11.3), so would a function obtained by adding a constant to the value of f_1. So equation (11.3) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0,\\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{11.4}$$

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 (11.4) 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 (11.4) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by } 2, \\ 1, & \text{if } n \text{ is divisible by } 3, \\ 2, & \text{otherwise.} \end{cases} \tag{11.5}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (11.5) doesn't define anything.

A Mysterious Function

Mathematicians have been wondering about this function specification for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{11.6}$$

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 (11.6), but it's not known if another function does too. 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 (11.6) equals 1 for all n up to over a billion.

Quick exercise: Why does the constant function 1 satisfy (11.6)?

11.3.3 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 obviously 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 11.3.4. The *evaluation function*, $\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in \text{Aexp}$, as follows. Let n be any integer.

- **Base cases:**

  1. Case[e is x]

     $$\text{eval}(x, n) ::= n.$$

     (The value of the variable, x, is given to be n.)

2. Case[e is k]
$$\text{eval}(\mathtt{k}, n) ::= k.$$
(The value of the numeral $\mathtt{k}$ is the integer k, no matter what value x has.)

- **Constructor cases:**

3. Case[e is $(e_1 + e_2)$]
$$\text{eval}((e_1 + e_2), n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n).$$

4. Case[e is $(e_1 * e_2)$]
$$\text{eval}((e_1 * e_2), n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n).$$

5. Case[e is $-(e_1)$]
$$\text{eval}(-(e_1), n) ::= -\text{eval}(e_1, n).$$

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

$$\begin{aligned}
\text{eval}((\mathtt{3} + (x * x)), 2) &= \text{eval}(\mathtt{3}, 2) + \text{eval}((x * x), 2) && \text{(by Def 11.3.4.3)}\\
&= 3 + \text{eval}((x * x), 2) && \text{(by Def 11.3.4.2)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 11.3.4.4)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 11.3.4.1)}\\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard, important operation. For example the result of substituting the expression $3x$ for x in the $(x(x - 1))$ 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. For instance,

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

This substitution function has a simple recursive definition:

Definition 11.3.5. The *substitution function* from Aexp $\times$ Aexp to Aexp is defined recursively on expressions, $e \in$ Aexp, as follows. Let f be any Aexp.

- **Base cases:**

1. Case[e is x]
$$\text{subst}(f, x) ::= f.$$
(The result of substituting f for the variable, x, is just f.)

2. Case[e is $\texttt{k}$]
$$\text{subst}(f, \texttt{k}) ::= \texttt{k}.$$
(The numeral, k, has no x's in it to substitute for.)

- **Constructor cases:**

3. Case[e is $(e_1 + e_2)$]
$$\text{subst}(f, (e_1 + e_2))) ::= (\text{subst}(f, e_1) + \text{subst}(f, e_2)).$$

4. Case[e is $(e_1 * e_2)$]
$$\text{subst}(f, (e_1 * e_2))) ::= (\text{subst}(f, e_1) * \text{subst}(f, e_2)).$$

5. Case[e is $-(e_1)$]
$$\text{subst}(f, -(e_1)) ::= -(\text{subst}(f, e_1)).$$

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

$$\begin{aligned}
\text{subst}(3x, (x(x-1))) &= \text{subst}(3x, (x * (x + -(1)))) && \text{(unabbreviating)}\\
&= (\text{subst}(3x, x) * \text{subst}(3x, (x + -(1)))) && \text{(by Def 11.3.5 4)}\\
&= (3x * \text{subst}(3x, (x + -(1)))) && \text{(by Def 11.3.5 1)}\\
&= (3x * (\text{subst}(3x, x) + \text{subst}(3x, -(1)))) && \text{(by Def 11.3.5 3)}\\
&= (3x * (3x + -(\text{subst}(3x, 1)))) && \text{(by Def 11.3.5 1 \& 5)}\\
&= (3x * (3x + -(1))) && \text{(by Def 11.3.5 2)}\\
&= 3x(3x-1) && \text{(abbreviation)}
\end{aligned}$$

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. In programming jargon, this would be called evaluation using the *Substitution Model*. Tracing through the steps in the evaluation, we find that the Substitution Model requires two substitutions for occurrences of x and 5 integer operations: 3 integer multiplications, 1 integer addition, and 1 integer negative operation. Note that in this Substitution Model the multiplication $3 \cdot 2$ was performed twice to get the value of 6 for each of the two occurrences of $3x$.

The other approach is called evaluation using the *Environment Model*. Namely, 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 $6 \cdot 5 = 30$. So the Environment Model requires 2 variable lookups and only 4 integer operations: 1

multiplication to find the value of $3x$, another multiplication to find the value $6 \cdot 5$, along with 1 integer addition and 1 integer negative operation.

So the Environment Model approach of calculating

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

instead of the Substitution Model approach of calculating

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

is faster. But how do we know that these final values reached by these two approaches always agree? We can prove this easily by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 11.3.6. *For all expressions $e, f \in$ Aexp and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{11.7}$$

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

Base cases:

- Case[e is x]

  The left hand side of equation (11.7) equals $\text{eval}(f, n)$ by this base case in Definition 11.3.5 of the substitution function, and the right hand side also equals $\text{eval}(f, n)$ by this base case in Definition 11.3.4 of eval.

- Case[e is $\mathtt{k}$].

  The left hand side of equation (11.7) equals $\mathtt{k}$ by this base case in Definitions 11.3.5 and 11.3.4 of the substitution and evaluation functions. Likewise, the right hand side equals $\mathtt{k}$ by two applications of this base case in the Definition 11.3.4 of eval.

Constructor cases:

- Case[e is $(e_1 + e_2)$]

  By the structural induction hypothesis (11.7), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

$$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{11.8}$$

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

$$\text{eval}(\text{subst}(f, (e_1 + e_2)), n) = \text{eval}((e_1 + e_2), \text{eval}(f, n)) \tag{11.9}$$

[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.

But the left hand side of (11.9) equals

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

by Definition 11.3.5.3 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 11.3.4.3 of eval for a sum expression. By induction hypothesis (11.8), 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 (11.9) by Definition 11.3.4.3 of eval for a sum expression. This proves (11.9) in this case.

- e is $(e_1 * e_2)$. Similar.
- e is $-(e_1)$. Even easier.

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

■

11.3.4 Problems

Practice Problems

Problem 11.1.

Definition. Consider a new recursive definition, MB_0, of the same set of "matching" brackets strings as MB (definition of MB is provided in the Appendix):

- **Base case:** $\lambda \in \text{MB}_0$.
- **Constructor cases:**

  (i) If s is in MB_0, then $[s]$ is in MB_0.

  (ii) If $s, t \in \text{MB}_0$, $s \neq \lambda$, and $t \neq \lambda$, then st is in MB_0.

(a) Suppose structural induction was being used to prove that $\text{MB}_0 \subseteq \text{MB}$. 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 $s \in \text{MB}$.
- $P_1(n) ::= |s| \leq n$ IMPLIES $s \in \text{MB}_0$.
- $P_2(s) ::= s \in \text{MB}$.
- $P_3(s) ::= s \in \text{MB}_0$.
- $P_4(s) ::= \ (s \in \text{MB}$ IMPLIES $s \in \text{MB}_0)$.

(b) The recursive definition MB_0 is *ambiguous*. Verify this by giving two different derivations for the string "[ ][ ][ ]" according to MB_0.

Class Problems

Problem 11.2.
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, $\mathrm{id}(x) ::= x$ 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, e^g (the constant e),
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, $\mathrm{id}^{(-1)}$ of the identity function $\mathrm{id}(x)$. The inverse $\mathrm{id}^{(-1)}$ 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 11.3.
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 $S ::= \{2^k 3^m 5^n \mid k, m, n \in \mathbb{N}\}$.

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

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

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 11.4.
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 [[[ ]] [ ]] [ ] :

$$\texttt{[[[]][]][]} \to \texttt{[[]]} \to \texttt{[]} \to \lambda.$$

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

$$\texttt{[]][[[[[]]} \to \texttt{][[[[]} \to \texttt{][[[} \not\to$$

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 11.3.7.

(a) Use structural induction to prove that

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

(b) Supply the missing parts of the following proof that

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

Proof. We prove by induction on the length, n, of strings, x, that if $x \in$ Erasable, then $x \in$ RecMatch. The induction predicate is

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

Base case:

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

Inductive step: To prove $P(n+1)$, suppose $|x| \leq n+1$ and $x \in$ Erasable. We need only show that $x \in$ RecMatch. Now if $|x| < n+1$, then the induction hypothesis, $P(n)$, implies that $x \in$ RecMatch, so we only have to deal with x of length exactly $n+1$.

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 $|y| = n - 1$, 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 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.
- **Subcase**($x = p$[s] t).
  Prove that $x \in$ RecMatch in this subcase.

The proofs of the remaining subcases are just like this last one. **List these remaining subcases.**

This completes the proof by 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,

$$\text{Erasable} \subseteq \text{RecMatch and hence Erasable} = \text{RecMatch}.$$

■

Problem 11.5.

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:**

$$|\langle \texttt{bintree}, G_1, G_2\rangle| ::= |G_1| + |G_2| + 1.$$

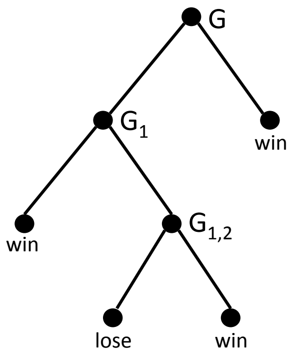


Figure 11.1: *A picture of a binary tree w.*

For example, for the size of the binary-2PTG, G, pictured in Figure 11.1, is 7.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 11.1.

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 11.1,

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

(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

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{11.10}$$

Homework Problems

Problem 11.6.

Definition 11.3.7. The set, RecMatch, of strings of matching brackets, is defined recursively as follows:

- **Base case:** $\lambda \in$ RecMatch.

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

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

There is a simple test to determine whether a string of brackets is in RecMatch: starting with zero, read the string from left to right adding one for each left bracket and -1 for each right bracket. A string has a *good count* when the count never goes negative and is back to zero by the end of the string. Let GoodCount be the bracket strings with good counts.

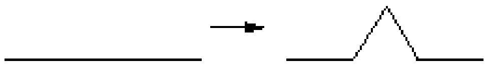
(a) Prove that GoodCount contains RecMatch by structural induction on the definition of RecMatch.

(b) Conversely, prove that RecMatch contains GoodCount.

Problem 11.7.
Fractals are example of a mathematical object 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.

- Recursive 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 below:



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.

11.4 Games as a Recursive Data Type

Chess, Checkers, and Tic-Tac-Toe are examples of *two-person terminating games of perfect information*, —2PTG's for short. These are games in which two players alternate moves that depend only on the visible board position or state of the game. "Perfect information" means that the players know the complete state of the game at each move. (Most card games are *not* games of perfect information because neither player can see the other's hand.) "Terminating" means that play cannot go on forever —it must end after a finite number of moves.[2]

[2]Since board positions can repeat in chess and checkers, termination is enforced by rules that prevent any position from being repeated more than a fixed number of times. So the "state" of these games is the board position *plus* a record of how many times positions have been reached.

We will define 2PTG's as a recursive data type. To see how this will work, let's use the game of Tic-Tac-Toe as an example.

11.4.1 Tic-Tac-Toe

Tic-Tac-Toe is a game for young children. There are two players who alternately write the letters "X" and "O" in the empty boxes of a 3×3 grid. Three copies of the same letter filling a row, column, or diagonal of the grid is called a *tic-tac-toe*, and the first player who gets a tic-tac-toe of their letter wins the game.

We're now going give a precise mathematical definition of the Tic-Tac-Toe *game tree* as a recursive data type.

Here's the idea behind the definition: at any point in the game, the "board position" is the pattern of X's and O's on the 3×3 grid. From any such Tic-Tac-Toe pattern, there are a number of next patterns that might result from a move. For example, from the initial empty grid, there are nine possible next patterns, each with a single X in some grid cell and the other eight cells empty. From any of these patterns, there are eight possible next patterns gotten by placing an O in an empty cell. These move possibilities are given by the game tree for Tic-Tac-Toe indicated in Figure 11.2.

Definition 11.4.1. A Tic-Tac-Toe *pattern* is a 3×3 grid each of whose 9 cells contains either the single letter, X, the single letter, O, or is empty.

A pattern, Q, is a *possible next pattern after* P, providing P has no tic-tac-toes and

- if P has an equal number of X's and O's, and Q is the same as P except that a cell that was empty in P has an X in Q, or

- if P has one more X than O's, and Q is the same as P except that a cell that was empty in P has an O in Q.

If P is a Tic-Tac-Toe pattern, and P has no next patterns, then the *terminated Tic-Tac-Toe game trees* at P are

- $\langle P, \langle \texttt{win} \rangle \rangle$, if P has a tic-tac-toe of X's.

- $\langle P, \langle \texttt{lose} \rangle \rangle$, if P has a tic-tac-toe of O's.

- $\langle P, \langle \texttt{tie} \rangle \rangle$, otherwise.

The *Tic-Tac-Toe game trees starting at* P are defined recursively:

Base Case: A terminated Tic-Tac-Toe game tree at P is a Tic-Tac-Toe game tree starting at P.

Constructor case: If P is a non-terminated Tic-Tac-Toe pattern, then the Tic-Tac-Toe game tree starting at P consists of P and the set of all game trees starting at possible next patterns after P.

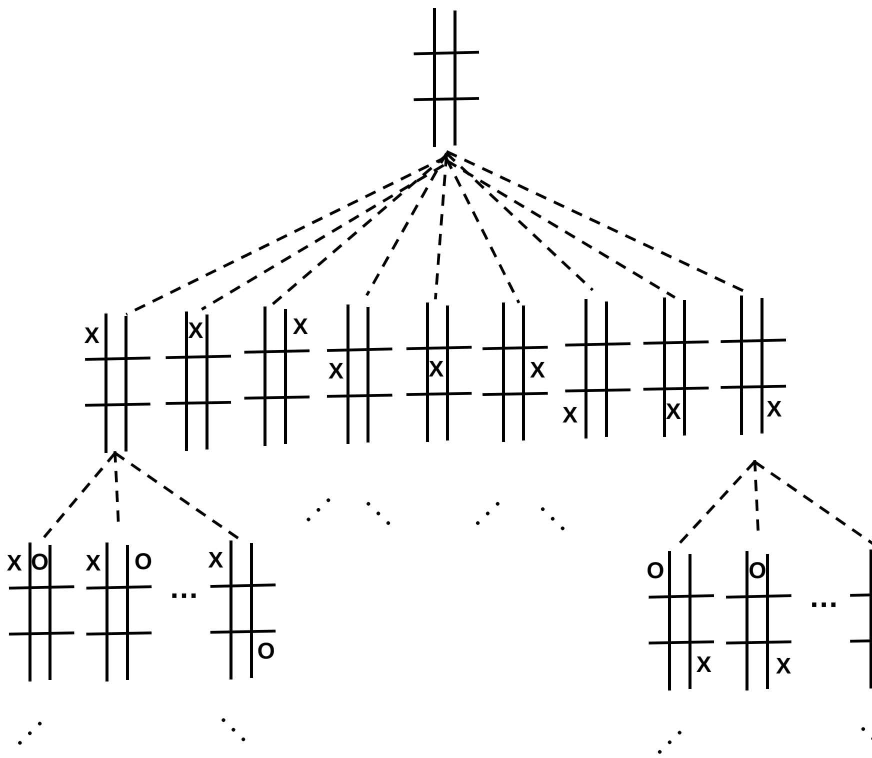


Figure 11.2: The Top of the Game Tree for Tic-Tac-Toe.

For example, if

$$P_0 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & & \end{array}$$

$$Q_1 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & & O \end{array}$$

$$Q_2 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & O & \end{array}$$

$$R = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & O & X \end{array}$$

the game tree starting at P_0 is pictured in Figure 11.3.

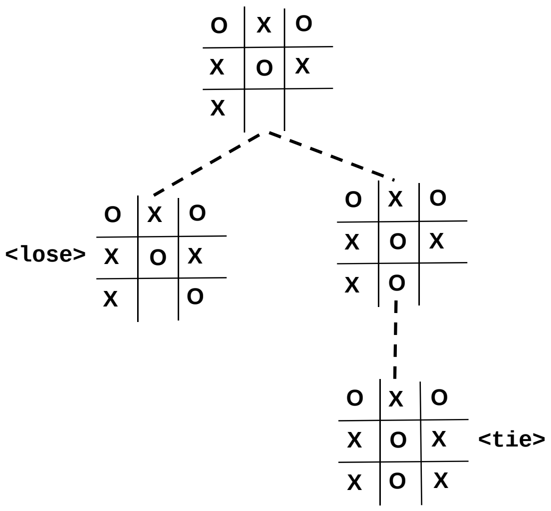


Figure 11.3: Game Tree for the Tic-Tac-Toe game starting at P_0.

Game trees are usually pictured in this way with the starting pattern (referred to as the "root" of the tree) at the top and lines connecting the root to the game trees that start at each possible next pattern. The "leaves" at the bottom of the tree (trees grow upside down in computer science) correspond to terminated games. A path from the root to a leaf describes a complete *play* of the game. (In English, "game" can be used in two senses: first we can say that Chess is a game, and second we can play a game of Chess. The first usage refers to the data type of Chess game trees, and the second usage refers to a "play.")

11.4.2 Infinite Tic-Tac-Toe Games

At any point in a Tic-Tac-Toe game, there are at most nine possible next patterns, and no play can continue for more than nine moves. But we can expand Tic-Tac-Toe into a larger game by running a 5-game tournament: play Tic-Tac-Toe five times and the tournament winner is the player who wins the most individual games. A 5-game tournament can run for as many as 45 moves.

It's not much of generalization to have an *n-game Tic-Tac-Toe tournament*. But then comes a generalization that sounds simple but can be mind-boggling: consolidate all these different size tournaments into a single game we can call *Tournament-Tic-Tac-Toe (T^4)*. The first player in a game of T^4 chooses any integer $n > 0$. Then the players play an n-game tournament. Now we can no longer say how long a T^4 play can take. In fact, there are T^4 plays that last as long as you might like: if you want a game that has a play with, say, nine billion moves, just have the first player choose n equal to one billion. This should make it clear the game tree for T^4 is infinite.

But still, it's obvious that every possible T^4 play will stop. That's because after the first player chooses a value for n, the game can't continue for more than $9n$ moves. So it's not possible to keep playing forever even though the game tree is infinite.

This isn't very hard to understand, but there is an important difference between any given n-game tournament and T^4: even though every play of T^4 must come to an end, there is no longer any initial bound on how many moves it might be before the game ends —a play might end after 9 moves, or $9(2001)$ moves, or $9(10^{10} + 1)$ moves. It just can't continue forever.

Now that we recognize T^4 as a 2PTG, we can go on to a *meta-T^4* game, where the first player chooses a number, $m > 0$, of T^4 games to play, and then the second player gets the first move in each of the individual T^4 games to be played.

Then, of course, there's meta-meta-T^4....

11.4.3 Two Person Terminating Games

Familiar games like Tic-Tac-Toe, Checkers, and Chess can all end in ties, but for simplicity we'll only consider win/lose games —no "everybody wins"-type games at MIT. `:-)`. But everything we show about win/lose games will extend easily to games with ties, and more generally to games with outcomes that have different payoffs.

Like Tic-Tac-Toe, or Tournament-Tic-Tac-Toe, the idea behind the definition of 2PTG's as a recursive data type is that making a move in a 2PTG leads to the start of a subgame. In other words, given any set of games, we can make a new game whose first move is to pick a game to play from the set.

So what defines a game? For Tic-Tac-Toe, we used the patterns and the rules of Tic-Tac-Toe to determine the next patterns. But once we have a complete game tree, we don't really need the pattern labels: the root of a game tree itself can play the role of a "board position" with its possible "next positions" determined by the roots of its subtrees. So any game is defined by its game tree. This leads to the following very simple —perhaps deceptively simple —general definition.

Definition 11.4.2. The 2PTG, *game trees for two-person terminating games of perfect information* are defined recursively as follows:

- **Base cases:**
$$\begin{aligned}\langle \texttt{leaf}, \texttt{win} \rangle &\in \text{2PTG, and} \\ \langle \texttt{leaf}, \texttt{lose} \rangle &\in \text{2PTG.}\end{aligned}$$

- **Constructor case:** If $\mathcal{G}$ is a nonempty set of 2PTG's, then G is a 2PTG, where
$$G ::= \langle \texttt{tree}, \mathcal{G} \rangle .$$
The game trees in $\mathcal{G}$ are called the possible *next moves* from G.

These games are called "terminating" because, even though a 2PTG may be a (very) infinite datum like Tournament^2-Tic-Tac-Toe, every play of a 2PTG must terminate. This is something we can now prove, after we give a precise definition of "play":

Definition 11.4.3. A *play* of a 2PTG, G, is a (potentially infinite) sequence of 2PTG's starting with G and such that if G_1 and G_2 are consecutive 2PTG's in the play, then G_2 is a possible next move of G_1.

If a 2PTG has no infinite play, it is called a *terminating* game.

Theorem 11.4.4. *Every 2PTG is terminating.*

Proof. By structural induction on the definition of a 2PTG, G, with induction hypothesis
$$G \text{ is terminating.}$$

Base case: If $G = \langle \texttt{leaf}, \texttt{win} \rangle$ or $G = \langle \texttt{leaf}, \texttt{lose} \rangle$ then the only possible play of G is the length one sequence consisting of G. Hence G terminates.

Constructor case: For $G = \langle \texttt{tree}, \mathcal{G} \rangle$, we must show that G is terminating, given the Induction Hypothesis that *every* $G' \in \mathcal{G}$ is terminating.

But any play of G is, by definition, a sequence starting with G and followed by a play starting with some $G_0 \in \mathcal{G}$. But G_0 is terminating, so the play starting at G_0 is finite, and hence so is the play starting at G.

This completes the structural induction, proving that every 2PTG, G, is terminating. ■

11.4.4 Game Strategies

A key question about a game is whether a player has a winning strategy. A *strategy* for a player in a game specifies which move the player should make at any point in the game. A *winning* strategy ensures that the player will win no matter what moves the other player makes.

In Tic-Tac-Toe for example, most elementary school children figure out strategies for both players that each ensure that the game ends with no tic-tac-toes, that is, it ends in a tie. Of course the first player can win if his opponent plays childishly, but not if the second player follows the proper strategy. In more complicated games like Checkers or Chess, it's not immediately clear that anyone has a winning strategy, even if we agreed to count ties as wins for the second player.

But structural induction makes it easy to prove that in any 2PTG, *somebody* has the winning strategy!

Theorem 11.4.5. ***Fundamental Theorem for Two-Person Games:*** *For every two-person terminating game of perfect information, there is a winning strategy for one of the players.*

Proof. The proof is by structural induction on the definition of a 2PTG, G. The induction hypothesis is that there is a winning strategy for G.

Base cases:

1. $G = \langle \texttt{leaf}, \texttt{win} \rangle$. Then the first player has the winning strategy: "make the winning move."

2. $G = \langle \texttt{leaf}, \texttt{lose} \rangle$. Then the second player has a winning strategy: "Let the first player make the losing move."

Constructor case: Suppose $G = \langle \texttt{tree}, \mathcal{G} \rangle$. By structural induction, we may assume that some player has a winning strategy for each $G' \in \mathcal{G}$. There are two cases to consider:

- some $G_0 \in \mathcal{G}$ has a winning strategy for its second player. Then the first player in G has a winning strategy: make the move to G_0 and then follow the second player's winning strategy in G_0.

- every $G' \in \mathcal{G}$ has a winning strategy for its first player. Then the second player in G has a winning strategy: if the first player's move in G is to $G_0 \in \mathcal{G}$, then follow the winning strategy for the first player in G_0.

So in any case, one of the players has a winning strategy for G, which completes the proof of the constructor case.

It follows by structural induction that there is a winning strategy for every 2PTG, G. ■

Notice that although Theorem 11.4.5 guarantees a winning strategy, its proof gives no clue which player has it. For most familiar 2PTG's like Chess, Go, . . . , no one knows which player has a winning strategy.[3]

[3]Checkers used to be in this list, but there has been a recent announcement that each player has a

11.4.5 Problems

Homework Problems

Problem 11.8.
Define *2-person 50-point games of perfect information*50-PG's, recursively as follows:

Base case: An integer, k, is a 50-PG for $-50 \leq k \leq 50$. This 50-PG called the *terminated game with payoff k*. A *play* of this 50-PG is the length one integer sequence, k.

Constructor case: If $G_0, \ldots, G_n$ is a finite sequence of 50-PG's for some $n \in \mathbb{N}$, then the following game, G, is a 50-PG: the possible first moves in G are the choice of an integer i between 0 and n, the possible second moves in G are the possible first moves in G_i, and the rest of the game G proceeds as in G_i.

A *play* of the 50-PG, G, is a sequence of nonnegative integers starting with a possible move, i, of G, followed by a play of G_i. If the play ends at the game terminated game, k, then k is called the *payoff* of the play.

There are two players in a 50-PG who make moves alternately. The objective of one player (call him the *max*-player) is to have the play end with as high a payoff as possible, and the other player (called the *min*-player) aims to have play end with as low a payoff as possible.

Given which of the players moves first in a game, a strategy for the max-player is said to *ensure* the payoff, k, if play ends with a payoff of at least k, no matter what moves the min-player makes. Likewise, a strategy for the min-player is said to *hold down* the payoff to k, if play ends with a payoff of at most k, no matter what moves the max-player makes.

A 50-PG is said to have *max value*, k, if the max-player has a strategy that ensures payoff k, and the min-player has a strategy that holds down the payoff to k, when the *max-player moves first*. Likewise, the 50-PG has *min value*, k, if the max-player has a strategy that ensures k, and the min-player has a strategy that holds down the payoff to k, when the *min-player moves first*.

The *Fundamental Theorem* for 2-person 50-point games of perfect information is that is that every game has both a max value and a min value. (Note: the two values are usually different.)

What this means is that there's no point in playing a game: if the max player gets the first move, the min-player should just pay the max-player the max value of the game without bothering to play (a negative payment means the max-player is paying the min-player). Likewise, if the min-player gets the first move, the min-player should just pay the max-player the min value of the game.

(a) Prove this Fundamental Theorem for 50-valued 50-PG's by structural induction.

(b) A meta-50-PG game has as possible first moves the choice of *any* 50-PG to play. Meta-50-PG games aren't any harder to understand than 50-PG's, but there is one notable difference, they have an infinite number of possible first moves. We could also define meta-meta-50-PG's in which the first move was a choice of any

strategy that forces a tie. (reference TBA)

50-PG *or* the meta-50-PG game to play. In meta-meta-50-PG's there are an infinite number of possible first *and* second moves. And then there's meta^3 − 50-PG ....

To model such infinite games, we could have modified the recursive definition of 50-PG's to allow first moves that choose any one of an infinite sequence

$$G_0, G_1, \ldots, G_n, G_{n+1}, \ldots$$

of 50-PG's. Now a 50-PG can be a mind-bendingly infinite datum instead of a finite one.

Do these infinite 50-PG's still have max and min values? In particular, do you think it would be correct to use structural induction as in part (a) to prove a Fundamental Theorem for such infinite 50-PG's? Offer an answer to this question, and briefly indicate why you believe in it.

11.5 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 a awful lot of things, including all step-by-step computational processes.

Structural induction then goes beyond natural number counting by offering a simple, natural approach to proving things about recursive computation and recursive data types. This makes it a technique every computer scientist should embrace.