

Recursive Definitions and Structural Induction

1 Recursive Definitions

Recursive definitions say how to build something from a simpler version of the same thing. They have two parts:

- Base case(s) that don't depend on anything else.
- Combination case(s) that depend on simpler cases.

Here are some examples of recursive definitions:

Example 1.1. Define a set, E , recursively as follows:

1. $0 \in E$,
2. if $n \in E$, then $n + 2 \in E$,
3. if $n \in E$, then $-n \in E$.

Using this definition, we can see that since $0 \in E$ by 1., it follows from 2. that $0 + 2 = 2 \in E$, and so $2 + 2 = 4 \in E$, $4 + 2 = 6 \in E$, \dots , and in fact any nonnegative even number is in E . Also, by 3., $-2, -4, -6, \dots \in E$.

Is anything else in E ? The definition doesn't say so explicitly, but an implicit condition on a recursive definition is that the only way things get into E is as a consequence of 1., 2., and 3. So clearly E is exactly the set of even integers.

Example 1.2. Define a set, S , of strings of a's and b's recursively as follows:

1. $\lambda \in S$, where λ is the *empty* string,
2. if $x \in S$, then $axb \in S$,
3. if $x \in S$, then $bxa \in S$,
4. if $x, y \in S$, then $xy \in S$.

Here we're writing xy to indicate the *concatenation* of the strings x and y , namely, xy is the string that starts with the sequence of a's and b's in x followed by the a's and b's in y .

Using this definition, we can see that since $\lambda \in S$ by 1., it follows from 2. that $a\lambda b = ab \in S$, so $aabb \in S$ by 2. and $baba \in S$ by 3. Likewise, $b\lambda a = ba \in S$ by 3., so $abab \in S$ by 2. and $bbaa \in S$ by 3. Also, since $ab \in S$ and $ba \in S$, we have $abba \in S$ as well as $asbaab \in S$ by 4.

Notice that every string in S has an equal number of a's and b's. This is easy to prove by induction on how a string gets to be in S .

Definition 1.3.

$$L ::= \{x \in \{a, b\}^* \mid \#a\text{'s in } x = \#b\text{'s in } x\}.$$

Lemma 1.4.

$$S \subseteq L.$$

Proof. Let $P(n)$ be the predicate that if any string, s , is in S as a consequence of n applications of rules 2., 3., and 4., then $s \in L$. We will prove that $P(n)$ holds for all $n \in \mathbb{N}$ by strong induction.

Base case ($n = 0$). The only way to get an $s \in S$ without using any of the rules 2., 3., and 4., is by rule 1., so $s = \lambda$ and s indeed has an equal number of a's and b's, namely zero of each.

Inductive step ($n \geq 0$). Assume $P(0), \dots, P(n)$ to prove $P(n + 1)$.

So suppose $s \in S$ as a consequence of $n + 1$ applications of rules 2., 3., and 4.

Case 1: The $n + 1$ st rule was 2. That is $s = axb$ for some x that is in S as a consequence of n applications of 2., 3., and 4. By induction hypothesis, x has an equal number of a's and b's, say k of each. Then, s also has an equal number of a's and b's, namely $k + 1$ of each, so $s \in L$.

Case 2: The $n + 1$ st rule was 3. Symmetric to Case 1.

Case 3: The $n + 1$ st rule was 4. So $s = xy$ where x and y are each in S as a consequence of at most n applications of 2., 3., and 4. By induction hypothesis, x has the same number, say k_x , of a's and b's, and y has the same number, say k_y , of a's and b's. So xy has an equal number of a's and b's, namely $k_x + k_y$ of each, and again $s \in L$ as required. \square

It's also not hard to show that every string with an equal number of a's and b's is in S :

Problem 1. Prove that $L = S$.

Example 1.5. The set, F , of *Fully Bracketed Arithmetic Expressions in x* is a set of strings over the alphabet $\{], [, + , - , * , \mathbf{x} \}$ defined recursively as follows:

1. The symbols $0, 1, \mathbf{x}$ are in F .
2. If e and e' are in F , then the string $[e+e']$ is in F ,
3. if e and e' are in F then the string $[e*e']$ is in F ,

4. if e is in F then the string $[-e]$ is in F .

Several basic examples of recursively defined data types are based on *rooted trees*. These are possibly infinite directed trees, $T = (V_T, E_T)$, with a necessarily unique “root” vertex $\text{root}(T)$, such that every vertex is reachable by a directed path from the root. Finite-Path Trees from [Week 5 Notes](#), for example, are the class of rooted trees which do not have any infinite directed path from the root.

Another important class of trees are the *ordered binary trees*. These are possibly infinite rooted trees with labelled edges, such that at most two edges leave each vertex. If two edges leave a vertex, one is labelled `left` and the other is labelled `right`. If one edge leaves a vertex, it is labelled either `left` or `right`.

Example 1.6. We will define a special class of ordered binary trees called the *recursive ordered binary trees*, `RecBinT`:

1. If G is a graph with one vertex and no edges, then G is a `RecBinT`. That is, $(\{v\}, \emptyset) \in \text{RecBinT}$ and $\text{root}(\{v\}, \emptyset) = v$.
2. If $T = (V, E)$ is in `RecBinT`, and \mathbf{n} is a “new” node not in V , then the graph, $\text{makeleft}(T)$, made by adding an edge labelled `left` from \mathbf{n} to $\text{root}(T)$ is a `RecBinT`. That is,

$$\text{makeleft}(T) ::= (V \cup \{\mathbf{n}\}, E \cup \{(\mathbf{n}, \text{root}(T), \text{left})\}) \in \text{RecBinT},$$

where (v, w, ℓ) is the directed edge from vertex v to vertex w with label ℓ .

3. Same as above, with “right” in place of “left.”
4. If $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are in `RecBinT`, V_1 and V_2 are disjoint, and \mathbf{n} is a “new” node not in $V_1 \cup V_2$, then the graph, $\text{makeboth}(T_1, T_2)$, made by adding an edge labelled `left` from \mathbf{n} to $\text{root}(T_1)$ and an edge labelled `right` from \mathbf{n} to $\text{root}(T_2)$ is a `RecBinT`. That is,

$$\text{makeboth}(T_1, T_2) ::= (V_1 \cup V_2 \cup \{\mathbf{n}\}, E_1 \cup E_2 \cup \{(\mathbf{n}, \text{root}(T_1), \text{left}), (\mathbf{n}, \text{root}(T_2), \text{right})\}) \in \text{RecBinT}.$$

These cases are illustrated in [Figure 1](#), with edges labelled “left” shown going down to the left, and edges labelled “right” shown going down to the right.

A special case of `RecBinT`'s are the *full* ordered binary trees, `FullBinT`. These are `RecBinT`'s in which every node is either a *leaf* (i.e., has out-degree zero) or has both a left and right subtree (see [Figure 2](#)). In other words rules 2. and 3. in the definition of `RecBinT` are not used in defining the subset `FullBinT` \subset `RecBinT`.

Note that `RecBinT` is precisely the set of *finite* ordered binary trees. Properly speaking, we should prove this. Namely, we should prove that every `RecBinT` is finite, and prove that every finite ordered binary tree is an `RecBinT`. We'll hold off on this, but will do a more interesting proof of this kind for infinite trees in [Theorem 3.2](#) below.

We can generalize `RecBinT`'s to the class of *Countable Ordered Trees*, `CT`, in which each vertex may have any finite number, $n \geq 0$, of edges consecutively labelled $1, 2, \dots, n$ leaving it, or may even have an infinite set of edges consecutively labelled with all the positive integers.

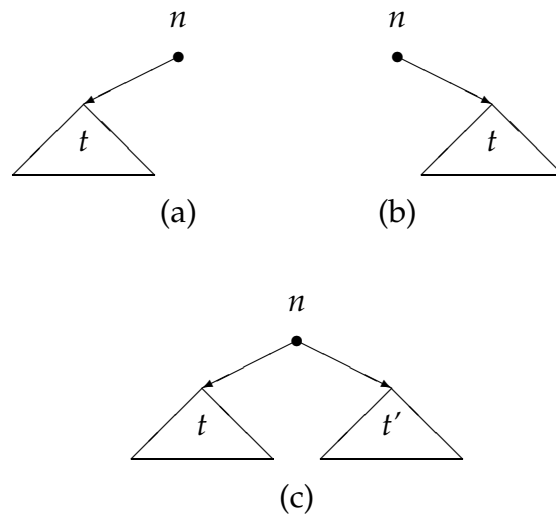


Figure 1: Building a binary tree: (a) *makeleft*(T), (b) *makeright*(T), and (c) *makeboth*(T_1, T_2).

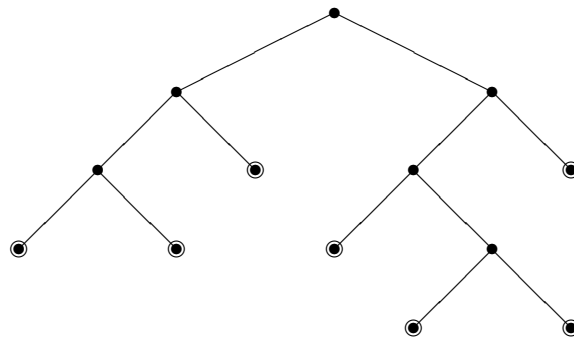


Figure 2: A full binary tree.

Example 1.7. We recursively define the set, RecCT, of *Recursive Countable Trees* as follows:

1. If G is a graph with one vertex and no edges, then G is a RecCT.
2. If $T_1 = (V_1, E_1), T_2 = (V_2, E_2), \dots$ is a finite or infinite sequence of RecCT's, such that V_i and V_j are disjoint for all $i \neq j$, and \mathbf{n} is a "new" node not in $\bigcup_1^\infty V_i$, then the graph, $\text{maketree}(T_1, T_2, \dots)$, made by adding an edge labelled i from \mathbf{n} to $\text{root}(T_i)$ for all $i > 0$ is a RecCT. That is,

$$\text{maketree}(T_1, T_2, \dots) ::= \left(\bigcup_1^\infty V_i \cup \{\mathbf{n}\}, \bigcup_1^\infty E_i \cup \{(\mathbf{n}, \text{root}(T_i), i) \mid i \in \mathbb{Z}^+\} \right) \in \text{RecCT}.$$

Example 1.8. The set, List, of *pure lists* is defined recursively by:

1. The 0-tuple, $()$, is in List.
2. If ℓ_1 and ℓ_2 are in List, then the pair (ℓ_1, ℓ_2) is in List.

In Lisp-like programming languages, the pairing operation is called `cons` and the 0-tuple is called `nil`.

2 Structural Induction on Recursive Data Type Definitions

Structural induction is used to prove a property P of all the elements of some recursively-defined data type. The proof consists of two steps:

- Prove P for the "base cases" of the definition.
- Prove P for the result of any combination rule, assuming that it is true for all the parts.

For example, structural induction on Fully Bracketed Arithmetic Expressions takes the form:

Proof. To prove $\forall e \in F P(e)$, show:

Base ($e = \mathbf{0}$). $P(\mathbf{0})$ holds.

Base ($e = \mathbf{1}$). $P(\mathbf{1})$ holds.

Base ($e = \mathbf{x}$). $P(\mathbf{x})$ holds.

Inductive step ($\llbracket e+e' \rrbracket$). Assume $P(e)$ and $P(e')$ to prove $P(\llbracket e+e' \rrbracket)$.

Inductive step ($\llbracket e*e' \rrbracket$). Assume $P(e)$ and $P(e')$ to prove $P(\llbracket e*e' \rrbracket)$.

Inductive step ($\llbracket -e \rrbracket$). Assume $P(e)$ to prove $P(\llbracket -e \rrbracket)$. □

Here's an actual example:

Theorem 2.1. *Every Fully Bracketed Arithmetic Expression has the same number of left and right brackets.*

Proof. This is just like the proof of Lemma 1.4 above, except we proved Lemma 1.4 by ordinary strong induction explicitly on the number of rules used to construct on element. Here we use structural induction, which lets us carry out the proof without having to count rule applications.

Define $P(e) ::=$ expression e has the same number of left brackets, $[$, and right brackets, $]$.

Base Cases ($e = \mathbf{0}$, $e = \mathbf{1}$, or $e = \mathbf{x}$). The expression e has no brackets.

Inductive step ($[e+e']$). Assume $P(e)$ and $P(e')$ to prove $P([e+e'])$.

But $P(e)$ implies e has k_e left brackets and k_e right brackets for some $k_e \in \mathbb{N}$; likewise e' has $k_{e'}$ left brackets and $k_{e'}$ right brackets. So $[e+e']$ has $k_e + k_{e'} + 1$ right brackets and the same number of left brackets.

Inductive step ($[e*e']$). Similar.

Inductive step ($[-e]$). Similar. □

3 Induction on Trees

Here's a proof using structural induction on binary trees:

Theorem 3.1. *The number of edges in any RecBinT is exactly one fewer than the number of vertices.*

Proof. Define

$$P(T) ::= T = (V, E) \in \text{RecBinT} \text{ and } |V| - 1 = |E|.$$

Base Case (T is the single-node tree). There are 0 edges and 1 node, so $P(T)$ holds.

Inductive step ($T = \text{makeleft}(S)$). Assume $P(S)$ to prove $P(T)$. That is, assuming

$$|V_S| - 1 = |E_S|, \tag{1}$$

show that

$$|V_T| - 1 = |E_T| \tag{2}$$

But

$$\begin{aligned} E_T &= E_S \cup \{(\mathbf{n}, \text{root}S, \text{left})\} && \text{so,} \\ |E_T| &= |E_S| + 1, \end{aligned} \tag{3}$$

and

$$\begin{aligned} |E_T| &= |V_S|, && \text{(by (3) and (1)), so,} \\ |E_T| + 1 &= |V_S| + 1. \end{aligned} \tag{4}$$

But

$$\begin{aligned} V_T &= V_S \cup \{\mathbf{n}\}, && \text{so} \\ |V_T| &= |V_S| + 1 \end{aligned} \tag{5}$$

Combining (5) and (4) immediately implies (2).

Inductive step ($T = \text{makeright}(S)$). Same as previous case with “right” replacing “left.”

Inductive step ($T = \text{makeboth}(T_1, T_2)$). Assume $P(T_1)$ and $P(T_2)$ to prove $P(T)$. That is, assuming

$$|E_{T_i}| = |V_{T_i}| - 1, \quad (6)$$

for $i = 1, 2$, prove

$$|E_T| = |V_T| - 1, \quad (7)$$

The similar proof for this case is left to the reader. \square

Structural induction is a correct and useful proof method on recursive data types even when they are infinite. For example, using structural induction on the definition of RecCT’s, we can easily prove that every RecCT is a Finite-path Countable Ordered Tree.

Theorem 3.2. *Every RecCT is a [Finite-Path CT](#).*

Proof. Define

$$P(T) ::= T \text{ is a Finite-path CT.}$$

We prove that $P(T)$ holds for all $T \in \text{RecCT}$ by Structural Induction on the definition of RecCT.

Base case (T has one vertex) T is a CT by definition of CT. There is only an “empty” of length zero in T . So T is also Finite-path.

Inductive step ($T = \text{maketree}(T_1, T_2, \dots)$). By structural induction hypothesis, we may assume that $P(T_i)$ holds for all T_i .

By definition of T , the edges from the root of T are labelled with consecutive integers. Any other vertex of T is a vertex in V_i which is labelled with consecutive integers because $T_i \in \text{CT}$ by hypothesis. Hence $T \in \text{CT}$.

Also by definition of T , the second node on any path from the root of T must be the root some T_i . The rest of the path is a directed path from the root of T_i , and so is finite because T_i is Finite-path by induction hypothesis. So the whole path from the root of T is also finite. \square

This proof by Structural Induction may seem obvious, but it is actually different in character from all the other Structural Induction proofs. Namely, all the other proofs could be reformulated as ordinary induction on the number of rule applications used in constructing an element of the recursive data type. But because a Recursive Countable Tree can be built by combining an infinite sequence of subtrees, which in total already take an infinite number of rule applications, the number of rule applications to construct a single RecCT may not be finite. So ordinary induction on the number of rule applications to construct a recursively defined object won’t work. As a matter of fact, Mathematical Logicians have proved that this kind structural induction on infinite data objects is actually *strictly more powerful* than ordinary induction. But nevertheless, Structural

Induction is a completely correct proof method, even for data types like RecCT that build up elements in an infinite number of steps.

Problem 2. Prove the converse of Theorem 3.2:

Theorem 3.3. *Every Finite-path CT is a RCT.*

Hint: Suppose not, and consider a minimal Finite-path CT (under the well-founded “strict-subtree” partial order on Finite-path trees defined in Week 5 Notes) that is not in RecCT.

4 Recursively-defined Functions on Recursively-defined Data Types

4.1 Some Recursively Defined Functions

Recursive definitions provide a natural way to define functions whose domains are recursively-defined data types.

Example 4.1. (Binary Trees)

Define the function $\text{numnodes}(T)$ (a recursive definition of $|V_T|$ for a RecBinT as follows:

1. $\text{numnodes}(\text{single node}) ::= 1,$
2. $\text{numnodes}(\text{makeleft}(S)) ::= \text{numnodes}(S) + 1,$
3. $\text{numnodes}(\text{makeright}(S)) ::= \text{numnodes}(S) + 1,$
4. $\text{numnodes}(\text{makeboth}(T_1, T_2)) ::= \text{numnodes}(T_1) + \text{numnodes}(T_2) + 1.$

Similarly, define $\text{numedges}(T)$:

1. $\text{numedges}(\text{single node}) ::= 0,$
2. $\text{numedges}(\text{makeleft}(S)) ::= \text{numedges}(S) + 1,$
3. $\text{numedges}(\text{makeright}(S)) ::= \text{numedges}(S) + 1,$
4. $\text{numedges}(\text{makeboth}(T_1, T_2)) ::= \text{numedges}(T_1) + \text{numedges}(T_2) + 2.$

Example 4.2. (Arithmetic Expressions) Given a value, n , for the variable, x , we can easily calculate the numerical value of any expression, e , in the set, F , of Fully Bracketed Arithmetic Expressions in x . The function, $\text{eval}(e, n)$, giving the value of expression e when x has value n has a simple recursive definition based on the definition of the recursive data type F . Namely, define

1. $\text{eval}(\mathbf{0}, n) ::= 0,$
2. $\text{eval}(\mathbf{1}, n) ::= 1,$

3. $\text{eval}(\mathbf{x}, n) ::= n,$
4. $\text{eval}(\mathbf{[e+e']}, n) ::= \text{eval}(e, n) + \text{eval}(e', n),$
5. $\text{eval}(\mathbf{[e*e']}, n) ::= \text{eval}(e, n) \cdot \text{eval}(e', n),$
6. $\text{eval}(\mathbf{[-e]}, n) = -\text{eval}(e, n).$

Another useful operation on arithmetic expressions is substituting one into another. Let $\text{subst}(e, f)$ be the result of substituting expression f for all occurrences of \mathbf{x} in e . The function subst also has a simple definition based on the definition of the recursive data type F . Namely, define $\text{subst}(e, f)$ recursively in e as follows:

1. $\text{subst}(\mathbf{0}, f) ::= 0,$
2. $\text{subst}(\mathbf{1}, f) ::= 1,$
3. $\text{subst}(\mathbf{x}, f) ::= f,$
4. $\text{subst}(\mathbf{[e+e']}, f) ::= \mathbf{[subst(e, f)+subst(e', f)]},$
5. $\text{subst}(\mathbf{[e*e']}, f) ::= \mathbf{[subst(e, f)*subst(e', f)]},$
6. $\text{subst}(\mathbf{[-e]}, f) = \mathbf{[-subst(e, f)]}.$

4.2 Recursive Function Definitions

In general, we can define a function, f , on a recursively defined data type by defining f on each of the elements in the base cases of the definition. Then define $f(d)$ in terms of $f(d_1), f(d_2), \dots$ where d is an element built from elements d_1, d_2, \dots by a combination rule. One warning though: f is only guaranteed to be well-defined if each element, d , can be constructed in *only one way*: only by a unique combination rule applied to unique elements d_1, d_2, \dots . So this kind of recursive definition will work fine for Fully Bracketed Arithmetic Expressions, binary trees, lists, and recursive trees, but might *not* yield a well-defined function if the definition of f was based on the recursive definition we gave for the set, S , of strings with an equal number of a's and b's. The reason is that some strings in S can be constructed in more than one way.

Problem 3. (a) What is the smallest string in S which can be constructed in two different ways using Definition 1.4 of S above?

(b) Find a recursive definition of S similar to the one above, but under which every string in S is constructed in exactly one way.

4.3 Proving Properties of Recursive Functions

When we have a recursive definition of a function on a recursive data type, we can use structural induction to prove properties of the function. We already did this in proving Theorem 3.1 relating the number of edges and nodes in a binary tree. A more interesting example using structural induction on the definition of Fully Bracketed Arithmetic Expressions to prove a fundamental relationship between numerical and symbolic calculation.

Namely, suppose we have an arithmetic expression e with variable x and we substitute another expression, f , for all the x in e to obtain a new expression, $e' ::= \text{subst}(e, f)$. Now suppose we want to evaluate e' when x has the value n . One way to do it would simply be to evaluate e' recursively, ignoring the way that e' was obtained from e and f . But another, usually more efficient, approach would be to find the value of f when x has the value n , and then evaluate e when x is given the value obtained from f . In Lisp programming terminology, the first approach corresponds to evaluation using a “substitution model,” and the second approach corresponds to evaluation using an “environment model.” We will prove that both models yield the same answer. More precisely, what we want to prove is

Theorem 4.3. For all expressions $e, f \in F$ and $n \in \mathbb{Z}$,

$$\text{eval}(\text{subst}(e, f), n) = \text{eval}(e, \text{eval}(f, n)). \quad (8)$$

Proof. The proof is by structural induction on e .

Base cases ($e = \mathbf{0}$, $e = \mathbf{1}$). Then the lefthand side of equation (8) equals e by the base cases of the definition of `subst`, and the righthand side equals e by the base cases of the definition of `eval`.

Base case ($e = \mathbf{x}$). Then the lefthand side of equation (8) equals $\text{eval}(f, n)$ by the base case for \mathbf{x} of the definition of `subst`, and the righthand side equals $\text{eval}(f, n)$ by the base case for \mathbf{x} of the definition of `eval`.

Inductive step ($\mathbf{[e+e']}$). Assume that for all $f \in F$ and $n \in \mathbb{N}$,

$$\text{eval}(\text{subst}(e, f), n) = \text{eval}(e, \text{eval}(f, n)) \quad (9)$$

$$\text{eval}(\text{subst}(e', f), n) = \text{eval}(e', \text{eval}(f, n)), \quad (10)$$

to prove that for all $f \in F$ and $n \in \mathbb{N}$,

$$\text{eval}(\text{subst}(\mathbf{[e+e']}, f), n) = \text{eval}(\mathbf{[e+e']}, \text{eval}(f, n)) \quad (11)$$

But the lefthand side of (11) equals

$$\text{eval}(\mathbf{[subst}(e, f)\mathbf{+subst}(e', f)\mathbf{]}, n)$$

by definition of `subst` for a sum expression, which equals

$$\text{eval}(\text{subst}(e, f), n) + \text{eval}(\text{subst}(e', f), n)$$

by definition of `eval` for a sum expression. By induction hypothesis, this equals

$$\text{eval}(e, \text{eval}(f, n)) + \text{eval}(e', \text{eval}(f, n)),$$

which equals the righthand side of (11) by definition of `eval` for a sum expression. This proves (11) in this case.

Inductive step ($\mathbf{[e*e']}$). Similar.

Inductive step ($\mathbf{[-e']}$). Similar. □

4.4 Recursive Functions on Natural Numbers

The recursive definitions of functions on recursively defined data types also applies to recursively defined functions on the natural numbers. One can think of the natural numbers, \mathbb{N} , as recursively defined by

1. $0 \in \mathbb{N}$,
2. if $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

Now ordinary induction is exactly structural induction based on this recursive definition. Treating the natural numbers as a recursively defined data type also justifies the use of familiar recursive definitions of functions on the natural numbers.

The Factorial function: A very useful function for counting and probability.

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

The Fibonacci numbers: These are interesting numbers that arise, e.g., in biology, where they model some types of growth processes (plants, cells, rabbit populations, etc.). The Fibonacci numbers are written as F_i , $i = 0, 1, 2, \dots$. They are defined recursively by:

$$\begin{aligned} F_0 &::= 0, \\ F_1 &::= 1, \\ F_i &::= F_{i-1} + F_{i-2} \text{ for } i \geq 2. \end{aligned}$$

Note there are two base cases, since each combination relies on previous two 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 $0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$

Σ notation (the traditional style): We've been using this notation informally already.

- $\Sigma_{i=1}^0 f(i) ::= 0$,
- $\Sigma_{i=1}^{n+1} f(i) ::= \Sigma_{i=1}^n f(i) + f(n + 1)$, for $n \geq 0$.

Simultaneous recursive definitions: You can define several things at the same time, in terms of each other. For example, we may define two functions f and g from \mathbb{N} to \mathbb{N} , recursively, by:

- $f(0) ::= 1$,
- $g(0) ::= 1$,
- $f(n + 1) ::= f(n) + g(n)$, for $n \geq 0$,
- $g(n + 1) ::= f(n) \times g(n)$, for $n \geq 0$.

We can use the recursive definitions of functions in proving their properties. As an illustration, we'll prove a cute identity involving Fibonacci numbers. Fibonacci numbers provide lots of fun for mathematicians because they satisfy many such identities.

Proposition 4.4. $\forall n \geq 0 (\sum_{i=0}^n F_i^2 = F_n F_{n+1})$.

Example: $n = 4$:

$$0^2 + 1^2 + 1^2 + 2^2 + 3^2 = 15 = 3 \cdot 5.$$

Let's try a proof by (standard, not strong) induction. The theorem statement suggests trying it with $P(n)$ defined as:

$$\sum_{i=0}^n F_i^2 = F_n F_{n+1}.$$

Base case ($n = 0$). $\sum_{i=0}^0 F_i^2 ::= (F_0)^2 = 0 = F_0 F_1$ because $F_0 ::= 0$.

Inductive step ($n \geq 0$). Now we stare at the gap between $P(n)$ and $P(n+1)$. $P(n+1)$ is given by a summation that's obtained from that for $P(n)$ by adding one term; this suggests that, once again, we subtract. The difference is just the term F_{n+1}^2 . Now, we are assuming that the original $P(n)$ summation totals $F_n F_{n+1}$ and want to show that the new $P(n+1)$ summation totals $F_{n+1} F_{n+2}$. So we would *like* the difference to be

$$F_{n+1} F_{n+2} - F_n F_{n+1}.$$

So, the actual difference is F_{n+1}^2 and the difference we want is $F_{n+1} F_{n+2} - F_n F_{n+1}$. Are these the same? We want to check that:

$$F_{n+1}^2 = F_{n+1} F_{n+2} - F_n F_{n+1}.$$

But this is true, because it is really the Fibonacci definition in disguise: to see this, divide by F_{n+1} .

5 Ill-formed Definitions

We must take care that functions defined recursively are well-defined. Below are some function specifications that look like definitions but aren't.

$f(n) = 2 + f(n-1)$. This "definition" has no base case. If some function, f , satisfied this equation, so would a function obtained by adding a constant, k , to the value of f . So this "definition" does not uniquely define f .

$f(n) = 0$ if n is divisible by 2, $f(n) = 1$ if n is divisible by 3, and $f(n) = 2$ otherwise. This "definition" is inconsistent: it requires $f(6) = 0$ and $f(6) = 1$, so it doesn't define anything.

$f(0) = 0$, otherwise $f(n) = f(n+1) + 1$. From this "definition," it follows that $f(1) > f(2) > f(3) > \dots$, so $f(1)$ cannot equal any integer. No total function on the natural numbers can satisfy this definition. However, it does uniquely determine a *partial* function on the natural numbers, namely, the function that is 0 at 0 and undefined everywhere else.

$f(0) = 0$, otherwise $f(n) = f(n+1)$. Lots of total functions satisfy the equations in the "definition." Namely, any function that is 0 at 0 and constant everywhere else.

A mystery. Mathematicians have been wondering about this one for a while:

- $f(1) = 1$.
- If n is even, then $f(n) = f(n/2)$.
- If n is odd, then $f(n) = f(3n + 1)$.

This “definition” of f in some cases defines $f(n)$ in terms of f applied to arguments larger than n , and so cannot be justified by induction on \mathbb{N} . It has been proven that if f is a function that satisfies the above equations, then $f(n) = 1$ for all n up to at least a billion, and it’s a good guess that the only f that works is the constant function equal to 1. But nobody knows if there is more than one function satisfying these equations.

6 Induction in Computer Science

We’ve spent a lot of time studying induction. This is because induction comes up all the time in analyzing computation. Why? Well, ordinary induction on natural numbers is a “one step at a time” proof method. Computations also evolve “one step at a time.”

Structural induction on recursive definitions lets us go beyond simple natural number counting. We can explain how to define recursive functions on recursive data types. And we can prove properties of recursively defined data and functions by structural induction on the definition of the data type. We even noted that Structural Induction is technically more powerful than ordinary induction. It is a technique which every Computer Scientist should firmly grasp.