CDM
Induction

Klaus Sutner

Carnegie Mellon Universality

10-induction 2017/12/15 23:20

1. Induction and Recursion

- Natural Numbers
- Induction Proofs
- Well-Orders
- Lists
- Trees

The key idea in induction is to consider an

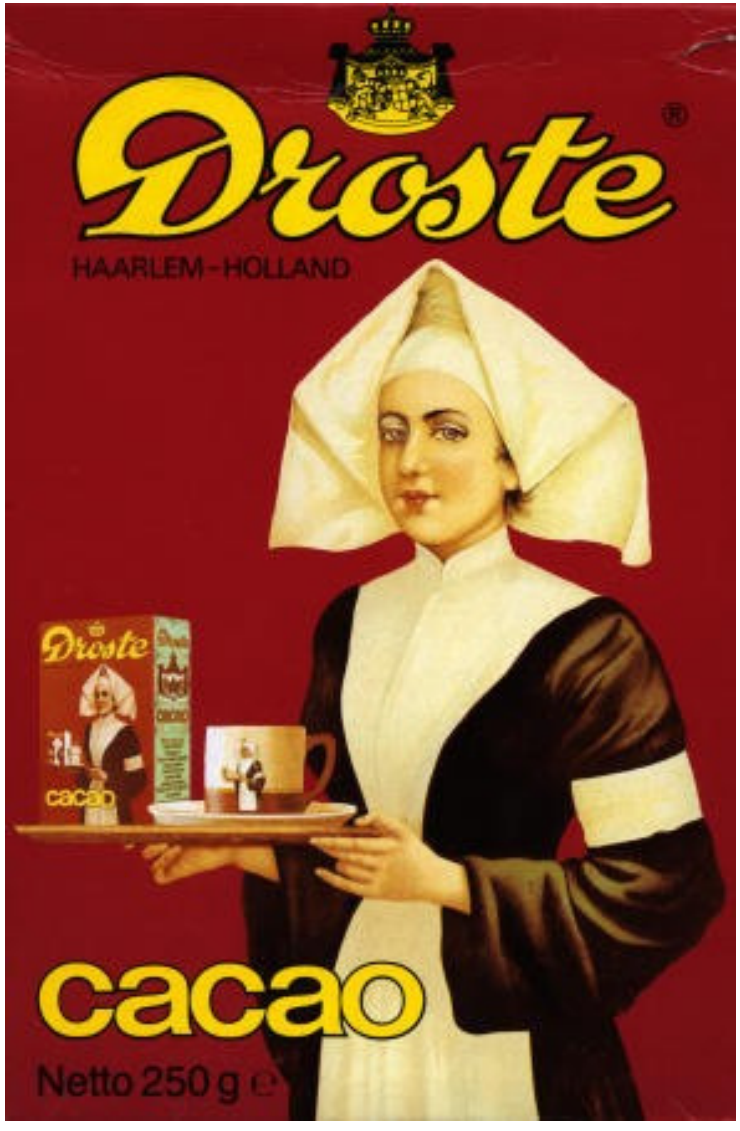
> Object that is similar to parts of itself.

So this object is somehow composed of smaller, yet similar objects. Since the smaller components are similar to the large object, they in turn can be decomposed into yet smaller ones, and so on.

Clearly, there are two scenarios:

- the descent goes on forever,
- after finitely many steps, the descent reaches indecomposable atoms, and stops.

For computational purposes, the finite descent case is much more important, but in geometry the infinite version is easier to understand.


Droste
HAARLEM - HOLLAND
Droste
cacao
cacao
Netto 250 g

In order to describe a terminating decomposition process carefully, we need

- a collection of indecomposable atoms

and

- one or several destructors.

We can apply a destructor to any composite (non-atomic) object and obtain "smaller" ones, until ultimately we get atoms.

For example, the computation of $50!$ contains a similar sub-computation for $49!$, which contains a sub-computation for $48!$ and so on, down to $0!$ where the recursion stops.

We can also turn decomposition upside down: with start with atoms and then use constructors to build composite objects. So this time we have

- a collection of indecomposable atoms

and

- one or several constructors.

We are only interested in the case where a constructor applies to finitely many objects. Hence termination is not really a problem here, every composite object can be reached after finitely many steps.

For example, we understand the computation for $0!$ and can use it to build up the computation of $1!$, $2!$, $3!$ all the way up to $50!$.

In mathematics one typically speaks about an inductively defined structure (or set). In computer science we have recursive datatypes which are really the same thing.

In general, induction refers to the bottom-up approach whereas recursion refers to the top-down approach. Implementation detail may differ, but the underlying idea is the same.

As we will see in a moment, induction can be used in particular to construct proofs and is absolutely critical in many arguments in number theory, combinatorics and data structures.

But somehow, no one ever seems to talk about a "proof by recursion."

Suppose we have some class C of inductively defined structures.

Definitions We can use either induction or recursion to define various new functions and relations on C.
This allows us to perform all kinds of computations.

Proofs Then we use induction to prove the critical properties of these functions and relations.
This allows us to reason about these computations.

Often these proofs are very mechanical, they almost write themselves. Except that it's not always clear which direction to move in.

It is customary to think of the constructors as mappings and to write

$$b = S(a_1, a_2, \ldots, a_k)$$

for the object b obtained by applying the $k$ary constructor S to objects a_1, $\ldots$, a_k.

That's fine, but be clear that this is just syntactic sugar, we might as well have written

$$b = S a_1 a_2 \ldots a_k$$

or perhaps

$$b = (\!| a_k a_{k-1} \ldots a_1 ; S |\!)$$

In the opposite direction, given a compound object

$$b = S(a_1, \ldots, a_k)$$

we can use destructors to obtain the components a_i.

Moreover, in all cases of interest to us, this decomposition process is unique: b cannot be produced in any other way than as indicated. In other words, the constructors always produce new objects.

This is not an essential feature, but often a very useful one.

- Induction and Recursion

2. Natural Numbers

- Induction Proofs
- Well-Orders
- Lists
- Trees

We can think of the natural numbers as being constructed from

- the atom 0

and

- the constructor S, the successor operation.

Thus we obtain 0, $S(0)$, $S(S(0))$, $S(S(S(0)))$, ...

Of course, there are much better notation systems, but the goal here is to define the actual objects, not to write them down in an algorithmically attractive way.

If you are a friend of set theory you will want to think of S as some kind of function.

For the construction to work, all we need to know about S is:

$$S(x) = S(y) \Rightarrow x = y$$

$$S(x) \neq 0$$

In other words, S must be injective and 0 cannot lie in its range. By injectivity the decomposition is unique.

Alternatively, we could think of S as an uninterpreted function symbol so that the "natural numbers" are just the terms $0, S(0), S(S(0)), \ldots$ This approach collides a bit with our intuition but there is nothing fundamentally wrong with it.

We can think of 0 as $\emptyset$ and $S(x) = x \cup \{x\}$.

Let's say that a set N is *closed* if $0 \in N$ and $x \in N$ implies $S(x) \in N$.

Definition (Natural Numbers, inductive style)

The set of natural numbers $\mathbb{N}$ is the least set that is closed.

"Least" here is meant in the sense of set inclusion.

If you prefer, you can write this impredicatively as

$$\mathbb{N} = \bigcap \{\, N \mid N \text{ is closed} \,\}$$

Since every element x of $\mathbb{N}$ is either 0 or (uniquely) of the form $x = S(y)$, we can use this inductive structure of $\mathbb{N}$ to define a simple predicate as follows.

$$Z(0) = \mathsf{tt}$$
$$Z(S(x)) = \mathsf{ff}$$

This is a zero-test: it returns true iff the input is 0.

Slightly more interesting is the predecessor function:

$$p(0) = 0$$
$$p(S(x)) = x$$

And here is addition:

$$\mathsf{add}(x, 0) = x$$
$$\mathsf{add}(x, S(y)) = S(\mathsf{add}(x, y))$$

And multiplication:

$$\begin{aligned} \mathsf{mult}(x, 0) &= 0 \\ \mathsf{mult}(x, S(y)) &= \mathsf{add}(x, \mathsf{mult}(x, y))) \end{aligned}$$

And so on. We can build all standard arithmetic functions this way.

This was first noticed by Grassmann and Dedekind in the middle of the 19th century.

G. Peano gave an axiomatization of the natural in the late 1800s. We'll write number instead of natural number.

- 0 is a number.
- The successor of a number is a number.
- 0 is not a successor.
- Two different numbers have different successors.
- If a property obtains at 0 and is inherited by the successor of every number with this property, then this property holds of all numbers.

The last axiom is the basis for proofs by induction.

Peano's system is based on 3 basic concepts

- number
- zero
- successor

and describes the relationship between these concepts.

Unfortunately, there are many unintentional ways one can interpret these concepts: zero is $1 \in \mathbb{Q}$, $S(x) = x/2$ and number $= \{\, 1/2^i \in \mathbb{Q} \mid i \geq 0 \,\}$.

This problem cannot be fixed in first-order logic.

In his 1919 pamphlet "Das Kontinuum," Hermann Weyl emphasizes the importance of this approach:

> *. . . dass die Vorstellung der Iteration, der natürlichen Zahlenreihe, ein letztes Fundament des mathematischen Denkens ist–trotz der Dedekindschen Kettentheorie.*
>
> *. . . that the concept of iteration, of the natural number-sequence, is the ultimate foundation of mathematical thought–in spite of Dedekind's theory of chains.*

Weyl's proposal got drowned out by set theory and type theory, but he makes a very good point.

- Induction and Recursion
- Natural Numbers

3. Induction Proofs

- Well-Orders
- Lists
- Trees

We can now use induction to construct proofs for these inductively defined functions. To wit:

In order to establish some assertion $\varphi(x)$ for all $x \in \mathbb{N}$ we need to

- Establish $\varphi(0)$

and

- show that $\varphi(x)$ implies $\varphi(S(x))$.

It is crucial that x in the second part is a free variable, it stands for a generic natural number and can not be assumed to have any special properties.

To prove some assertion about all natural numbers, say, $\forall\, x\, \varphi(x)$, it suffices to show

$$\varphi(0) \wedge \forall\, x\, (\varphi(x) \Rightarrow \varphi(S(x)))$$

Definition

This is called the Induction Principle on $\mathbb{N}$ (IND).

(IND) is crucially important in any theory of arithmetic, it is enshrined in the induction axiom schema of Peano arithmetic.

As an example, let us prove that the function add from above is commutative. We will need a few auxiliary results.

Claim

$\mathsf{add}(0, y) = y$

Base case: $\mathsf{add}(0, 0) = 0$

Inductive step: $\mathsf{add}(0, S(y)) = S(\mathsf{add}(0, y)) =_{\mathrm{IH}} S(y)$.

Claim

$\mathsf{add}(x, S(y)) = \mathsf{add}(S(x), y)$

Base case: $y = 0$

$$\mathsf{add}(x, S(0)) = S(\mathsf{add}(x, 0)) = S(x) = \mathsf{add}(S(x), 0)$$

Inductive step:

$$\mathsf{add}(x, S(S(y))) = S(\mathsf{add}(x, S(y))) =_{\mathrm{IH}} S(\mathsf{add}(S(x), y)) = \mathsf{add}(S(x), S(y)).$$

Claim

$\mathsf{add}(x, y) = \mathsf{add}(y, x)$

Base case: $y = 0$ is Claim 1.

Inductive step:

$$\mathsf{add}(x, S(y)) = S(\mathsf{add}(x, y)) =_{\mathrm{IH}} S(\mathsf{add}(y, x)) = \mathsf{add}(y, S(x)) =_{\mathrm{C2}} \mathsf{add}(S(y), x).$$

This is admittedly a bit tedious, but note that the argument is crystal clear, there are no hidden assumptions or appeals to intuition.

It is time to lighten notation a bit. We can think of the standard numerals as abbreviations for nested S-terms:

$$1 = S(0), 2 = S(1) = S(S(0)), 3 = S(2), \ldots$$

and so on. Then

$$\mathsf{add}(3,2) = S(\mathsf{add}(3,1)) = S(S(\mathsf{add}(3,0))) = S(S(3)) = S(4) = 5$$

and things work as advertised.

Of course, we would usually write $x + y$ instead of $\mathsf{add}(x, y)$, but that is just syntactic sugar (and more work for the parser, though it's easier on human eyes).

There is a secret conspiracy among mathematicians that requires the following, unbearably boring, exercise to be part of any discussion of induction.

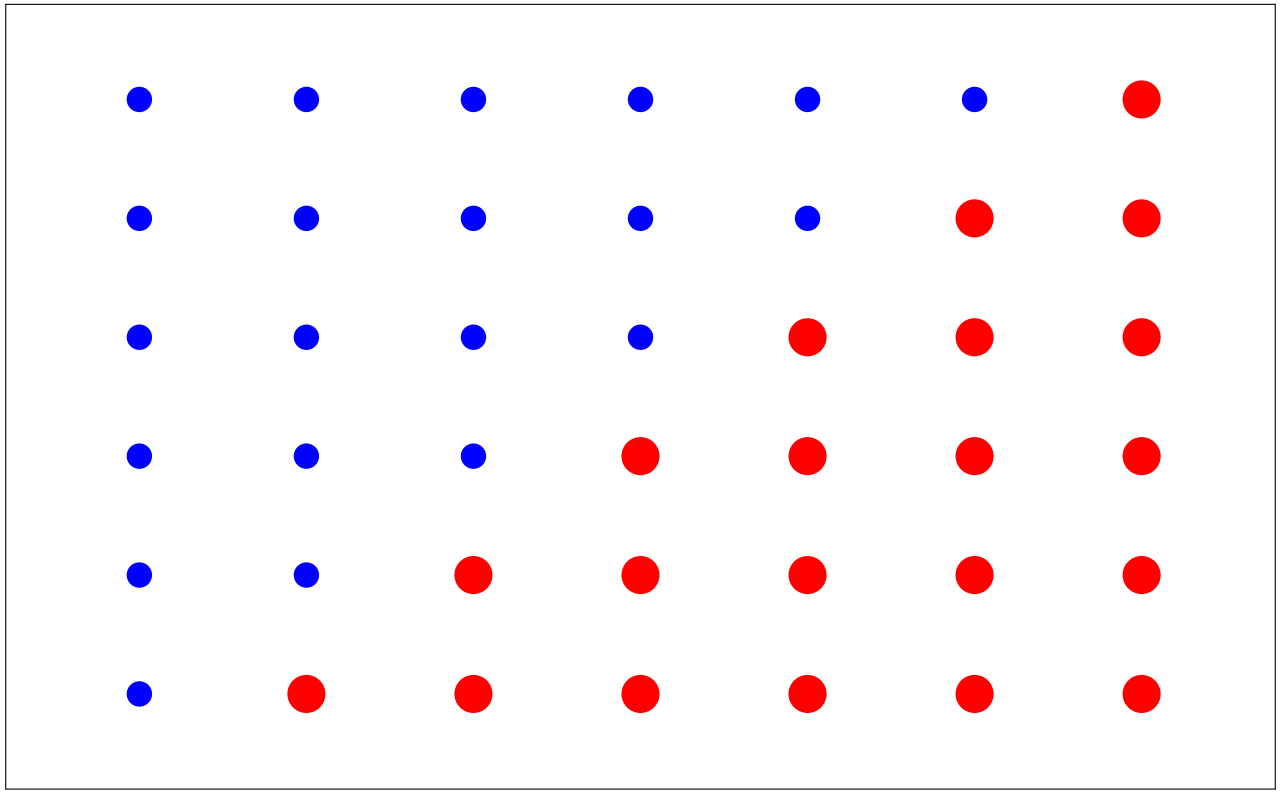
Show that

$$\sum_{i=1}^{n} i = n(n+1)/2$$

for all natural numbers n.

Note that one can establish this result (apparently) without induction: the average of the n numbers is $(n+1)/2$, so the sum must be $n(n+1)/2$.

Of course, inquisitive minds would want to know what we mean by average, and how exactly the result follows. Averages may sound patently harmless, but how about the following?



There is a proof in this picture, somewhere, but it is not exactly clear how we would formalize this argument. Could a computer perform this type of proof?

We are trying to show

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

- Base case: $\sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$ clearly true.
- Induction step:
  Assume $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

$$\begin{aligned}\sum_{i=1}^{n+1} i &= \sum_{i=1}^{n} i + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)(n+2)}{2}\end{aligned}$$

Done.

A very similar argument shows that

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

The assertion here is $\varphi(n) \equiv \sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.

It is straightforward to check that $\varphi(0)$ holds.

The only difference between this and the previous argument is that the algebraic operations required to verify that $\varphi(n)$ implies $\varphi(n+1)$ are slightly more complicated.

In fact, the whole procedure is so routine that many summation identities can be "proved automatically" using a computer algebra system to do the dirty work. The general idea is to show

$$\sum_{i=0}^{n} f(i) = g(n)$$

Here $g(n)$ is supposed to be a reasonably simple function, a so-called closed form. Polynomials are OK, exponentials, factorials, and a few more exotic functions.

Once we have $g(n)$, the induction proof boils down to show that

$$f(0) = g(0)$$
$$f(n+1) = g(n+1) - g(n)$$

The induction is now replaced by establishing two identities with free variables.

Equivalently, we have to show that certain terms are identically 0:

$$f(0) - g(0) = 0$$
$$f(n+1) - g(n+1) + g(n) = 0$$

For example, if the functions in questions are polynomials this is quite straightforward and can be handled by a standard simplification routine in a modern computer algebra system.

Of course, this approach requires prior knowledge of the closed form solution of the summation. Some CAS have tools to determine these closed forms in some cases.

Example

Mathematica can simplify the following summation.

$$\sum_{i=1}^{n} i^3\, 2^i = 2^{n+1}(n^3 - 3n^2 + 9n - 13) + 26$$

The real reason this approach works is that summation itself is a recursive procedure:

$$\sum_{i=0}^{0} f(i) = f(0)$$

$$\sum_{i=0}^{n+1} f(i) = \sum_{i=0}^{n} f(i) + f(n+1)$$

Or, to make it look more like a program:

$$\mathsf{sum}(0; f) = f(0)$$

$$\mathsf{sum}(n+1; f) = \mathsf{sum}(n; f) + f(n+1)$$

So we are using inductive reasoning to describe an inductively defined function. The process is so mechanical (though by no means trivial), that the CAS can perform it automatically.

But note that not all induction proofs are quite so straightforward. Here is a little lemma in plane geometry.

> Place n distinct points into the interior of an equilateral triangle (some may be collinear). Then draw additional lines between these points and the vertices of the triangle until all the regions are themselves triangles.
>
> **Claim:** There are exactly $2n+1$ regions in the fully triangulated figure.

Since we have no information about the placement of the points this looks like an induction problem.

Claim

There are exactly $2n + 1$ regions in the fully triangulated figure.

Proof.

Induction on n, the number of points.

Base case: For $n = 0$ or 1 the claim clearly holds.

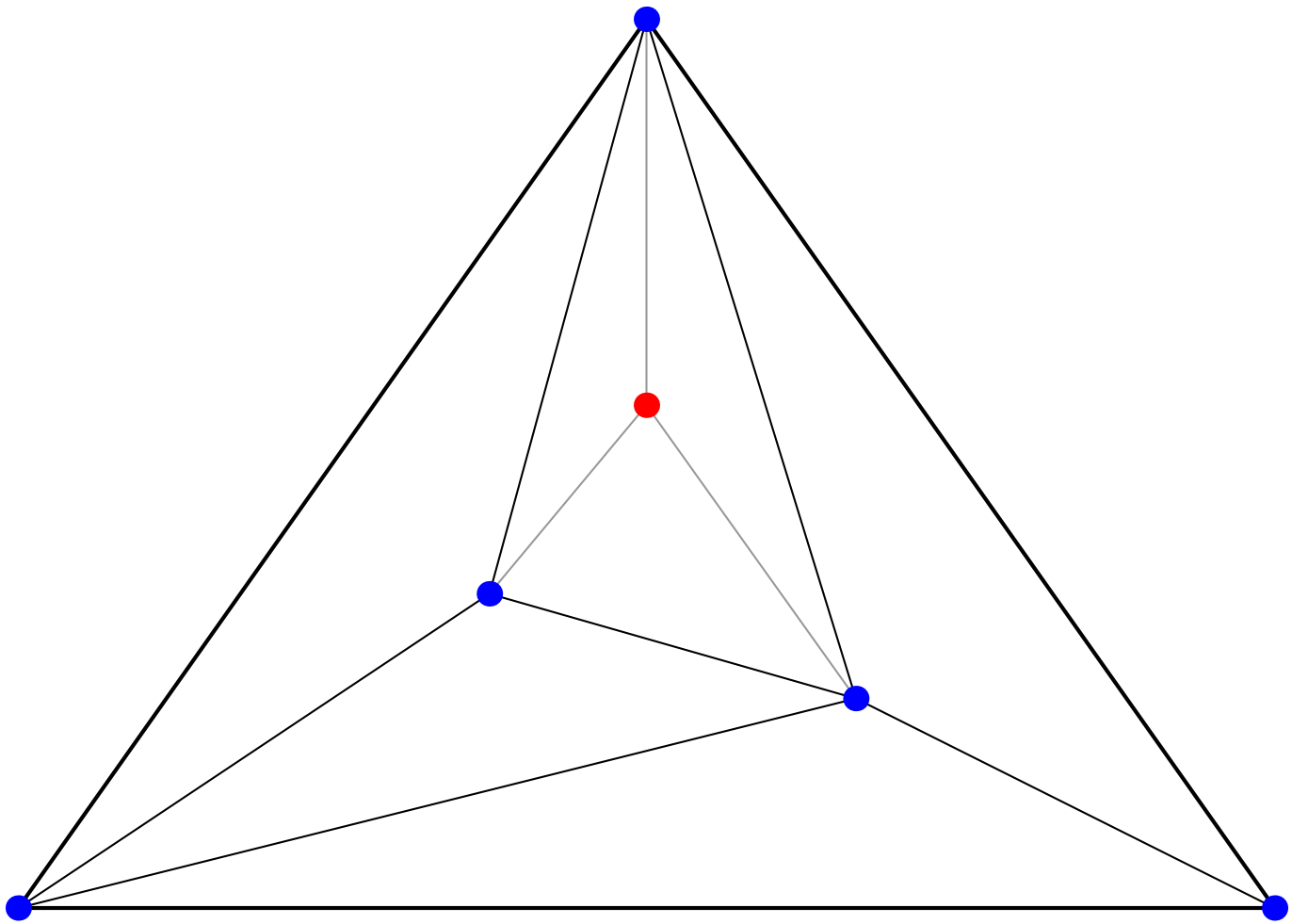
Induction step:

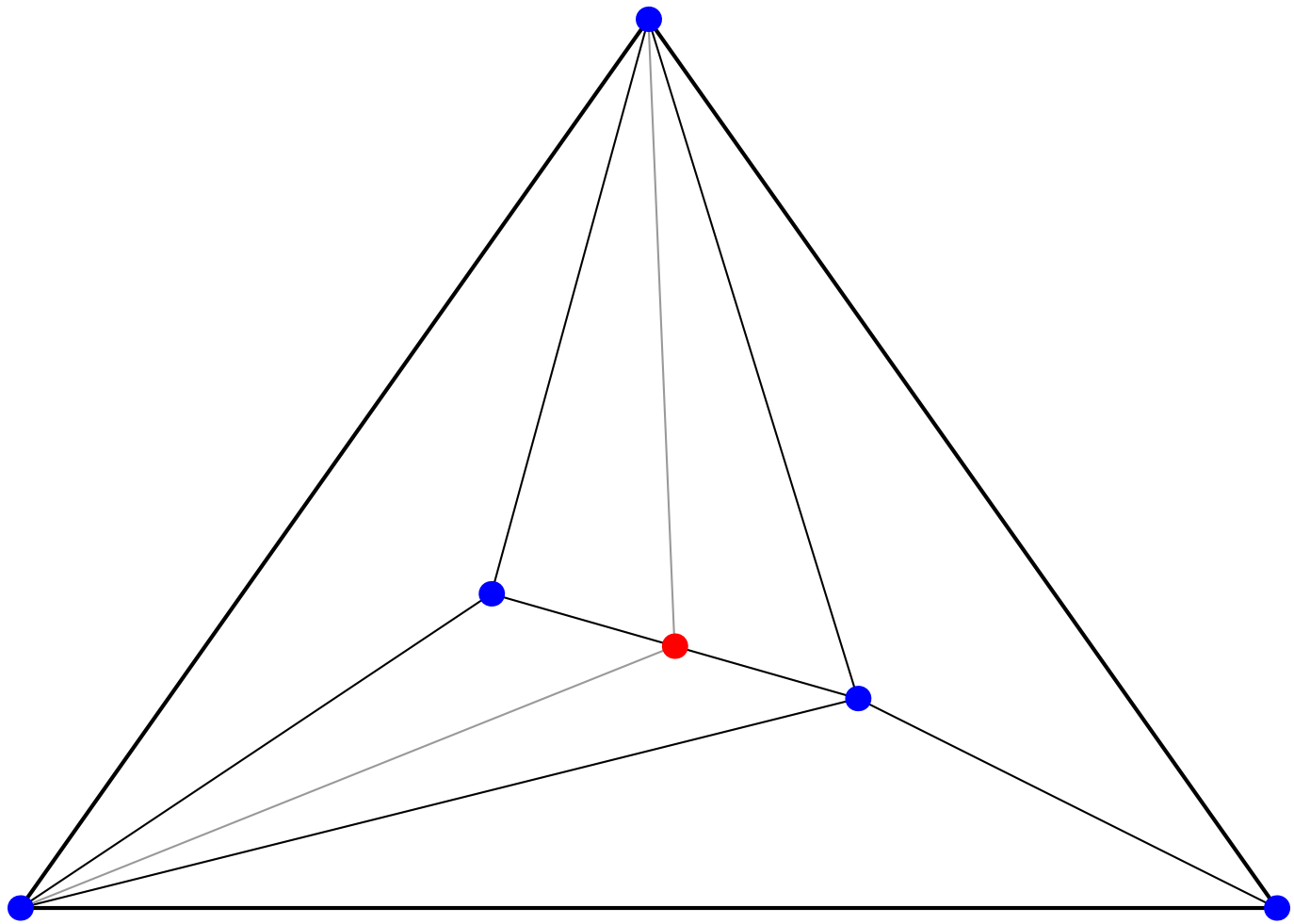
Add one more point p to the figure.

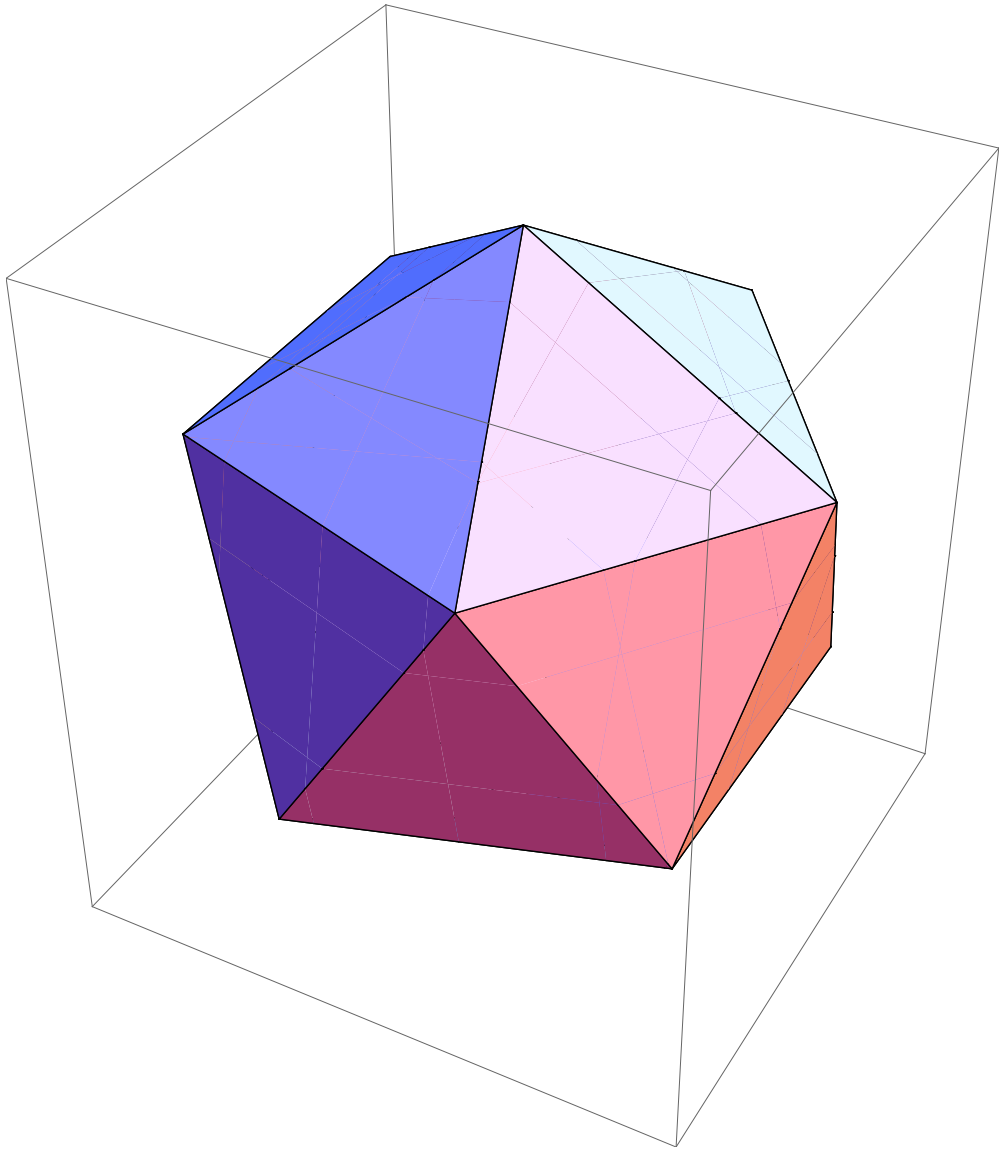
p is either in the interior of a triangle or on a line.

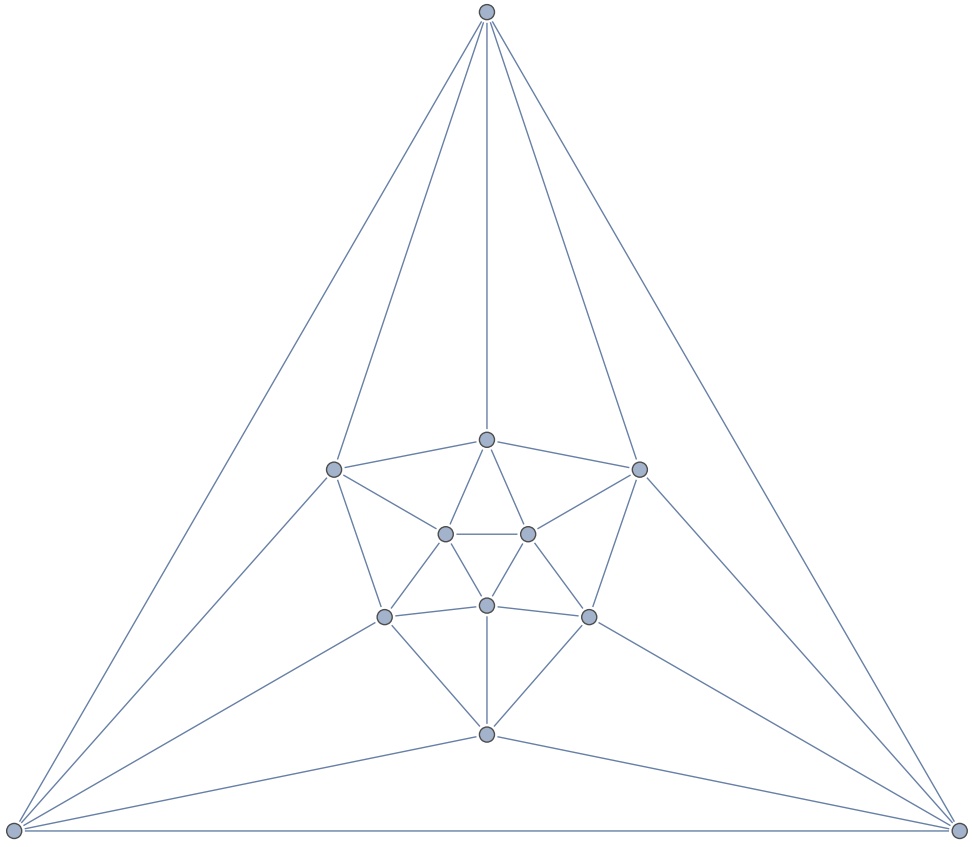
In either case, completing the triangulation produces two more regions.

□









Alas, there is a problem. While the claim holds for the triangulation above, it cannot be generated by the steps in the induction argument.

Exercise

Explain exactly what went wrong in the induction proof.

Exercise

Give a valid proof for the triangulation claim.

- Induction and Recursion
- Natural Numbers
- Induction Proofs

4. Well-Orders

- Lists
- Trees

Induction on $\mathbb{N}$ is so useful that it is worth to try to find alternate forms (which may be easier to use in some circumstances).

The first variant is an assertion about the order $\langle \mathbb{N}, < \rangle$.

Definition (LEP)

Every non-empty subset of $\mathbb{N}$ has a least element.

Proposition

(LEP) and (IND) are equivalent.

You might argue that both (IND) and (LEP) are clearly true, so they are trivially equivalent–but that is emphatically not what we mean.

Instead, we claim that given a weak theory of the natural numbers a background, adding (IND) will allow us to prove (LEP), and conversely.

Proof. Assume (LEP).

Suppose we have $\varphi(0)$ and $\forall\, x\, (\varphi(x) \to \varphi(x+1))$ for some φ.

If (IND) were to fail for φ there would have to be some $a \in \mathbb{N}$ so that $\neg\varphi(a)$.

But then

$$A = \{\, x \in \mathbb{N} \mid \neg\varphi(x) \,\}$$

is non-empty, hence has a least element a_0 by (LEP).

Clearly, $a_0 \neq 0$ by the base case. But then $\varphi(a_0 - 1)$ by definition of A. Hence $\varphi(a_0)$ holds by the induction assumption above, contradiction.

Now assume (IND).

Suppose $A \subseteq \mathbb{N}$ has no least element and consider the assertion

$$\varphi(n) = \forall\, x \leq n\, (x \notin A)$$

Then $\varphi(0)$ holds: otherwise $0 \in A$ would be the least element.

Likewise, $\varphi(n)$ implies $\varphi(n+1)$: otherwise $n+1$ would be the least element of A.

By then by (IND) $\forall\, n\, \varphi(n)$ and A is empty. Done.

□

While (LEP) and (IND) are equivalent the Least Element Principle has the advantage that it generalizes nicely to more complicated situations: there is no mention of the successor function.

Here is another way one can get rid of the successor function.

Call a formula $\varphi(x)$ inductive if

$$\forall\, x\, (\forall\, z < x\, \varphi(z) \to \varphi(x))$$

Definition (SIND)

If $\varphi(x)$ is inductive, then $\forall\, x\ \varphi(x)$.

The part $\forall\, z < x\, \varphi(z)$ functions like the Induction Hypothesis.

At first glance, it looks like we have lost the base case.

Not to worry, it's still there, just let $x = 0$: $\forall\, z < 0\, \varphi(z)$ is vacuously true: there are no $z < 0$ in $\mathbb{N}$. Hence, we have to prove $\varphi(0)$ from scratch, just as before.

The advantage of (SIND) over (IND) is that we have a whole collection of induction hypotheses, not just one. On occasion, that is necessary for the proof to go through.

Example (Prime Divisors)

Every integer $n \geq 2$ has a prime divisor.

We need to know that all m, $2 \leq m \leq n$, have a prime divisor in order to get a prime divisor for $n+1$.

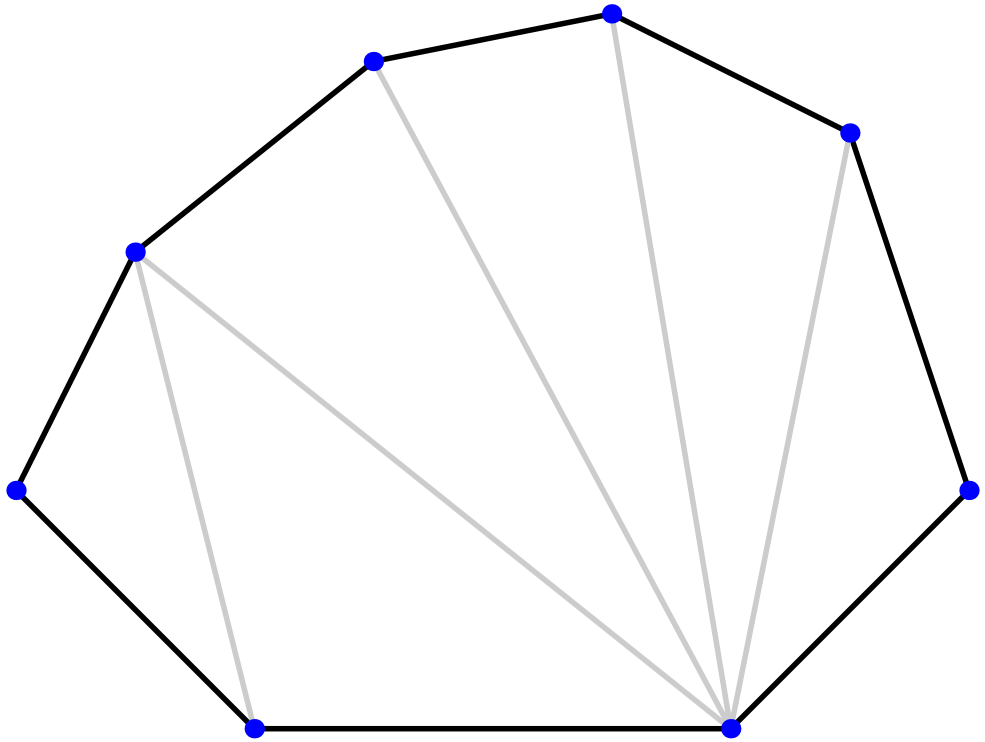
Similar problems are often encountered in parsing: we need to have dealt with all the subexpressions (of unpredictable size) before tackling the main expression.

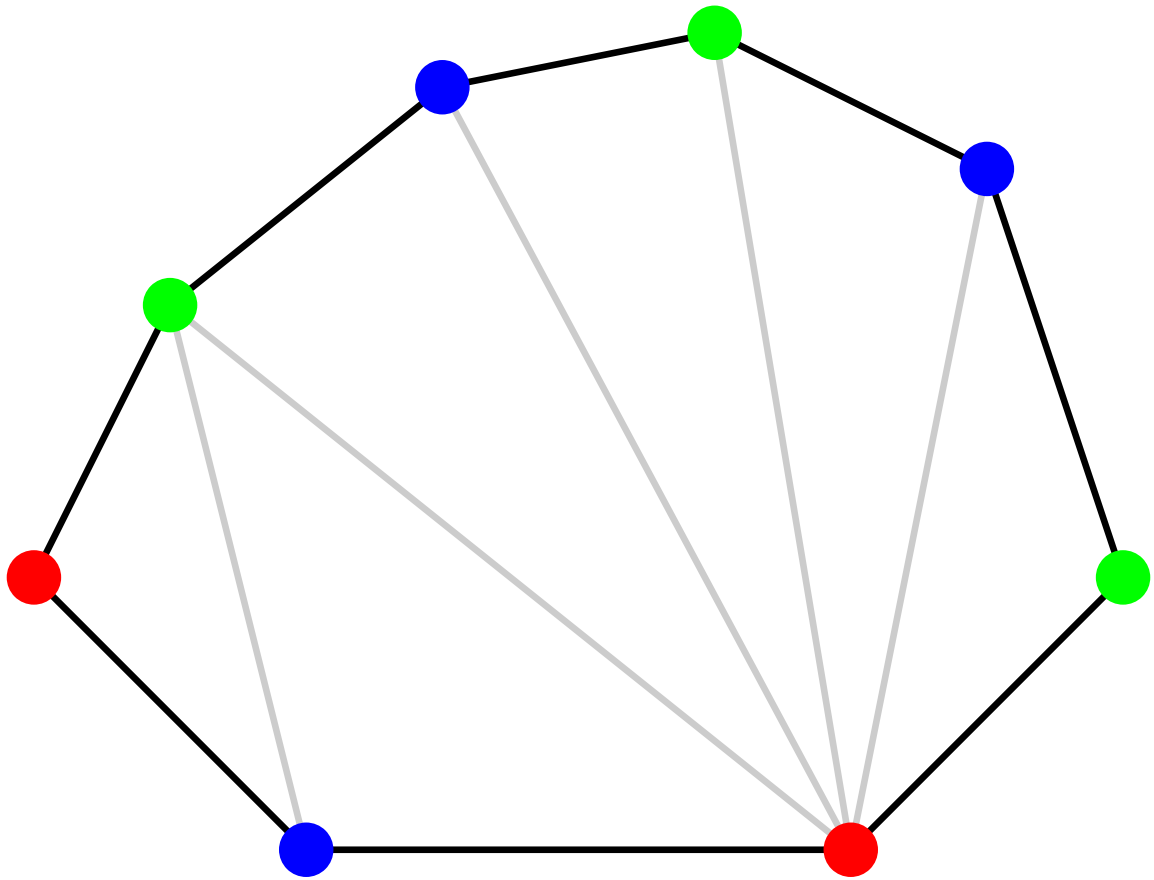
In programming, (SIND) corresponds to recursive functions where $f(n)$ is defined in terms of arbitrary $f(i)$, $i < n$; not just $f(n-1)$. A typical and important case is when the recursion looks like

$$f(n) = \ldots f(\lfloor n/2 \rfloor) \ldots f(\lceil n/2 \rceil) \ldots$$

Note that this type of recursion terminates in $O(\log n)$ steps.

Claim: Let A be a triangulated convex polygon on $n \geq 3$ nodes. Then we can color the nodes with 3 colors so that no two adjacent nodes have the same color.





A solution.

Let $P(n)$ be the colorability claim for all triangulated n-gons. We need to show that P is inductive.

The claim clearly holds for $n = 3$.

So assume $n > 3$.

Consider a triangle with nodes p, q and r.

Say, the line segment pq lies on the boundary of A.

Case 1: qr is also on the boundary.

Remove point q.

The remaining figure B is a convex polygon on $n-1$ points and has a coloring by IH. Clearly, the coloring can be extended to q.

Case 2: Only pq is on the boundary.

Remove pq to obtain two convex polygons joined at r.

Call them B and C where p is in B and q in C.

By IH we can color both B and C.

Moreover, we can make sure that r has the same color in both pieces: just rename the colors.

If p in B and q in C have different colors we are done.

Otherwise change the color of q in C (but not of p). Done.

□

Exercise

Fill in all the gaps in the last proof.

Exercise

There is more geometric way to find a coloring. Explain informally how this method works.

Exercise

Then give a strict proof of your method. Most likely your proof will involve induction on the number of nodes.

So is (SIND) really a new induction principle?

Proposition

(LEP), (IND) and (SIND) are all equivalent.

Of course, both (IND) and (SIND) are true, we are not making the utterly useless claim that true is equivalent to true.

The real claim is that, using only very weak axioms, the assumption of (IND) allows us to give a short proof of (SIND).

Conversely, the assumption of (SIND) allows us to give a short proof of (IND).

The two principles have the same proof power.

Assume (LEP).

Suppose φ is inductive: $\forall\, z < x\, \varphi(z) \Rightarrow \varphi(x)$ for all x.

If (SIND) failed, we could invoke the (LEP) to define the set of counterexamples

$$A = \{\, x \in \mathbb{N} \mid \neg\varphi(x) \,\}$$

where A is non-empty, hence has a least element a_0. But then $\forall\, z < a_0\, \varphi(z)$ by definition, so $\varphi(a_0)$, contradiction.

Hence (LEP) implies (SIND).

Assume (SIND).

Suppose we have $\varphi(0)$ and $\forall\, x\, (\varphi(x) \Rightarrow \varphi(x+1))$.

Let $\Phi(x) \equiv \forall\, z \preccurlyeq x\, \varphi(z)$.

Then Φ is inductive and it follows that $\forall\, x\, \varphi(x)$.

Hence (SIND) implies (IND).

Now assume (IND).

Suppose $A \subseteq \mathbb{N}$ has no least element and consider the assertion

$$\varphi(n) \equiv \forall\, x \leq n\, (x \notin A)$$

Then $\varphi(0)$ holds: otherwise $0 \in A$ would be the least element.

Likewise, $\varphi(n)$ implies $\varphi(n+1)$: otherwise $n+1$ would be the least element of A.

By then by (IND) $\forall\, n\, \varphi(n)$ and A is empty. So (LEP) holds and we are done.

The argument for (IND) is entirely similar.

□

Yet another variant of induction that is often useful in computer science is the following Modular Induction Principle (MIP). In order to show $\forall\, x\, \varphi(x)$ it suffices to establish

$$\varphi(0) \wedge \forall\, x\, (\varphi(x) \to \varphi(2x) \wedge \varphi(2x+1))$$

Of course, this principle generalizes to other moduli, but the binary case is probably the most important since we are often dealing with binary expansions of natural numbers.

As an example, consider the Thue sequence T_n, a binary sequence defined by $T_0 = 0$ and

$$T_n = \begin{cases} T_{n/2} & \text{if } n \text{ even,} \\ T_{n-1} & \text{otherwise.} \end{cases}$$

Then (MIP) shows that

$$T_n = \mathsf{digsum}(n) \bmod 2$$

Recall that we interested in chains of decomposition steps that are guaranteed to terminate: after finitely many steps an atom must appear. Is there a mathematical concept that can make this idea precise? And, perhaps, generalize beyond just the natural numbers? Here is one attempt at generalization, using (LEP) as the starting point.

Definition

Suppose $\prec$ is a strict partial order on some set A. $\prec$ is well-founded if

$$\forall\, \emptyset \neq X \subseteq A \ (X \text{ has a } \prec\text{-least element}).$$

If the order is total then we speak of a well-order.

By $\prec$-least element we mean some $a \in X$ such that $\forall\, x \in X \ (x \not\prec a)$.

In other words, (LEP) holds for $\langle A, \prec \rangle$ rather than $\langle \mathbb{N}, < \rangle$.

In dealing with recursive datatypes we often don't have a total order, just a partial one – which does not affect the applicability of induction.

Lemma

$\prec$ is well-founded if, and only if, there is no infinite descending chain

$$x_0 \succ x_1 \succ x_2 \succ \ldots \succ x_n \succ \ldots$$

Proof.

Suppose we have a strictly descending chain (x_i) in A.

Then $A = \{\, x_i \mid i \geq 0 \,\}$ has no least element.

On the other hand suppose A has no least element. Pick $x_0 \in A$ arbitrary.

By induction, choose an element $x_{n+1} \in A$ smaller than x_n, the current tail element of the sequence. Iterating we obtain a strictly descending chain.

Note that this direction requires something like the Axiom of Choice. □

- $\langle \mathbb{N}, < \rangle$
- $\langle \mathbb{N}, | \rangle$ where $|$ stand for "divides"
- words with length-lex ordering
- $\prec$ on $\mathbb{N} \times \mathbb{N}$ defined by

$$(a, b) \prec (a', b') \iff (a < a') \vee (a = a' \wedge b < b')$$

  This is the strict lexicographic order on $\mathbb{N}^2$.
- finite lists with length ordering or sublist ordering
- finite trees with subtree ordering

We'll come back to the last two examples.

- $\langle \mathbb{N}, > \rangle$
- $\langle \mathbb{Z}, < \rangle$ or $\langle \mathbb{Q}, < \rangle$ or $\langle \mathbb{R}, < \rangle$
- words with lexicographic ordering:
  $b > ab > aab > aaab > \ldots > a^n b > \ldots$
- triangles with inclusion (or size by area)
- $\langle \mathfrak{P}(\mathbb{N}), \subset \rangle$

Here is a nasty one: define a relation on $\mathbb{N}_+$ by the taking the transitive closure of

$$x \prec 2x \quad \text{and} \quad 3x + 1 \prec x \text{ for } x \text{ odd}, \; x > 1$$

Is there a descending chain?

To apply inductive reasoning, we have to deal with predicates that behave properly with respect to the well-founded relation. Here is the set-theoretic version of proper behavior.

Definition

Let $\langle A, \prec \rangle$ be well-founded.
$B \subseteq A$ is inductive if $\forall\, x \in A \big(\forall\, z \prec x\, (z \in B) \to x \in B\big)$.

Theorem (Induction Theorem)

Let $\langle A, \prec \rangle$ be well-founded and $B \subseteq A$ inductive. Then $B = A$.

Proof. The proof uses the (LEP) for the well-founded relation.

Suppose there is a counterexample, let a be the least such.

Then by definition, $\forall\, z \prec a\, z \in B$. But then $a \in B$, contradiction. □

It is time for a hard example of an inductive structure, certain tame subsets of the reals.

It was recognized more than a century ago that the collection of all subsets of the reals is hopelessly complicated. For example, one cannot even determine whether there is a set of reals whose cardinality lies strictly between the cardinality of $\mathbb{N}$ and $\mathbb{R}$ (Cantor's infamous Continuum Hypothesis).

Fortunately, for most purposes one only needs to deal with fairly tame sets of reals. Émile Borel came up with a particularly useful definition of a well-behaved yet large family of sets of reals.

Borel sets are defined inductively as follows.

- Induction and Recursion
- Natural Numbers
- Induction Proofs
- Well-Orders

5. Lists

- Trees

Fix some ground set A. Here is an inductive definition of the collection List(A) of all lists over A.

Suppose $a \in A$ and $L \in \mathsf{List}(A)$.

- Atom: the empty list nil.
- Constructor: the prepend operation prep(a, L).

To lighten notation we usually write $a :: L$ instead of prep(a, L).

What is ordinarily written as the list $(a_1, a_2, \ldots, a_n)$ is now represented by the composite object

$$a_1 :: a_2 :: \ldots :: a_n :: \mathsf{nil}$$

This is a basic idea in Lisp and has since been incorporated in many programming languages.

We can introduce destructors

$$\begin{aligned}\mathsf{head} &: \mathsf{List}(A) \to A\\ \mathsf{tail} &: \mathsf{List}(A) \to \mathsf{List}(A)\end{aligned}$$

such that

$$K = \mathsf{prep}(a, L) \quad \text{implies} \quad a = \mathsf{head}(K), L = \mathsf{tail}(K)$$

Note that both operations are undefined for nil but for $n > 0$ we have

$$L = a_1 :: a_2 :: \ldots :: a_n :: \mathsf{nil}$$

yields

$$\begin{aligned}\mathsf{head}(L) &= a_1\\ \mathsf{tail}(L) &= a_2 :: \ldots :: a_n :: \mathsf{nil}\end{aligned}$$

One usually does not bother to spell this out, but as before for $\mathbb{N}$, we want List(A) to be the least set that

- contains the atom nil
- contains prep(a, L) for any $a \in A$ whenever it contains L.

Just as in the case of $\mathbb{N}$, this minimality condition excludes weird and unintended monsters like infinite lists.

Theorem (Induction for Lists)

Suppose $X \subseteq \mathsf{List}(A)$, $\mathsf{nil} \in X$ *and for all* $L \in X$, $a \in A$ *we have* $\mathsf{prep}(a, L) \in X$.

Then $X = \mathsf{List}(A)$.

Informally: any property that nil has, and that is inherited by L from $\mathsf{tail}(L)$, must already hold for all lists.

As before for natural numbers, this inductive framework can be used to

- define operations on lists, and
- prove basic properties these operations.

Note that when $A = \{\bullet\}$ then we are basically dealing again with the natural numbers, so this is a direct generalization.

One could also use an append operation instead of prepend (see below for definitions). As a result, there are two types of induction.

Standard induction on the left:

- Base case (empty list): show $\varphi(\mathsf{nil})$
- Induction step:
  assuming $\varphi(L)$, show $\varphi(\mathsf{prep}(a, L))$

Alternatively, we can use induction on the right:

- Base case (empty list): show $\varphi(\mathsf{nil})$
- Induction step:
  assuming $\varphi(L)$, show $\varphi(\mathsf{app}(a, L))$

Careful, sometimes one version is significantly easier than the other.

Here is a definition of append in our framework.

$$\begin{aligned} \mathsf{app}(a, \mathsf{nil}) &= a :: \mathsf{nil} \\ \mathsf{app}(a, b :: L) &= b :: \mathsf{app}(a, L) \end{aligned}$$

Joining two lists together

$$\begin{aligned} \mathsf{join}(\mathsf{nil}, K) &= K \\ \mathsf{join}(a :: L, K) &= a :: \mathsf{join}(L, K) \end{aligned}$$

For legibility we often write $L :: a$ instead of $\mathsf{app}(a, L)$ and $K :: L$ instead of $\mathsf{join}(K, L)$.

Careful with parens, though. The law for append says $(b :: L) :: a = b :: (L :: a)$.

Erasing all occurrences of a from a list

$$\begin{aligned} \mathsf{erase}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{erase}(a :: L) &= \mathsf{erase}(L) \\ \mathsf{erase}(b :: L) &= b :: \mathsf{erase}(L) \qquad a \neq b \end{aligned}$$

Keeping the first occurrence of a:

$$\begin{aligned} \mathsf{keep1}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{keep1}(a :: L) &= a :: \mathsf{erase}(L) \\ \mathsf{keep1}(b :: L) &= b :: \mathsf{keep1}(L) \qquad a \neq b \end{aligned}$$

The objects involved need not all be lists.

For example, we can define the length of a list as follows:

$$\mathsf{len}(\mathsf{nil}) = 0$$
$$\mathsf{len}(a :: L) = \mathsf{len}(L) + 1$$

We will use the naturals informally here, but one could express everything quite easily in terms of the inductive definitions from above.

Claim

$\mathsf{len}(\mathsf{join}(K, L)) = \mathsf{len}(K) + \mathsf{len}(L)$

Claim: $\mathsf{len}(\mathsf{join}(K, L)) = \mathsf{len}(K) + \mathsf{len}(L)$

Proof.

Base case: $K = \mathsf{nil}$

$$\mathsf{len}(\mathsf{nil} :: L) = \mathsf{len}(L) = 0 + \mathsf{len}(L) = \mathsf{len}(\mathsf{nil}) + \mathsf{len}(L)$$

Induction step: $K = a :: A$.

$$\begin{aligned}\mathsf{len}((a :: A) :: L) &= \mathsf{len}(a :: (A :: L)) = 1 + \mathsf{len}(A :: L) \\ &= 1 + \mathsf{len}(A) + \mathsf{len}(L) = \mathsf{len}(K) + \mathsf{len}(L)\end{aligned}$$

□

Claim

The basic operations empty, head, tail *and* prep *can all be implemented in* $O(1)$ *time.*

- app, join and erase are all linear time.
- nodup is quadratic time.

Let $n = |L|$, from repeated calls to erase get $\sum_{i=1}^{n} O(n-i) = O(n^2)$.

There is a faster algorithm based on sorting (takes $O(n \log n)$ steps), followed by a scan (takes linear time).

Exercise

Do both algorithms produce the same output?

Forming all pairs (a_i, b_i) from two given lists $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$

$$\begin{aligned} \mathsf{pair}(\mathsf{nil}, \mathsf{nil}) &= \mathsf{nil} \\ \mathsf{pair}(a :: L, b :: K) &= (a, b) :: \mathsf{pair}(L, K) \end{aligned}$$

Recall that (a, b) is just short for $\mathsf{prep}(a, \mathsf{prep}(b, \mathsf{nil}))$.

Note that this operation assumes input lists are of equal length. The output type is a list of lists.

Exercise

Implement a class `NList` that provides arbitrarily nested lists of, say, integers, together with a nice collection of operations.

Here is a definition of the reversal operation on lists:

$$\mathsf{rev}(\mathsf{nil}) = \mathsf{nil},$$
$$\mathsf{rev}(a :: L) = \mathsf{rev}(L) :: a$$

If you worry about implementation this may look unappealing: append as defined is linear time on singly-linked lists, so this definition would produce a quadratic time reversal.

Solution: change the data structure.

Don't worry about implementation details too soon.

Claim

$\mathsf{rev}(L :: a) = a :: \mathsf{rev}(L)$ *for all* L, a.

Proof.

Base case: $L = \mathsf{nil}$

$$\begin{aligned}\mathsf{rev}(\mathsf{nil} :: a) &= \mathsf{rev}(a :: \mathsf{nil}) \\ &= \mathsf{rev}(\mathsf{nil}) :: a \\ &= \mathsf{nil} :: a \\ &= a :: \mathsf{nil}\end{aligned}$$

Induction step: let $L = b :: K$.

$$\begin{aligned}\mathsf{rev}((b :: K) :: a) &= \mathsf{rev}(b :: (K :: a)) \\ &= \mathsf{rev}(K :: a) :: b \\ &= (a :: \mathsf{rev}(K)) :: b \\ &= a :: (\mathsf{rev}(K) :: b) \\ &= a :: \mathsf{rev}(b :: K)\end{aligned}$$

Note that every single step in this type of proof is really simple: we only need to decide which axiom to use and when to apply the induction hypothesis.

This type of argument is called equational logic and is relatively easy to automate.

Alas, for humans it's not so simple: everyone's eyes glaze over after half a dozen steps. Plus, it's really easy to make silly mistakes.

Exercise

Prove the following claims by induction on lists.

Claim

$\mathsf{rev}(L :: K) = \mathsf{rev}(K) :: \mathsf{rev}(L)$ *for all* L, K.

Claim

$\mathsf{rev}(\mathsf{rev}(L)) = L$ *for all* L.

Exercise

Write $\mathsf{rot}(L)$ *for the result of rotating* L *cyclically by one place to the left. Give an inductive definition of* rot *and characterize the lists* L *such that* $\mathsf{rot}(L) = L$.

Problem:	**Rotation**
Instance:	An array of A, a positive integer s.
Solution:	Rotate A by s places.

Of course, the challenge is to do this with minimal resources.

How about linear time and $O(1)$ extra space?

This is surprisingly difficult. Clearly, we can rotate by one place in linear time and $O(1)$ extra space. But we cannot repeat $s = O(n)$ times without violating the linearity constraint.

Alternatively, we can use scratch space $O(s)$ to move the first s elements out of the way, and move everything in linear time, but that violates the space constraint.

A clever and far from obvious trick is to use reversal to implement rotation. The key observation is that

$$\mathsf{rot}(u :: v, s) = \mathsf{rev}(\mathsf{rev}(u) :: \mathsf{rev}(v))$$

where u has length s.

In other words, reverse the initial segment of A of length s, then reverse the remainder, and in one last step reverse the whole array.

Since reversal can clearly be handled in linear time and $O(1)$ extra space, done.

```
// reverse block from lo to hi, inclusive
void  reverse( int lo, int hi ) {
   int i,j, m = (hi-lo)/2;
   for( i=lo,j=hi; i<m; i++,j-- )
                swap( i, j );
}

// rotate left, len length of array
void  rotate_left( int s ) {
        s = s mod len;
        reverse( 0, s-1   );
        reverse( s, len-1 );
        reverse( 0, len-1 );
}
```

Exercise

What happens if we perform the `reverse(0, len-1)` *operation first?*

Define a partial order on List(A) by taking the transitive closure of $L \prec \mathsf{prep}(a, L)$.

Lemma

$\prec$ is well-founded.

This is essentially induction on the length of the list: assume claim is true for shorter lists.

Boils down to

- Base case (empty list): show $\varphi(\mathsf{nil})$
- Induction step:
  show $\varphi(\mathsf{prep}(a, L))$ assuming $\varphi(L)$.

Alternatively, we can use "induction on the right":

- show $\varphi(\mathsf{app}(a, L))$ assuming $\varphi(L)$.

Note, though, that in the usual singly-linked pointer implementation, recursion on the left is more efficient.

Forming all pairs (a_i, b_i) from two given lists $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$

$$\begin{aligned} \mathsf{pair}(\mathsf{nil}, \mathsf{nil}) &= \mathsf{nil} \\ \mathsf{pair}(a :: L, b :: K) &= (a, b) :: \mathsf{pair}(L, K) \end{aligned}$$

Recall that (a, b) is just short for $\mathsf{prep}(a, \mathsf{prep}(b, \mathsf{nil}))$.

Note that this operation assumes input lists are of equal length. The output type is a list of lists.

Exercise

Implement a class `NList` *that provides arbitrarily nested lists of, say, integers, together with a nice collection of operations.*

Problem:	**Rotation**
Instance:	An array of A, a positive integer s.
Solution:	Rotate A by s places.

Of course, the challenge is to do this with minimal resources.

How about linear time and $O(1)$ extra space?

This is surprisingly difficult. Clearly, we can rotate by one place in linear time and $O(1)$ extra space. But we cannot repeat $s = O(n)$ times without violating the linearity constraint.

Alternatively, we can use scratch space $O(s)$ to move the first s elements out of the way, and move everything in linear time, but that violates the space constraint.

A clever and far from obvious trick is to use reversal to implement rotation. The key observation is that

$$\mathsf{rot}(u :: v, s) = \mathsf{rev}(\mathsf{rev}(u) :: \mathsf{rev}(v))$$

where u has length s.

In other words, reverse the initial segment of A of length s, then reverse the remainder, and in one last step reverse the whole array.

Since reversal can clearly be handled in linear time and $O(1)$ extra space, done.

```
// reverse block from lo to hi, inclusive
void  reverse( int lo, int hi ) {
   int i,j, m = (hi-lo)/2;
   for( i=lo,j=hi; i<m; i++,j-- )
                swap( i, j );
}

// rotate left, len length of array
void  rotate_left( int s ) {
        s = s mod len;
        reverse( 0, s-1   );
        reverse( s, len-1 );
        reverse( 0, len-1 );
}
```

Exercise

What happens if we perform the `reverse(0, len-1)` *operation first?*

- Primitive elements are all open sets.
- There are two constructors: complement and countable union.

So initially we have only open sets.

By complementation we obtain all closed sets.

Countable unions of closed sets are known in analysis as F_σ sets.

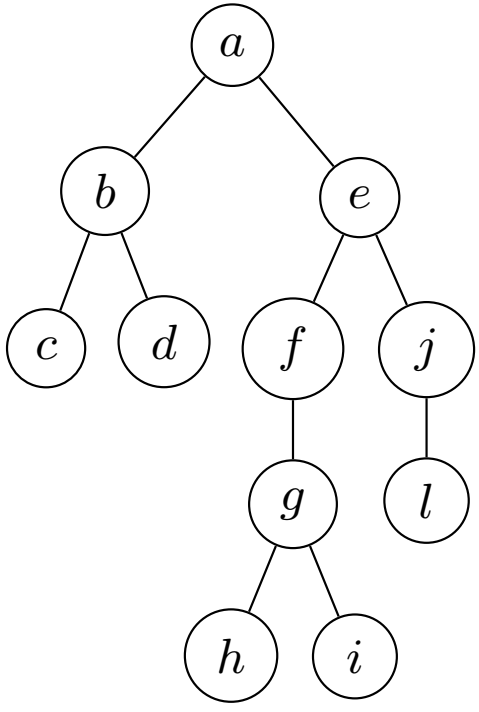
Countable intersections of open sets are G_δ, and so on and so forth.

One can show that this hierarchy is proper and extends all the way to the first uncountable ordinal (there are as many Borel sets as there are reals).

One important property of Borel sets is that they are measurable; another is that they obey the Continuum Hypothesis.

- Induction and Recursion
- Natural Numbers
- Induction Proofs
- Well-Orders
- Lists

6. Trees



a is the root, c, d, h, i, j, l are leaves, everybody else (including the root) is an interior node.

A branch is a root-to-leaf path such as a, e, f, g, i.

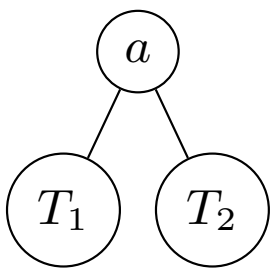
The depth of the tree is the maximum length of any branch. The sample tree has depth 4.

Data structures for trees are very similar to lists.

For simplicity, consider binary trees over a groundset A. Atoms and constructors are as follows:

- The empty tree nil is in BTree(A).
- For $a \in A$ and $T_1, T_2 \in$ BTree(A), cons(a, T_1, T_2) is in BTree(A).

So here nil stands for the empty tree, and cons(a, T_1, T_2) is the tree



We can dismantle a non-empty tree like so:

- cons$(a, T, T') \neq$ nil.
- cons$(a, T_1, T_2) =$ cons(b, T_1', T_2') implies $a = b$ and $T_i = T_i'$.

Thus we obtain an element in the groundset (the label of the root) plus two trees, and these pieces are uniquely determined by the given tree.

Theorem (Induction for Trees)

Suppose $X \subseteq \mathsf{BTree}(A)$, $\mathsf{nil} \in X$ *and for all* $T_i \in X$, $a \in A$ *we have* $\mathsf{cons}(a, T_1, T_2) \in X$. *Then* $X = \mathsf{BTree}(A)$.

The underlying well-founded partial order is the transitive closure of

$$T_1, T_2 \prec \mathsf{cons}(a, T_1, T_2)$$

Exercise

Explain what this has to do with induction on the depth of the tree.

Exercise

Generalize to other kinds of trees

As an example of an inductively defined operation, consider the depth function for finite trees.

$$d(\mathsf{nil}) = -1$$
$$d(\mathsf{cons}(a, T_1, T_2)) = \max(d(T_1), d(T_2)) + 1$$

Note the depth of the empty tree.

And here is the number of nodes in the tree.

$$nc(\mathsf{nil}) = 0$$
$$nc(\mathsf{cons}(a, T_1, T_2)) = nc(T_1) + nc(T_2) + 1$$

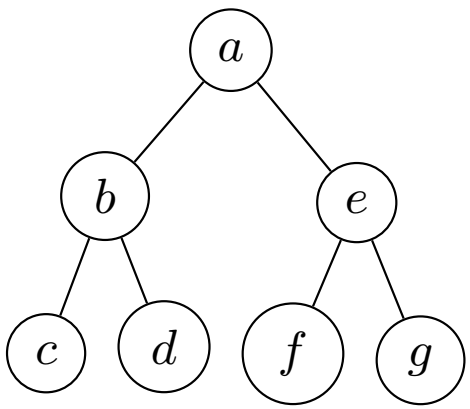
Exercise

Find an inductive way to count the number of leaves in a tree.

Here is an operation that turns a tree into a linear list, inorder traversal.

$$\mathsf{flat}(\mathsf{nil}) = \mathsf{nil}_L$$
$$\mathsf{flat}(\mathsf{cons}(a, T_1, T_2) = \mathsf{join}(\mathsf{flat}(T_1), (a), \mathsf{flat}(T_2))$$

We have written nil_L for the empty list to avoid confusion.



flattens out to (d,b,e,a,f,c,g)

Definition

A labeled binary tree T is a binary search tree (BST) if $flat(T)$ is sorted.

We can search very efficiently in a BST:

$$\begin{aligned}
\mathsf{search}(\mathsf{nil}, a) &= \mathsf{ff} \\
\mathsf{search}(\mathsf{cons}(a, T_1, T_2), a) &= \mathsf{tt} \\
\mathsf{search}(\mathsf{cons}(b, T_1, T_2), a) &= \mathsf{search}(T_1, a) && \text{if } a < b \\
\mathsf{search}(\mathsf{cons}(b, T_1, T_2), a) &= \mathsf{search}(T_2, a) && \text{if } a > b
\end{aligned}$$

This algorithm only walks down one branch, so most nodes in the tree are never touched.

Thus, the worst case cost of a search is the depth of the tree.

Binary trees are important since they are particularly easy to implement.

Question: Which data structure is more expressive: Binary trees or nested lists?

We can convert nested lists (over some ground set A) into binary trees (with leaves labeled in A), and back.

$$l2t : \mathsf{NList} \to \mathsf{BTree}$$
$$t2l : \mathsf{BTree} \to \mathsf{NList} \qquad \text{(partial)}$$

We want

$$l2t \circ t2l = I_{\mathsf{NList}} \qquad \text{and} \qquad t2l \circ l2t \subseteq I_{\mathsf{BTree}}$$

The map $t2l$ is partial since not every tree corresponds to a nested list.

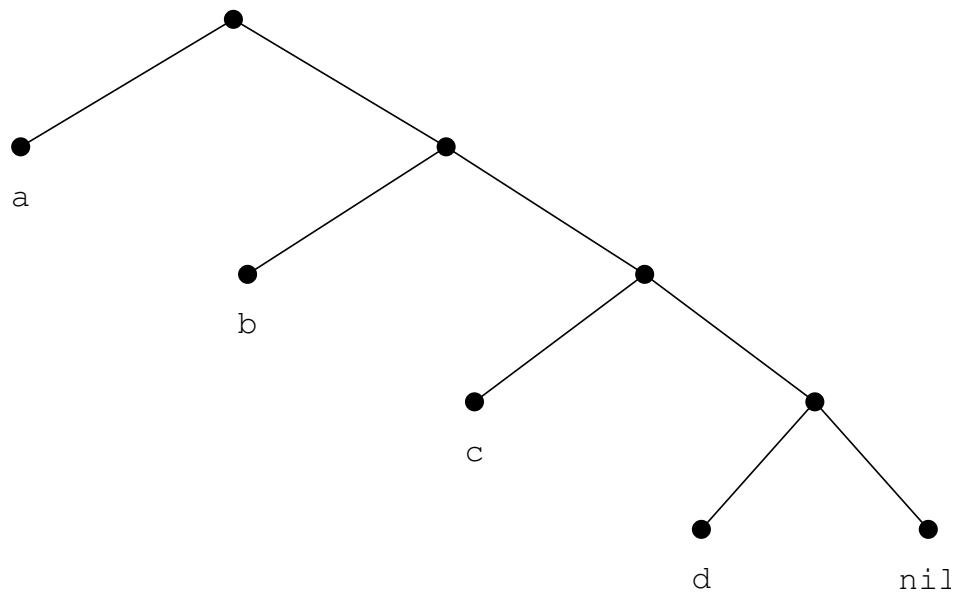
Again we write nil_L for the empty list and nil for the empty tree.

$$l2t(\mathsf{nil}_L) = \mathsf{nil}$$
$$l2t(a :: L) = \mathsf{cons}(a, l2t(L))$$
$$l2t(K :: L) = \mathsf{cons}(l2t(K), l2t(L))$$

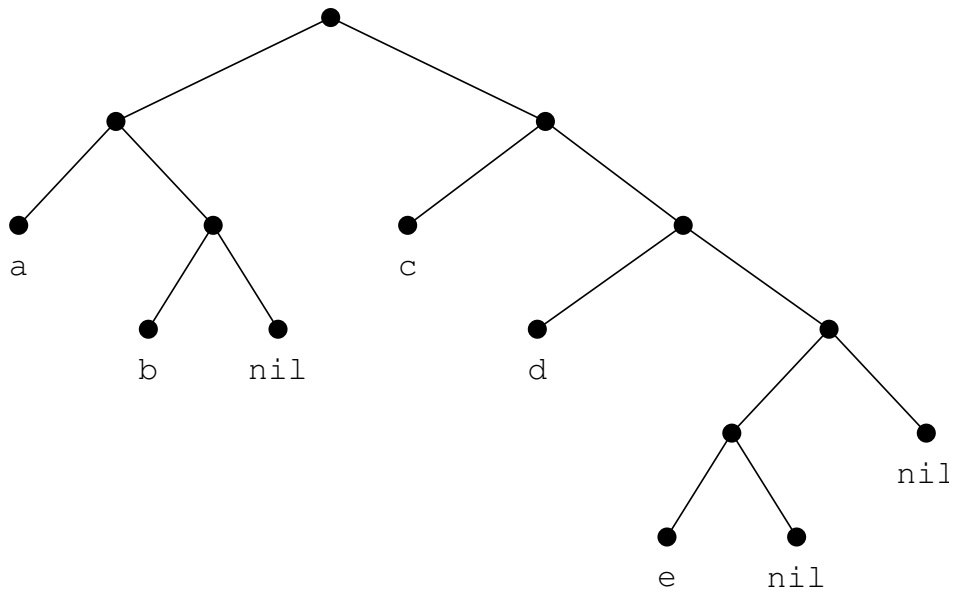
We have slightly adjusted the cons operation since only leaves are labeled.

Example

List (a, b, c, d) translates into the following binary tree.



Translating the nested list $((a, b), c, d, (e))$.



Note the subtrees for (a, b) and (e).

Now in the opposite direction:

$$t2l(\mathsf{nil}) = \mathsf{nil}_L$$
$$t2l(\mathsf{cons}(a, T_2)) = a :: t2l(T_2)$$
$$t2l(\mathsf{cons}(T_1, T_2)) = t2l(T_1) :: t2l(T_2)$$

Exercise

Find a simple decision procedure for the domain of $t2l$.

Exercise

Prove that these functions are mutually inverse (in the sense specified a while ago).