

Recursion and trees



The cow shown laughing on the Laughing Cow® box holds, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably — I say “presumably” because here my eyesight fails me, I don’t know about yours — holding, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably holding... (you get the idea).

This 1921 advertising gimmick, still doing very well, is an example of a structure defined *recursively*, in the following sense:

www.bel-group.com.
Picture credit:
page 847.

Recursive definition

A definition for a concept is recursive if it involves one or more instances of the concept itself.

“*Recursion*” — the use of recursive definitions — has applications throughout programming: it yields elegant ways to define *syntax structures*; we will also see recursively defined *data structures* and *routines*.

We may say “*recursive*” as an abbreviation for “recursively defined”: recursive grammar, recursive data structure, recursive routine. But this is only a convention, because we cannot say that a concept or a structure is by itself recursive: all we know is that we can *describe* it recursively, according to the above definition. Any particular notion — even the infinite Laughing Cow structure — may have both recursive and non-recursive definitions.

When proving properties of recursively defined concepts we will use recursive *proofs*, which generalize inductive proofs as performed on integers.

Recursion is *direct* when the definition of A cites an instance of A ; it is *indirect* if for $1 \leq i < n$ (for some $n \geq 2$) the definition of every A_i cites an instance of A_{i+1} , and the definition of A_n cites an instance of A_1 .

In this chapter we are interested in notions for which a recursive definition is elegant and convenient. The examples include recursive routines, recursive syntax definitions and recursive data structures. We will also get a glimpse of recursive proofs.

One class of recursive data structures, the *tree* in its various guises, appears in many applications and embodies the very idea of recursion. This chapter covers the important case of *binary* trees.

14.1 BASIC EXAMPLES

At this point you may be wondering whether a recursive definition makes any sense at all. How can we define a concept in terms of itself? Does such a definition mean anything at all, or is it just a vicious circle?

You are right to wonder. Not all recursive definitions define anything at all. When you ask for a description of someone and all you get is “*Sarah? She is just Sarah, what else can I say?*” you are not learning much. So we will have to look for criteria that guarantee that a definition is useful even if recursive.

Before we do this, however, let us convince ourselves in a more pragmatic way by looking at a few typical examples where recursion is obviously useful and seems, just as obviously, to make sense. This will give us a firm belief — little more than a belief indeed, based on hope and a prayer — that recursion is a practically useful way to define grammars, data structures and algorithms. Then it will be time to look for a proper mathematical basis on which to establish the soundness of recursive definitions.

→ “*Making sense of recursion*”, 14.7, page 473.

Recursive definitions

With the introduction of genericity, we were able to define a *type* as either:

T1 A non-generic class, such as *INTEGER* or *STATION*.

T2 A generic derivation, of the form $C [T]$, where C is a generic class and T is a type.

← “*Definitions: Class type, generically derived, base class*”, page 370.

This is a recursive definition; it simply means, using the generic classes *ARRAY* and *LIST*, that valid classes are:

- *INTEGER*, *STATION* and such: non-generic classes, per case T1.
- Through case T2, direct generic derivations: *ARRAY [INTEGER]*, *LIST [STATION]* etc.
- Applying T2 again, recursively: *ARRAY [LIST [INTEGER]]*, *ARRAY [ARRAY [LIST [STATION]]]* and so on: generic derivations at any level of nesting.

You may consider using a similar technique to answer the exercise which, in the first chapter, asked you to define “alphabetical order”.

← 1-E.3, page 14.

Recursively defined grammars

Consider an Eiffel subset with just two kinds of instruction:

- Assignment, of the usual form *variable* := *expression*, but treated here as a terminal, not specified further.
- Conditional, with only a **then** part (no **else**) for simplicity.

← This discussion was previewed in “Recursive grammars”, page 307.

A grammar defining this language is:

```

Instruction  $\triangleq$  Assignment | Conditional
Conditional  $\triangleq$  if Condition then Instruction end
```

For our immediate purposes **Condition** is, like **Assignment**, a terminal. This grammar is recursive, since the definition of **Instruction** involves **Conditional** as one of the choices, and **Conditional** in turn involves **Instruction** as part of the aggregate. But since there is a non-recursive alternative, **Assignment**, the grammar productions clearly imply what an instruction may look like:

- Just an assignment.
- A **Conditional** containing an assignment: **if *c* then *a* end**.
- The same with any degree of nesting: **if *c*₁ then if *c*₂ then *a* end end, if *c*₁ then if *c*₂ then if *c*₃ then *a* end end end** and so on.

Recursive grammars are indeed an indispensable tool for any language that — like all significant programming languages — supports nested structures.

Recursively defined data structures

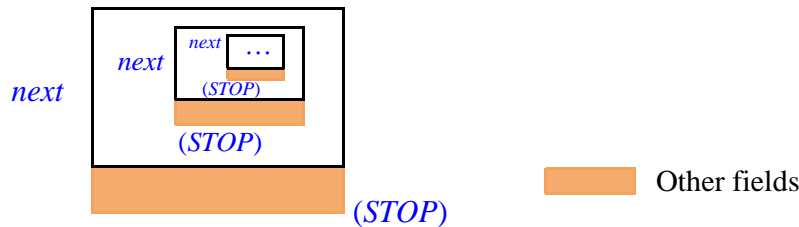
The class *STOP* represented the notion of stop in a metro line:

← Page 123.

```

class STOP create
  ...
feature
  next: STOP
    -- Next stop on same line.
  ... Other features omitted (see page 123) ...
end
```

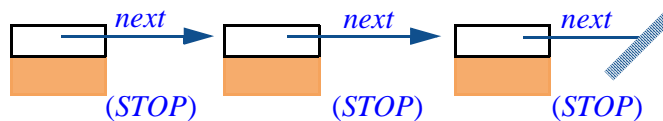
A naïve interpretation would deduce that every instance of *STOP* contains an instance of *STOP*, which itself contains another ad infinitum, as in the Laughing Cow scheme. This would indeed be the case if *STOP* were an expanded type:



*Nested fields
(not the correct
interpretation)*

This is impossible, however, and *STOP* is in any case a **reference** type, like any type defined as `class X ...` with no other qualification. So the real picture is the one originally shown:

← Page 116.



A linked line

Recursion in such a data structure definition simply indicates that every instance of the class contains a reference to a potential instance of the same class — “potential” because the reference may be void, as for the last stop in the figure.

In the same chapter we encountered another example of self-referential class definition: a class *PERSON* with an attribute *spouse* of type *PERSON*.

This is a very common case in definitions of useful data structures. From linked lists to *trees* of various kinds (such as the binary trees studied later in this chapter), the definition of a useful object type often includes references to objects of the type being defined, or (indirect recursion) a type that depends on it.

Recursively defined algorithms and routines

The famous Fibonacci sequence, enjoying many beautiful properties and many applications to mathematics and the natural sciences, has the following definition:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \quad \text{-- For } i > 1 \end{aligned}$$

Touch of History: Fibonacci's rabbits

Leonardo Fibonacci from Pisa (1170-1250) played a key role in making Indian and Arab mathematics known to the West and, through many contributions of his own, helping to start modern mathematics. He stated like this the problem that leads to his famous sequence (which was already known to Indian mathematicians):

About Fibonacci:
[www.mcs.surrey.ac.uk/
Personal/R.Knott/
Fibonacci/fib.html](http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html);
about the sequence:
[www-gap.dcs.st-and.
ac.uk/~history/Mathema
ticians/Fibonacci.html](http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Fibonacci.html).

A man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if every month each pair begets a new pair, which becomes productive from the second month on?

The answer is that the pairs at month i include those already present at month $i - 1$ (no rabbits die), numbering F_{i-1} , plus those begot by pairs already present at month $i - 2$ (since pairs are fertile starting the second month), numbering F_{i-2} . This gives the above formula; successive values are 0, 1, 1, 2, 3, 5, 8 and so on, each the sum of the previous two.

The formula yields a recursive routine computing F_n for any n :

```

fibonacci (n: INTEGER): INTEGER
-- Element of index n in the Fibonacci sequence.
require
  non_negative: n >= 0
do
  if n = 0 then
    Result := 0
  elseif n = 1 then
    Result := 1
  else
    Result := fibonacci (n - 1) + fibonacci (n - 2)
  end
end

```

Programming Time! **Recursive Fibonacci**

Write a small system that includes the above recursive routine and prints out its result. Try it for a few values of n — including 12, as in Fibonacci's original riddle — and verify that the results match the expected values.

The function includes two recursive calls, highlighted. That it works at all may look a bit mysterious (that's why it is good to check it for a few values); as you progress through this chapter, the legitimacy of such recursively defined routines should become increasingly convincing.

The principal argument in favor of writing the function this way is that it elegantly matches the original, mathematical definition of the Fibonacci sequence. On further look it is not that exciting, because a non-recursive version is also easy to obtain.

Programming Time! **Non-recursive Fibonacci**

Can you write (without first turning the page) a function that computes any Fibonacci number, using a loop rather than recursion?



Fibonacci

The following function indeed yields the same result as the above *fibonacci* (try it for a few values too):

```

fibonacci1 (n: INTEGER): INTEGER
  -- Element of index n in the Fibonacci sequence.
  -- (Non-recursive version.)
require
  positive: n >= 1
local
  i, previous, second_previous: INTEGER
do
from
  i := 1 ; Result := 1
invariant
  Result = fibonacci (i)
  previous = fibonacci (i - 1)
until i = n loop
  i := i + 1
  second_previous := previous
  previous := Result
  Result := previous + second_previous
variant
  n - i
end
end

```

For convenience this version assumes $n \geq 1$ rather than $n \geq 0$. Thanks to the initialization rules *previous* starts out as zero, ensuring the initial satisfaction of the invariant since $F_0 = 0$. The variable *second_previous* is set anew in each loop iteration and does not need specific initialization.

This version, just a trifle more remote from the original mathematical definition, is still simple and clear; note in particular the loop invariant (which, however, refers for convenience to the recursive function, which it takes as the official mathematical definition). Some may prefer the recursive version anyway, but this is largely a matter of taste. Depending on the compiler, that version may (as we will see) be less efficient at run time.

Taste and efficiency aside, if it were only for such examples we would have a hard time convincing ourselves of the indispensability of recursive routines. We need cases in which recursion provides a definite plus, for example because any non-recursive competitor is significantly more abstruse.

Such problems indeed abound. One that concentrates many of the interesting properties of recursion, with the smallest amount of irrelevant detail, arises from a delightful puzzle: the Tower of Hanoi.

14.2 THE TOWER OF HANOI

In the great temple of Benares, under the dome that marks the center of the world, three needles of diamond are set on top of a brass plate. Each needle is a cubit high, and thick as the body of a bee. On one of these needles God strung, at the beginning of ages, sixty-four disks of pure gold. The largest disk rests on the brass and the others, ever smaller, rest over each other all the way to the top. That is the sacred tower of Brahma.

Night and day the priests, following one another on the steps of the altar, work to transfer the tower from the first diamond needle to the third, without deviating from the rules just stated, set by Brahma. When all is over, the tower and the Brahmins will fall, and it will be the end of the worlds.



*Tower of Hanoi
(or should it be
Benares?) with 9
disks, initial state*

In spite of its oriental veneer, this story is the creation of the French mathematician Édouard Lucas (signing as “N. Claus de Siam”, anagram of “Lucas d’Amiens”, after his native city). On a market in Thailand — Siam indeed — I bought the above rendition of his tower. The labels **A**, **B** and **C** are my addition. I will not expand on why I chose a model made of wood rather than diamond, gold and brass, but it is legitimate, since I did have a large suitcase, to ask why it has only nine disks:

Quiz time!
Hanoi tower size

Why do commercially available models of the Towers of Hanoi puzzle have far fewer than 64 disks?

(Hint: the game comes with a small sheet of paper listing a solution to the puzzle, in the form of a sequence of moves: **A** to **C**, **A** to **B** etc.)

To answer this question, we may assess the minimum number H_n of individual “move” operations required — if there is a solution — to transfer n disks from needle **A** to needle **B**, using needle **C** as intermediate storage and following the rules of the game; n is 64 in the original version and 9 for the small model.

We observe that any strategy for moving n disks from **A** to **B** must at some point move the largest disk from **A** to **B**. This is only possible, however, if needle **B** is free of any disks at all, and **A** contains only the largest disk, all others having been moved to **C** — since there is no other place for them to go:



Intermediate state

What is the minimum number of moves to reach this intermediate situation? We must have transferred $n - 1$ disks (all but the largest) from **A** to **C**, using **B** as intermediate storage; the largest disk, which must stay on **A**, plays no role in this operation. The problem is symmetric between **B** and **C**; so the minimum number of moves to achieve the intermediate situation is H_{n-1} .

Once we have reached that situation, we must move the largest disk from **A** to **B**; it remains then to transfer the $n - 1$ smaller disks from **C** to **B**. In all, the minimum number of moves H_n for transferring n disks, for $n > 0$, is

$$H_n = 2 * H_{n-1} + 1$$

(H_{n-1} moves to transfer $n - 1$ disks from **A** to **C**, one move to take the largest disk from **A** to **B**, and H_{n-1} again to transfer the $n - 1$ smaller disks from **A** to **C**). Since $H_0 = 0$, this gives

$$H_n = 2^n - 1$$

and, as a consequence, the answer to our quiz: remembering that 2^{10} (that is, 1024) is over 10^3 , we note that 2^{64} is over $1.5 * 10^{19}$; that’s a lot of moves.

A year is around 30 million seconds. At one second per move — very efficient priests — the world will collapse in about 500 billion years, over 30 times the estimated age of the universe. As to the paper for printing the solution to a 64-disk game, it would require cutting down the forests of a few planets.

This reasoning for the evaluation of H_n was *constructive*, in the sense that it also gives us a **practical strategy** for moving n disks (for $n > 0$) from **A** to **B** using **C** as intermediate storage:

- Move $n - 1$ disks from **A** to **C**, using **B** as intermediate storage, and respecting the rules of the game.
- Then **B** will be empty of any disk, and **A** will only have the largest disk; transfer that disk from **A** to **B**. This respects the rules of the game since we are moving a single disk, from the top of a needle, to an empty needle.
- Then move $n - 1$ disks from **C** to **B**, using **A** as intermediate storage, respecting the rules of the game; **B** has one disk, but it will not cause any violation of the rules of the game since it is larger than all the ones we want to transfer.

This strategy turns the number of moves $H_n = 2^n - 1$ from a theoretical minimum into a practically achievable goal. We may express it as a recursive routine, part of a class *NEEDLES*:

```

hanoi (n: INTEGER; source, target, other: CHARACTER)
  -- Transfer n disks from source to target, using other as
  -- intermediate storage, according to the rules of the
  -- Tower of Hanoi puzzle.
require
  non_negative: n >= 0
  different1: source /= target
  different2: target /= other
  different3: source /= other
do
  if n > 0 then
    hanoi (n-1, source, other, target)
    move (source, target)
    hanoi (n-1, other, target, source)
  end
end

```

The discussion of contracts for recursive routines will add other precondition clauses and a postcondition.

By convention, we represent the needles as characters: 'A', 'B' and 'C'. Another convention for this chapter (already used in previous examples) is to highlight recursive branches; *hanoi* contains two such calls.

2^{13} sheets per tree
(tinyurl.com/6azah);
 2^{10} moves per page
(very small print);
double-sided since we
are environmentally
conscious; maybe 400
billion (over 2^{38})
usable trees on earth
(tinyurl.com/yfpppyd):
adding three similar
planets will get
us started.

→ “Contracts for recursive routines”, 14.8, page 485.

The basic operation *move* (*source*, *target*) moves a single disk, the top one on needle *source*, to needle *target*; its precondition is that there is at least one disk on *source*, and that on *target* either there is no disk or the top disk is larger than the top disk on *source*. If you have access to the wireless network of the Great Temple of Benares you can program *move* to send an instant message to the cell phone of the appropriate priest or an email to her Blackberry, directing her to move a disk from *source* to *target*. For the rest of us you can write *move* as a procedure that displays a one-disk-move instruction in the console:

```

move (source, target: CHARACTER)
  -- Prescribe move from source to target.
  do
    io.put_character (source)
    io.put_string (" to ")
    io.put_character (target)
    io.put_new_line
  end

```

Programming Time! **The Tower of Hanoi**

Write a system with a root class *NEEDLES* including the procedures *hanoi* and *move* as shown. Try it for a few values of *n*.

For example executing the call

```
hanoi (4, 'A', 'B', 'C')
```

will print out the sequence of fifteen ($2^4 - 1$) instructions

A to C	B to C	B to A
A to B	A to C	C to B
C to B	A to B	A to C
A to C	C to B	A to B
B to A	C to A	C to B

Shown here split into three columns; read it column by column, top to bottom in each column. The move of the biggest disk has been highlighted.

which indeed moves four disks successfully from **A** to **B**, respecting the rules of the game.

One way to look at the recursive solution — procedure *hanoi* — is that it works as if we were permitted to move the top $n-1$ disks all at once to a needle that has either no disk, or the biggest disk only. In that case we would start by performing this operation from *source* to *other* (here **A** to **C**):



*Fictitious initial
global move*

Then we would move the biggest disk from **A** to **B**, our final *target*; this single-disk move is clearly legal since there is nothing on **B**. Finally we would again perform a global move of $n-1$ disks from **C**, where we have parked them, to **B**, which is OK because they are in order and the largest of them is smaller than the disk now on **B**.

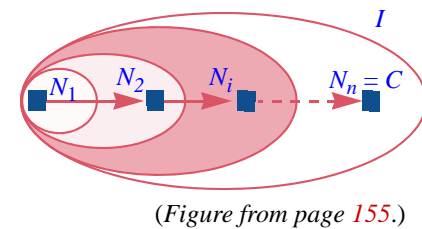
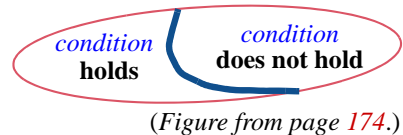
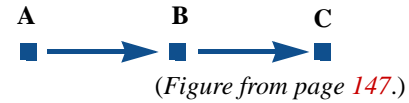
Of course this is a fiction since we are only permitted to move one disk at a time, but to move $n-1$ disks we may simply apply the same technique recursively, knowing that the target needle is either empty or occupied by a disk larger than all those we manipulate in this recursive application. If $n = 0$, we have nothing to do.

Do not be misled by the apparent frivolity of the Tower of Hanoi example. The solution serves as a model for many recursive programs with important practical applications. The simplicity of the algorithm, resulting from the use of two recursive calls, makes it an ideal testbed to study the properties of recursive algorithms, as we will do when we return to it later in this chapter.

14.3 RECURSION AS A PROBLEM-SOLVING STRATEGY

In earlier chapters we saw control structures as problem-solving techniques:

- A **compound** (sequence) solution means “I know someone who can get me from here to B and someone else who can get me from B to C, so let me ask them one then the other and that will get me to C”.
- A **conditional** solution means “I know someone who can solve the problem in one case and someone else for the other possible case, so let me ask them separately”.
- A **loop** solution means “I do not know how to get to C, but I know a region *I* (the invariant) that contains C, someone (the initialization) to take me into *I*, and someone else (the body) who whenever I am in *I* and not yet in C can take me closer to C, decreasing the distance (the variant) in such a way that I will need her only a finite number of times; so let me ask my first friend once to get into *I*, then bug my other friend as long as I have not reached C yet”.
- A **routine** solution means “I know someone who has solved this problem in the general case, so let me just phrase my special problem in his terms and ask him to solve it for me”.



What about a recursive solution? Whom do I ask?

I ask myself.

Possibly several times! (As in the *Hanoi* case and many to follow.)

Why rely on someone else when I trust myself so much more? (At least I think I do.)

By now we know that this strategy is not as silly as it might sound at first. I ask myself to solve the same problem, but on a **subset** of the original data, or several such subsets. Then I may be able to pull it off, if I have a way to extend these partial solutions into a solution to the entire problem.

Such is the idea of recursion viewed as a general problem-solving strategy. It is related to some of the earlier strategies:

- Recursion resembles the *routine* strategy, since it relies on an existing solution, but in this case we use a solution to the *same problem* — not only that, the *same solution* to that problem: the solution that we are currently building and that we just pretend, by a leap of faith, already exists. ← Chapter 8.
- Recursion also shares properties with a *loop* solution: both techniques approximate the solution to the whole problem by solutions covering part of the data. But recursion is more general, since each step may combine more than one such partial solution. Later in this chapter we will have the opportunity of comparing the loop and recursion strategies in detail. ← “The loop strategy”, page 155.
← “From loops to recursion”, 14.6, page 471.

14.4 BINARY TREES

If the Tower of Hanoi solution is the quintessential recursive routine, the binary tree is the quintessential recursive data structure. We may define it as follows:

Definition: binary tree

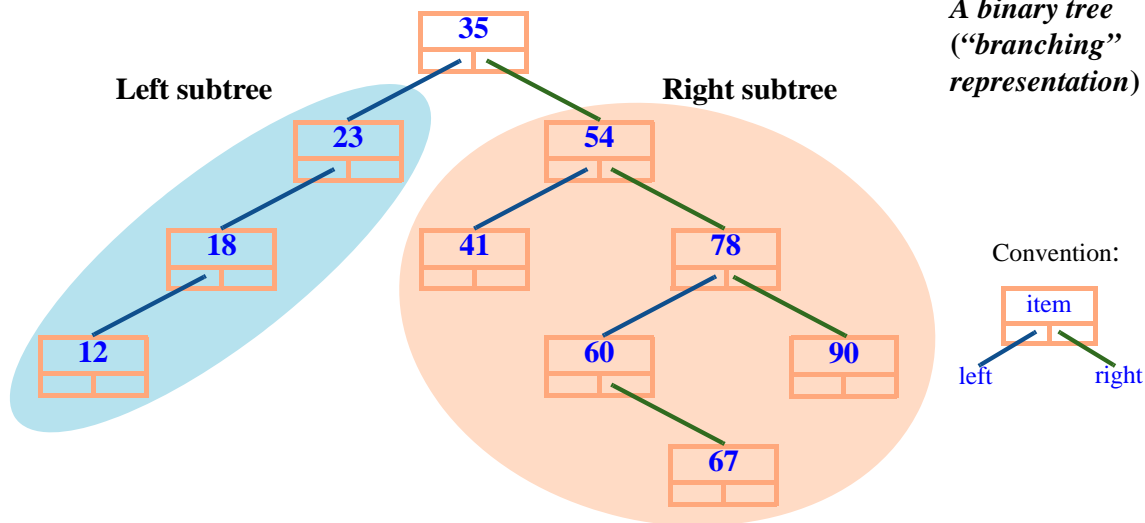
A binary tree over G , for an arbitrary data type G , is a finite set of items called **nodes**, each containing a value of type G , such that the nodes, if any, are divided into three disjoint parts:

- A single node, called the **root** of the binary tree.
- (Recursively) two **binary trees** over G , called the *left subtree* and *right subtree*.

It is easy to express this as a class skeleton, with no routines yet:

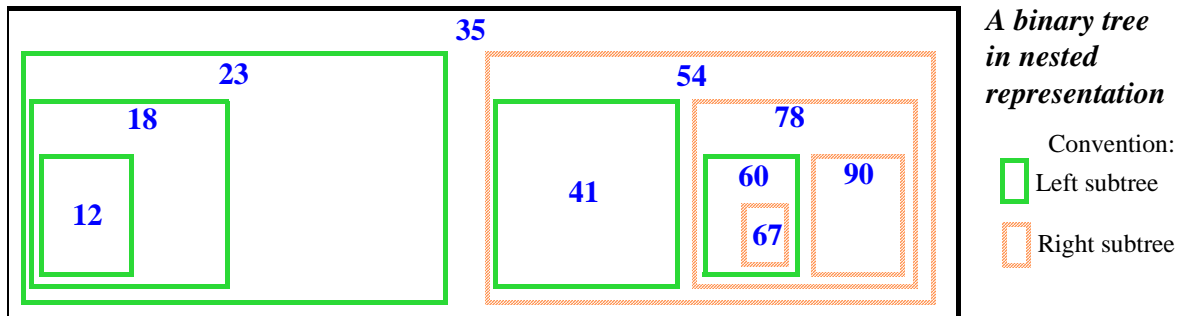
```
class BINARY_TREE [G] feature
  item: G
  left, right: BINARY_TREE [G]
end
```

where a void reference indicates an empty binary tree. We may illustrate a binary tree — here over *INTEGER* — as follows:



This “branching” form is the most common style of representing a binary tree, but not the only one; as in the case of abstract syntax trees, we might opt for a *nested* representation, which here would look like the following.

← “*Nesting and the syntax structure*”, page 40.



The definition explicitly allows a binary tree to be empty (“the nodes, if any”). Without this, of course, the recursive definition would lead to an infinite structure, whereas our binary trees are, as the definition also prescribes, finite.

If not empty, a binary tree always has a root, and may have: no subtree; a left subtree only; a right subtree only; or both.

Any node n of a binary tree B itself defines a binary tree B_n . The association is easy to see in either of the last two figures: for the node labeled **35**, B_n is the full tree; for **23** it is the left subtree; for **54**, the right subtree; for **78**, the tree rooted at that node (right subtree of the right subtree); and so on. This allows us to talk about the left and right subtrees of a *node* — meaning, of its associated subtree. We can make the association formal through another example of recursive definition, closely following the structure of the definition of binary trees:

Definition: Tree associated with a node

Any node n of a binary tree B defines a binary tree B_n as follows:

- If n is the root of B , then B_n is simply B .
- Otherwise we know from the preceding definition that n is in one of the two subtrees of B . If B' is that subtree, we define B_n as B'_n (the node associated with n , recursively, in the corresponding subtree).

A recursive routine on a recursive data structure

Many routines of a class that defines a data structure recursively will follow the definition’s recursive structure. A simple example is a routine computing the number of nodes in a binary tree. The node count of an empty tree is zero; the node count of a non-empty tree is one — corresponding to the root — plus (recursively) the **node counts** of the left and right subtrees, if any. We may turn this observation into a recursive function of the class `BINARY_TREE`:

```

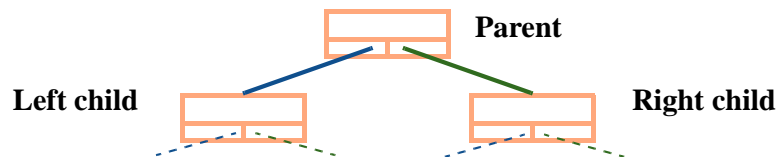
count: INTEGER
  -- Number of nodes.
do
  Result := 1
  if left /= Void then Result := Result + left.count end
  if right /= Void then Result := Result + right.count end
end

```

Note the similarity of the recursive structure to procedure *Hanoi*.

Children and parents

The **children** of a node — nodes themselves — are the root nodes of its left and right subtrees:



*A binary tree
("branching"
representation)*

If C is a child of B , then B is a **parent** of C . We may say more precisely that B is “*the*” parent of C thanks to the following result:

Theorem: Single Parent

Every node in a binary tree has exactly one parent, except for the root which has no parent.

The theorem seems obvious from the picture, but we have to prove it; this gives us an opportunity to encounter *recursive proofs*.

Recursive proofs

The recursive proof of the Single Parent theorem mirrors once more the structure of the recursive definition of binary trees.

If a binary tree BT is empty, the theorem trivially holds. Otherwise BT consists of a root and two disjoint binary trees, of which we assume — this is the “recursion hypothesis” — that they both satisfy the theorem. It follows from the definitions of “binary tree”, “child” and “parent” that a node C may have a parent P in BT only through one of the following three ways:

- P1 P is the root of BT , and C is the root of either its left or right subtree.
- P2 They both belong to the left subtree, and P is the parent of C in that subtree.
- P3 They both belong to the right subtree, and P is the parent of C in that subtree.

In case **P1**, *C* has, from the recursion hypothesis, no parent in its subtree; so it has one parent, the root, in *BT* as a whole. In cases **P2** and **P3**, again by the recursion hypothesis, *P* was the single parent of *C* in their respective subtree, and this is still the case in the whole tree.

Any node *C* other than the root falls into one of these three cases, and hence has exactly one parent. In none of these cases can *C* be the root which, as a consequence, has no parent. This completes the proof.

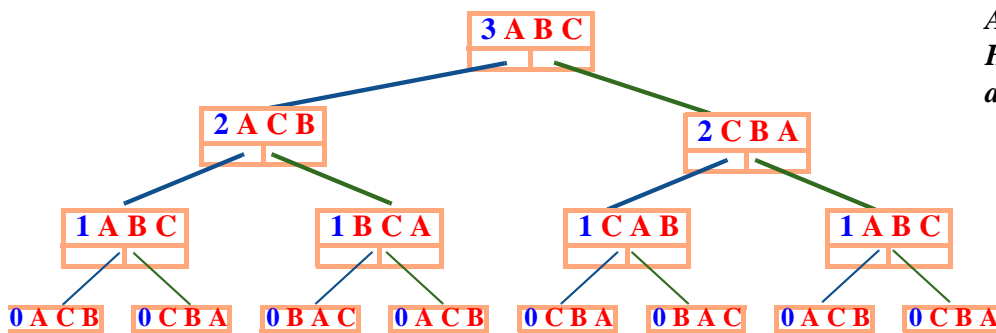
Recursive proofs of this kind are useful when you need to establish that a certain property holds for all instances of a recursively defined concept. The structure of the proof follows the structure of the definition:

- For any non-recursive case of the definition, you must prove the property directly. (In the example the non-recursive case is an empty tree.)
- A case of the definition is recursive if it defines a new instance of the concept in terms of existing instances. For those cases you may assume that the property holds of these instances (this is the *recursion hypothesis*) to prove that it holds of the new one.

This technique applies to recursively defined concepts in general. We will see its application to recursively defined routines such as *hanoi*.

A binary tree of executions

An interesting example of a binary tree is the one we obtain if we model an execution of the *hanoi* procedure, for example with three disks on needles **A**, **B**, **C**. Each node contains the arguments to the given call; the left and right subtrees correspond to the first and second recursive calls.



*An execution of
Hanoi viewed as
a binary tree*

By adding the *move* operations you may reconstruct the sequence of operations; we will see this formally below.

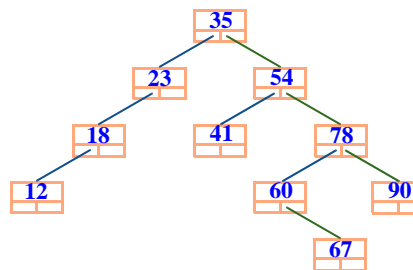
→ Page 454.

This example illustrates the connection between recursive algorithms and recursive data structures. For routines that have a variable number of recursive calls, rather than exactly two as *hanoi*, the execution would be modeled by a general tree rather than a binary tree.

More binary tree properties and terminology

As noted, a node of a binary tree may have:

- Both a left child and a right child, like the top node, labeled **35**, of our example.
- Only a left child, like all the nodes of the left subtree, labeled **23**, **18**, **12**.
- Only a right child, like the node labeled **60**.
- No child, in which case it is called a **leaf**. In the example the leaves are labeled **12**, **41**, **67** and **90**.



(From the figure on page 447.)

We define an **upward path** in a binary tree as a sequence of zero or more nodes, where any node in the sequence is the parent of the previous one if any. In our example, the nodes of labels **60**, **78**, **54** form an upward path. We have the following property, a consequence of the Single Parent theorem:

Theorem: Root Path

From any node of a binary tree, there is a single upward path to the root.

Proof: consider an arbitrary node C and the upward path starting at C and obtained by adding the parent of each node on the path, as long as there is one; the Single Parent theorem ensures that this path is uniquely defined. If the path is finite, its last element is the root, since any other node has a parent and hence would allow us to add one more element to the path; so to prove the theorem it suffices to show that all paths are finite.

The only way for a path to be infinite, since our binary trees are finite sets of nodes, would be to include a **cycle**, that is to say if a node n appeared twice (and hence an infinite number of times). This means the path includes a subsequence of the form $n \dots n$. But then n appears in its own left or right subtree, which is impossible from the definition of binary trees.

Considering downward rather than upward paths gives an immediate consequence of the preceding theorem:

Theorem: Downward Path

For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of *left* and *right* links.

The **height** of a binary tree is the maximum number of nodes on a downward path from the root to a leaf (or the reverse upward path). In the example (see figure above) the height is 5, obtained through the path from the root to the leaf labeled **67**.

It is possible to define this notion recursively, following again the recursive structure of the definition of binary trees: the height of an empty tree is zero; the height of a non-empty tree is one plus the maximum of (recursively) the heights of its two subtrees. We may add the corresponding function to class *BINARY_TREE*:

```

height: INTEGER
  -- Maximum number of nodes on a downward path.
  local
    lh, rh: INTEGER
  do
    if left /= Void then lh := left.height end
    if right /= Void then rh := right.height end
    Result := 1 + lh.max(rh)
  end

```

x.max(y) is the maximum of *x* and *y*.

This adapts the recursive definition to the convention used by the class, which only considers non-empty binary trees, although either or both subtrees, *left* and *right*, may be empty. Note again the similarity to *hanoi*.

Binary tree operations

Class *BINARY_TREE* as given so far has only three features, all of them queries: *item*, *left* and *right*. We may add a creation procedure ← Page 447.

```

make(x: G)
  -- Initialize with item value x.
  do
    item := x
  ensure
    set: item = x
  end

```

and commands for changing the subtrees and the root value:

```

add_left(x: G)
  -- Create left child of value x.
  require
    no_left_child_behind: left = Void
  do
    create left.make(x)
  end
add_right ... Same model as add_left ...
replace(x: G)
  -- Set root value to x.
  do item := x end

```

Note the precondition, which prevents overwriting an existing child. It is possible to add procedures *put_left* and *put_right*, which replace an existing child and do not have this precondition.

In practice it is convenient to specify *replace* as an assigner command for the corresponding query, by changing the declarations of this query to

```
item: G assign replace
making it possible to write bt.item := x rather than bt.put (x).
```

← “Bracket notation and assigner commands”, page 384.

Traversals

Being defined recursively, binary trees lead, not surprisingly, to many recursive routines. Function *height* was one; here is another. Assume that you are requested to print all the *item* values associated with nodes of the tree. The following procedure, to be added to the class, does the job:

```
print_all
-- Print all node values.
do
  if left /= Void then print_all (left) end
  print (item)
  if right /= Void then print_all (right) end
end
```

This uses the procedure *print* (available to all classes through their common ancestor *ANY*) which prints a suitable representation of a value of any type; here the type is *G*, the generic parameter in *BINARY_TREE [G]*.

→ “Overall inheritance structure”, 16.10, page 586.

Remarkably, the structure of *print_all* is identical to the structure of *hanoi*.

Although the business of *print_all* is to print every node item, the algorithm scheme is independent of the specific operation, here *print*, that we perform on *item*. The procedure is an example of a binary tree **traversal**: an algorithm that performs a certain operation once on every element of a data structure, in a precisely specified order. Traversal is a case of *iteration*.

For binary trees, three traversal orders are often useful:

← “Definition: Iterating”, page 397. For further study see “Agents for iteration”, 17.3, page 627.

Binary tree traversal orders

- **Inorder**: traverse left subtree, visit root, traverse right subtree.
- **Preorder**: visit root, traverse left, traverse right.
- **Postorder**: traverse left, traverse right, visit root.

In these definitions, “visit” means performing the individual node operation, such as *print* in the *print_all* example; “traverse” means a recursive application of the algorithm to a subtree, or no action if the subtree is empty.

Preorder and other traversals that always go as deep as possible into a subtree before trying other nodes are known as *depth-first*.

The procedure *print_all* is an illustration of inorder traversal. We may easily express the other two variants in the same recursive form; for example, a routine *post* for postorder traversal will have the routine body

```

if left /= Void then post (left)  end
if right /= Void then post (right) end
visit (item)

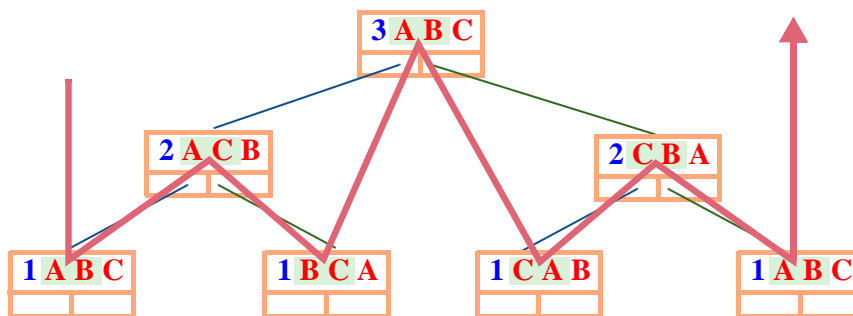
```

where *visit* is the node operation, such as *print*.

In the quest for software reuse, it is undesirable to write a different routine for variants of a given traversal scheme just because the *visit* operation changes. To avoid this, we may use the operation itself as an argument to the traversal routine.

This will be possible through the notion of **agent** in a later chapter.

As another illustration of inorder traversal, consider again the binary tree of executions of *hanoi*, for $n = 3$, with the nodes at level 0 omitted since nothing interesting happens there:



→ “Writing an iterator”, page 631.

Hanoi
execution as
inorder
traversal

(From the figure on
page 450)

 Traversal
(inorder)

Procedure *hanoi* is the mother of all inorder traversals: traverse the left subtree if any; visit the root, performing *move* (*source*, *target*), as highlighted for each node (*source* and *target* are the first two needle arguments); traverse the right subtree if any. The inorder traversal, as illustrated by the bold line, produces the required sequence of moves **A B, A C, B C, A B, C A, C B, A B**.

Binary search trees

For a general binary tree, procedure *print_all*, implementing inorder traversal, prints the node values in an arbitrary order. For the order to be significant, we must move on from binary trees to binary *search* trees.

The set G over which a general binary tree is defined can be any set. For binary search trees, we assume that G is equipped with a **total order relation** enabling us to compare two arbitrary elements of G through the boolean

→ We will learn more
about total orders in
the study of topological
sort: “Total
orders”, page 514.

expression $a < b$, such that exactly one of $a < b$, $b < a$ and $a \sim b$ (object equality) is true. Examples of such sets include *INTEGER* and *REAL*, with the usual $<$ relation, but G could be any other set on which we know a total order.

As usual we write $a \leq b$ for $(a < b)$ or $(a \sim b)$, and $a > b$ for $b < a$. Over such totally ordered sets we may define binary search trees:

Definition: binary search tree

A binary search tree over a totally ordered set G is a binary tree over G such that, for any subtree of root *item* value r :

- The item value le of any node in the left subtree satisfies $le < r$.
- The item value ri of any node in the right subtree satisfies $ri > r$.

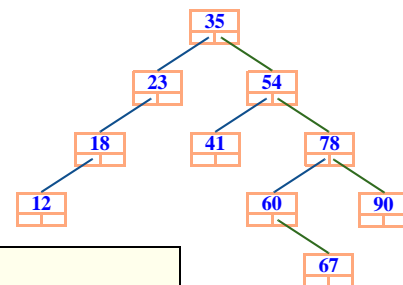
The EiffelBase class is *BINARY_SEARCH_TREE*.

The node values in the left subtree are less than the value for the root, and those in the right subtree are greater; this property must apply not only to the tree as a whole but also, recursively, to any of its immediate or indirect subtrees. We will call it the **Binary Search Tree Invariant**.

This definition implies that all the *item* values of the tree's node are different. We will use this convention for simplicity. It is also possible to accept duplications; then the conditions in the definitions become $le \leq r$ and $r \leq ri$. An exercise asks you accordingly to adapt the binary search tree algorithms that we are going to see.

Our example binary tree of integers is a binary search tree: all the values in the left subtree are less than the root value, **35**, all those in the right subtree are greater, and the same properties hold recursively in every subtree.

The procedure *print_all*, applied to a binary search tree, will print all the node items in order, from smallest to greatest.



→ Exercise 14-E.3, page 500.

Programming Time! Printing values in order

Using the procedures given so far, write a program that builds the example tree, then prints the node items using *print_all*. Check that the values are in order.

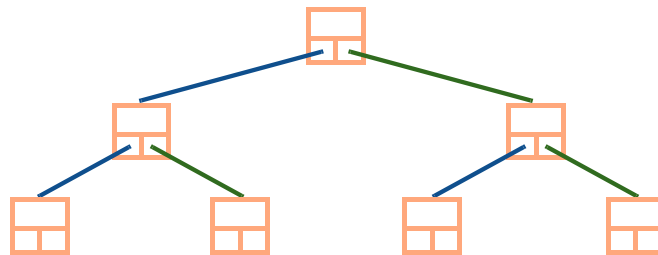
Performance

Let us look more closely at why binary search trees are useful as container structures — a potential competitor to hash tables. Indeed they usually provide much better performance than sequential lists. Assuming random data, a sequential list provides us, n being the number of items, with

- $O(1)$ insertion (if we keep the items in the order of insertion).
- $O(n)$ search.

← Second performance table on page 407.

With a binary search tree, both operations can be $O(\log n)$, much better than $O(n)$ for large n . (Remember that in big- O notation it does not matter what base we choose for the logarithms.) Here is the analysis for a **full** binary tree, that is to say one in which both subtrees of any given node have exactly the same height h :

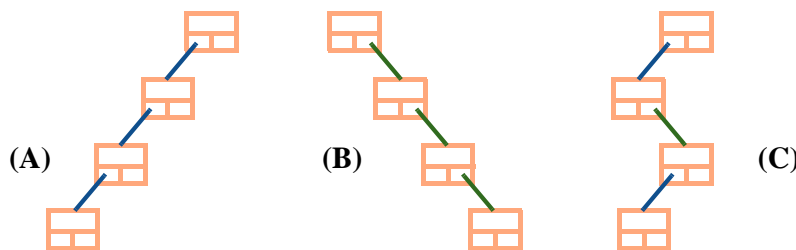


A full binary tree

It is clear, by induction on h , that the number of nodes n in a full tree of height h is $2^h - 1$ (in the above figure, h is 3 and n is 7). This implies that for a given number of nodes n the height is $\log_2(n + 1)$, which is $O(\log n)$. In a full tree, both a search and an insertion — using algorithms given below, which you can already guess — will start from the root and follow a downward path to a leaf, taking $O(\log n)$ time. This is the major attraction of binary search trees.

← “Theorem: Downward Path”, page 451.

Of course most practical binary trees are not full; if you are out of luck with the order of insertion, the performance can be as bad as with sequential lists, $O(n)$ — with added storage costs since each node has both a *left* field and a *right* field where a linked list cell has just one. The following figure shows such cases: insertions in descending order (A), ascending order (B), greatest then smallest then second greatest and so on (C).



Some binary search tree schemes causing $O(n)$ behavior

With a random enough order of insertions, however, the binary search tree will remain sufficiently close to full to ensure $O(\log n)$ behavior. You can actually *guarantee* $O(\log n)$ insertions, searches and deletions by using the **AVL** or “red-black” variants of binary search trees, which remain near-full.

On these techniques, see the bibliographic references of the previous chapter, for example Cormen et al. (page 433).

Inserting, searching, deleting

Here is a recursive routine for searching a binary search tree (this routine and the following ones are to be added to the binary search tree class):


```

has (x: G): BOOLEAN
  -- Does x appear in any node?
  require
    argument_exists: x /= Void
  do
    if x ~ item then
      Result := True
    elseif x < item then
      Result := (left /= Void) and then left.has (x)
    else -- x > item
      Result := (right /= Void) and then right.has (x)
    end
  end
end

```

~ is object equality.

The algorithm is $O(h)$ where h is the height of the tree, meaning $O(\log n)$ for full or near-full trees.

In this case there is a reasonably simple non-recursive version, using a loop:

```

has1 (x: G): BOOLEAN
  -- Does x appear in any node?
  require
    argument_exists: x /= Void
  local
    node: BINARY_TREE [G]
  do
    from
      node := Current
    until
      Result or node = Void
    invariant
      -- x does not appear above node on downward path from root
    loop
      if x < item then
        node := left
      elseif x > item then
        node := right
      else
        Result := True
      end
    end
    variant
      -- (Height of tree) – (Length of path from root to node)
    end
  end
end

```

← The variant and invariant are pseudocode; see “Touch of Style: Highlighting pseudocode”, page 109.

For *inserting* an element, we may use the following recursive procedure:

```

put (x: G)
  -- Insert x if not already present.
  require
    argument_exists: x /= Void
  do
    if x < item then
      if left = Void then
        add_left (x)
      else
        left.put (x)
      end
    elseif x > item then
      if right = Void then
        add_right (x)
      else
        right.put (x)
      end
    end
  end
end

```

← About *add_left* and *add_right* see page 452.

The absence of an **else** clause for the outermost **if** reflects the decision to ban duplicate information. As a consequence, a call to *put* with an already present value will have no effect. This is correct behavior (“*not a bug but a feature*”), since the header comment is clear. Some users might, however, prefer a different API with a precondition stating **not has** (*x*).

← See page 455.

The non-recursive version is left as an exercise.

→ 14-E.5, page 502.

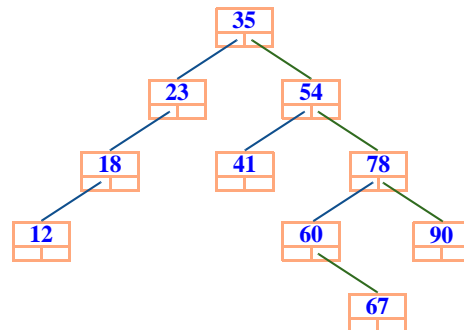
The next natural question is how to write a deletion procedure *remove* (*x*: *G*). This is less simple because we cannot just remove the node containing *x* (unless it is a leaf and not the root, in which case we make the corresponding *left* or *right* reference void); we also cannot leave an arbitrary value there since it would destroy the Binary Search Tree Invariant.

More precisely we could put a special boolean attribute in every node, indicating whether the *item* value is meaningful, but that makes things too complicated, wastes space and affects the other algorithms.

What we should do is reorganize the node values, moving up some of those found in subtrees of the node where we find *x* to reestablish the Binary Search Tree Invariant.

In the example binary search tree, a call *remove* (35), affecting the value in the root node, might either:

- Move up all the values from the left subtree (where each node has a single child, on the left).
- Move up the value in the right child, 54, then recursively apply a similar choice to move values up in one of its subtrees.



(From the figure on page 447.)

Like search and insertion, the process should be $O(h)$ where h is the height of the tree, in favorable cases.

The deletion procedure is left as an exercise; I suggest you try your hand at it now, following the inspiration of the preceding routines:

Programming Time!
Deletion in a binary search tree

Write a procedure *remove* ($x: G$) that removes from a binary search tree the node, if any, of item value x , preserving the Binary Search tree Invariant.

14.5 BACKTRACKING AND ALPHA-BETA

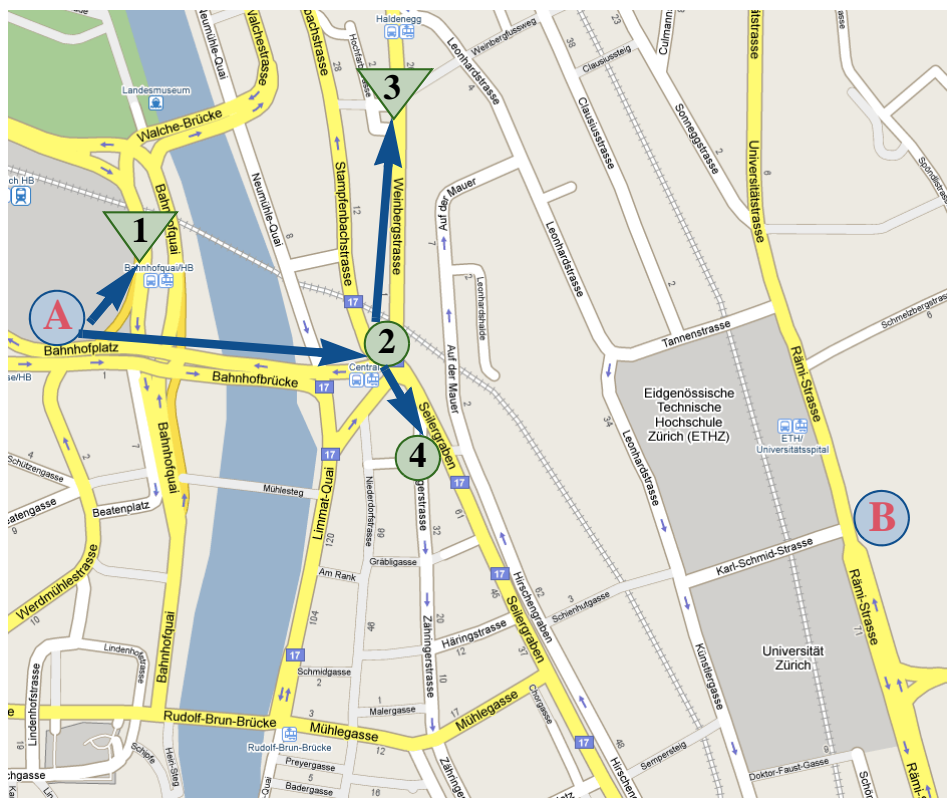
Before we explore the theoretical basis of recursive programming, it is useful to look into one more application, or rather a whole class of applications, for which recursion is the natural tool: backtracking algorithms.

The name carries the basic idea: a backtracking algorithm looks for a solution to a certain problem by trying successive paths and, whenever a path reaches a dead end, backing up to a previous path from which not all possible continuations have been tried. The process ends successfully if it finds a path that yields a solution, and otherwise fails after exhausting all possible paths, or hitting a preset termination condition such as a search time limit.

A problem may be amenable to backtracking if every potential solution can be defined as a sequence of choices.

The plight of the shy tourist

You may have applied backtracking, as a last resort, to reach a travel destination. Say you are at position **A** (Zurich main station) and want to get to **B** (between the main buildings of ETH and the University of Zurich):



Trying and backtracking

● Intermediate state

▼ Dead-end

Not having a map and too shy to ask for directions, you are reduced to trying out streets and checking, after each try, if you have arrived (you do have a photo of the destination). You know that the destination is towards the East; so, to avoid cycles, you ignore any westward street segment.

At each step you try street segments starting from the north, clockwise: the first attempt takes you to position **1**. You realize that it is not your destination; since the only possible segment from there goes west, this is a dead end: you backtrack to **A** and try the next choice from there, taking you to **2**. From there you try **3**, again a dead end as all segments go west. You backtrack to the previous position, **2**.

If all valid (non-westward) positions had been tried, **2** would be a dead-end too, and you would have to go back to **A**, but there remains an unexplored choice, leading to **4**.

The process continues like this; you can complete the itinerary on the map above. While not necessarily the best technique for traveling, it is sometimes the only possible one, and it is representative of the general trial-and-error scheme of backtrack programming. This scheme can be expressed as a recursive routine:

```

find (p: PATH): PATH
    -- Solution, if any, starting at p.
require
    meaningful: p /= Void
local
    c: LIST [CHOICE]
do
    if p.is_solution then
        Result := p
    else
        c := p.choices
        from c.start until
            (Result /= Void) or c.after
        loop
            Result := find (p + c)
            c.forth
        end
    end
end

```

This uses the following conventions: the choices at every step are described by a type *CHOICE* (in many cases you may be able to use just *INTEGER*); there is also a type *PATH*, but a path is simply a sequence of choices, and $p + c$ is the path obtained by appending c to p . We identify a solution with the path that leads to it, so *find* returns a *PATH*; by convention that result is void if *find* finds no solution. To know if a path yields a solution we have the query *is_solution*. The list of choices available from p — an empty list if p is a dead end — is $p.choices$.

To obtain the solution to a problem it suffices to use *find* ($p0$) where $p0$ is an initial, empty path.

As usual, **Result** is initialized to **Void**, so that if in a call to *find* (p) none of the recursive calls on possible extensions $p + c$ of p yields a solution — in particular, if there are no such extensions as $p.choices$ is empty — the loop will terminate with *c.after*; then *find* (p) itself will return **Void**. If this was the original call *find* ($p0$), the process terminates without producing a solution; but if not, it is a recursively triggered call, and the parent call will simply resume by trying the next remaining choice if any (or returning **Void** too if none are left).

If, on the other hand, the call finds p to be a solution, it returns p as its result, and all the callers up the chain will return it as well, terminating their list traversals through the **Result /= Void** clause of the exit condition.

Recursion is clearly essential to handle such a scheme. It is a natural way to express the trial-and-error nature of backtracking algorithms; the machinery of recursion takes care of everything. To realize its contribution, imagine for a second how you would program such a scheme without recursion, keeping track of previously visited positions. (I am not suggesting you actually write out the full non-recursive version, at least not until you have read about derecursification techniques further in this chapter.)

The later discussion also shows how to improve the efficiency of the given algorithm by removing unnecessary bookkeeping. For example it is not really necessary to pass the path p as an explicit argument, taking up space on the call stack; p can instead be an attribute, if we add $p := p + x$ before the recursive call and $p := p.head$ after it (where $head$ yields a copy of a sequence with its last element removed). We will develop a general framework allowing us to carry out such optimizations safely.

→ “Implementation of recursive routines”, 14.9, page 486.

→ “Preserving and restoring the context”, page 488.

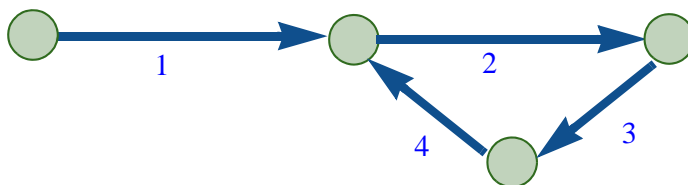
Getting backtracking right

The general backtracking scheme requires some tuning for practical use. First, as given, it is not guaranteed to terminate, as it could go on exploring ever longer paths. To ensure that any execution terminates, you should either:

- Have a guarantee (from the problem domain) that there are no infinite paths; in other words, that repeatedly extending any path will eventually yield a path with an empty *choices* list.
- Define a maximum path length and adapt the algorithm so that it treats any path as a dead-end when it reaches that limit. Instead of the path length you may also limit the computation time. Either variant is a simple change to the preceding algorithm.

→ Exercise “Backtracking curtailed”, 14-E.8, page 503.

In addition, a practical implementation can usually detect that a path is equivalent to another; for example, with the situation pictured



Path with a cycle

the paths [1 2 3 4], [1 2 3 4 2 3 4], [1 2 3 4 2 3 4 2 3 4] etc. are all equivalent. The example of finding an itinerary to a destination avoided this problem through an expedient — never go west, young man — but this is not a generalizable solution. To avoid running into such cycles, the algorithm should keep a list of previously visited positions, and ignore any path leading to such a position.

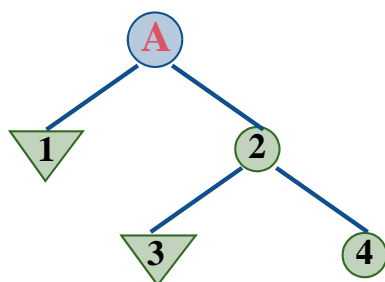
→ Exercise “Cycles despised”, 14-E.9, page 503.

Backtracking and trees

Any problem that lends itself to a backtracking solution also lends itself to modeling by a tree. In establishing this correspondence, we use trees where a node may have any number of children, generalizing the concepts defined earlier for binary trees. A path in the tree (sequence of nodes) corresponds to a path in the backtracking algorithm (sequence of choices); the tree of the itinerary example, limited to the choices that we tried, is:

← “Binary trees”, 14.4, page 447.

→ “Trying and backtracking”, page 460.



Backtrack tree

We can represent the entire town map in this way: nodes for locations, connected by edges representing street segment. The result is a *graph*. A graph only yields a tree if it has no cycles. Here this is not the case, but we can get a tree, called a *spanning tree* for the graph, containing all of its nodes and some of its edges, through one of the techniques mentioned earlier: using a cycle-avoiding convention such as never going west, or building paths from a root and excluding any edge that leads to a previously encountered node. The above tree is a spanning tree for the part of our example that includes nodes **A**, **1**, **2**, **3** and **4**.

With this tree representation of the problem:

- A solution is a node that satisfies the given criterion (the property earlier called *is_solution*, adapted to apply to nodes rather than paths).
- An execution of the algorithm is simply a **preorder** (depth-first) traversal of the tree.

← About this adaptation see “Definition: Tree associated with a node”, page 448.

In the example, our preorder traversal visited nodes **A**, **1**, **2**, **3** and **4** in this order.

This correspondence indicates that “Preorder” and “backtracking” are essentially the same idea : the rule that whenever we consider a possible path we exhaust all its possible extensions — all the subtrees of its final node — *before* we look at any of the alternative choices at the same level, represented by siblings of its node. For example if **A** in the previous figure has a third child, the traversal will not consider it before it has exhausted all the subtrees of **2**.

The only property distinguishing a backtracking algorithm from an ordinary preorder traversal is that it stops as soon as it finds a node satisfying the given criterion.

“Preorder” was defined for binary trees as root first, then left subtree, then right subtree. The left-to-right order — generalized to arbitrary trees by assuming that the children of each node are ordered — is not essential here; “depth-first” does not imply any such ordering. It is just as simple, however, to assume that the choices open to the algorithm at every stage are numbered, and tried in order.

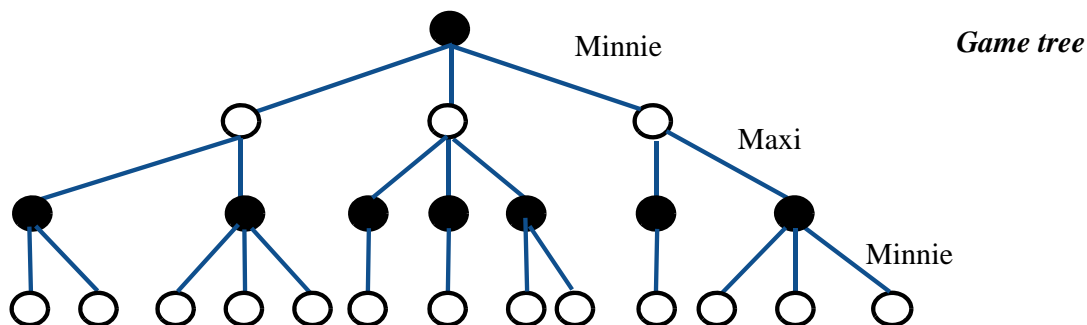
Minimax

An interesting example of the backtracking strategy, also modeled naturally as a tree, is the “minimax” technique for games such as chess. It is applicable if you can make the following assumptions about the game:

- It is a two-player game. We assume two players called Minnie and Maximilian, the latter familiarly known as Maxi.
- To evaluate the situation at any time during a game, you have an *evaluation function* with a numerical value, devised so that a lower value is better for Minnie and a higher one for Maxi.

A primitive evaluation function in checkers, assuming Maxi is Black, would be $(mb - mw) + 3 * (kb - kw)$ where mb, mw are the numbers of black and white “men” and kb, kw the corresponding numbers of “kings”; the evaluation function considers a king to be worth three times as much as a man. Good game-playing programs use far more sophisticated functions.

Minnie looks for a sequence of moves leading to a position that minimizes the evaluation function, and Maxi for one that maximizes it.



Each player uses the minimax strategy to choose, from a game position, one of the legal moves. The tree model represents possible games; successive levels of the tree alternatively represent the moves of each player.

In the figure, we start from a position where it is Minnie's turn to play. The goal of the strategy is to let Minnie choose, among the moves available from the current position (three in the figure), the one that guarantees the best outcome — meaning, in her case, the minimal guaranteed evaluation function value in leaves of the tree. The method is symmetric, so Maxi would rely on the same mechanism, maximizing instead of minimizing.

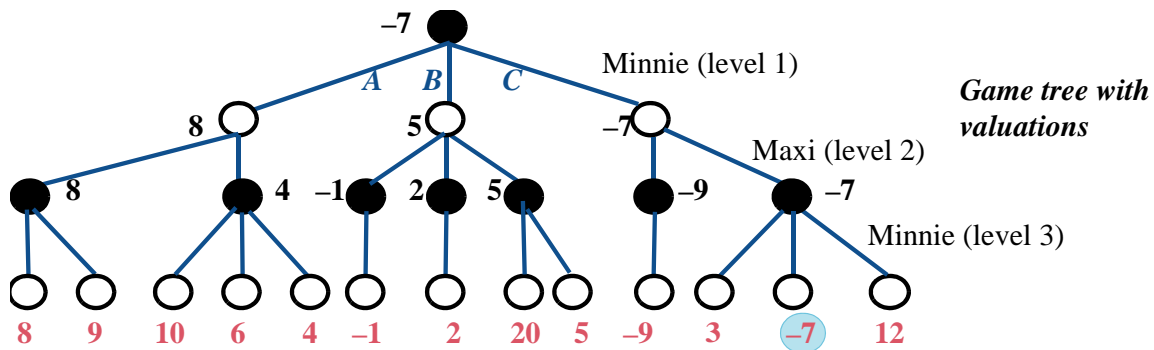
This assumption of symmetry is essential to the minimax strategy, which performs a depth-first traversal of the tree of moves to assign a value to every node:

M1 The value of a leaf is the result of applying the evaluation function to the corresponding game position.

M2 The value of an internal node from which the moves are Maxi's is the *maximum* of the values of the node's children.

M3 In Minnie's case it is the *minimum* of the children's values.

The value of the game as a whole is the value associated with the root node. To obtain a strategy we must retain for each internal node, in cases M2 and M3, not only the value but also the child choice that leads to this value. Here is an illustration of the strategy obtained by assuming some values for the evaluation function (shown in color) in the leaves of our example tree:



You can see that the value at each node is the minimum (at levels 1 and 3) or maximum (at level 2) of the values of the children. The desirable move for Minnie, guaranteeing the minimum value -7 , is choice *C*.

Backtracking is appropriate for minimax since the strategy must obtain the values for every node's children before it can determine the value for the node itself, requiring a depth-first traversal.

The following algorithm, a variation on the earlier general backtracking scheme, implements these ideas. It is expressed as a function *minimax* returning a pair of integers: guaranteed value from a starting position *p*, initial choice leading to that value. The second argument *l* is the level at which position *p* appears in the overall game tree; the first move from that position, returned as part of the result, is Minnie's move as in the figures if *l* is odd, and Maxi's if *l* is even.

```

minimax (p: POSITION; l: INTEGER): TUPLE [value, choice: INTEGER]
  -- Optimal strategy (value + choice) at level l starting from p.
  local
    next: TUPLE [value, choice: INTEGER]
  do
    if p.is_terminal (l) then
      Result := [value: p.value; choice: 0]
    else
      c := p.choices
      from
        Result := worst (l)
          c.start
      until c.after loop
        next := minimax (p.moved (c.item), l + 1)
        Result := better (next, Result, l)
      end
    end
  end
end

```

To represent the result, we use a tuple of integers representing the value and the choice.

The auxiliary functions *worst* and *better* are there to switch between Minnie's and Maxi's viewpoints: the player is minimizing for any odd level l and maximizing for any even l .

```
worst (l: INTEGER): INTEGER
```

```
-- Worst possible value for the player at level l.
```

```
do
```

```
  if  $l \parallel 2 = 1$  then Result := Max else Result := Min end
```

```
end
```

\parallel is integer remainder.

```
better (a, b: TUPLE [value, choice: INTEGER]; l: INTEGER):
```

```
    TUPLE [value, choice: INTEGER]
```

```
-- The better of a and b, according to their value, for player at level l.
```

```
do
```

```
  if  $l \parallel 2 = 1$  then
```

```
    Result := (a.value < b.value)
```

```
  else
```

```
    Result := (a.value > b.value)
```

```
  end
```

```
end
```

To avoid the repeated use of the *TUPLE* type, you may instead define a small class *GAME_RESULT* with integer attributes *value* and *choice*.

To determine the *worst* possible value for either player we assume constants *Max*, with a very large value, and *Min*, with a very small value, for example the largest and smallest representable integers.

Function *minimax* assumes the following features from class *POSITION*:

- *is_terminal* indicates that no moves should be explored from a position.
- In that case *value* gives the value of the evaluation function. (The query *value* may have the precondition *is_terminal*.)
- For a non-terminal position *choices* yields the list of choices, each represented by an integer, leading to a legal moves.
- If *i* is such a choice, *moved* (*i*) gives the position resulting from applying the corresponding move to the current position.

The simplest way to ensure that the algorithm terminates is to limit the depth of the exploration to a set number of levels *Limit*. This is why *is_terminal* as given includes the level *l* as argument; it can then be written as just

```

is_terminal (l: INTEGER): BOOLEAN
    -- Should exploration, at level l, stop at current position?
    do
        Result := (l = Limit) or choices.is_empty
    end
  
```

In practice a more sophisticated cutoff criterion is appropriate; for example the algorithm could keep track of CPU time and stop exploration from a given position when the exploration time reaches a preset maximum.

To run the strategy we call *minimax* (*initial*, 1) where *initial* is the initial game position. Level 1, odd, indicates that the first move is Minnie's.

Alpha-beta

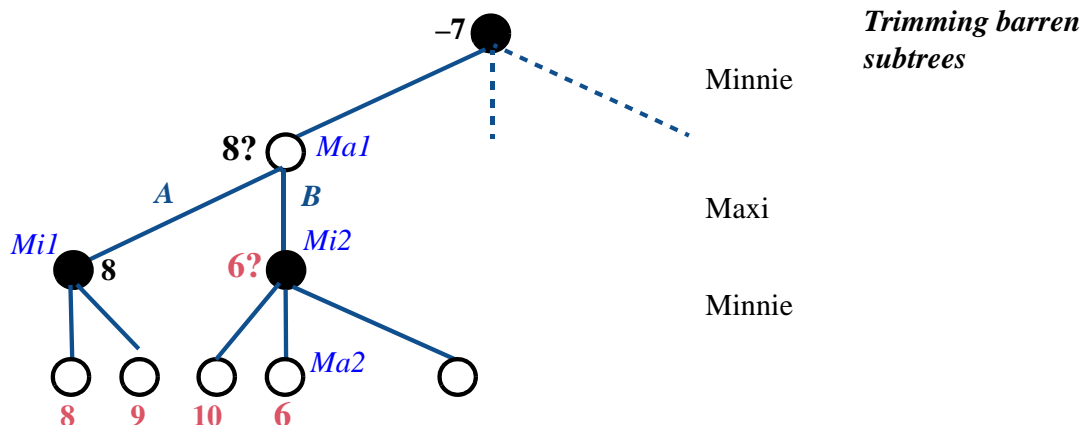
The minimax strategy as seen so far always performs a full backtracking traversal of the tree of relevant moves. An optimization known as alpha-beta pruning can significantly improve its efficiency by skipping the exploration of entire subtrees. It is a clever idea, worth taking a look at not just because it is clever but also as an example of refining a recursive algorithm.

Alpha-beta is only meaningful if, as has been our assumption for minimax, the game strategy for each of the two players assumes that the *other* player's strategy is reversed (one minimizes, the other maximizes) but otherwise identical.

The insight that can trim entire subtrees in the exploration is that it is not necessary for a player at level *l* + 1 to continue exploring a subtree if it finds that this could only deliver a result better for the player itself, and hence worse for its adversary, than what the adversary has already guaranteed at level *l*: the adversary, which uses the reversed version of the strategy, would never select that subtree.

This discussion refers to a player as "it" since our players are program elements.

The previous example provides an illustration. Consider the situation after the minimax algorithm has explored some of the initial nodes:



We are in the process of computing the value (a maximum) for node *Mal*, and as part of this goal the value (a minimum) for node *Mi2*. From exploring the first subtree of *Mal*, rooted at *Mi1*, we already have a tentative maximum value for *Mal*: **8**, signaled by a question mark since it is only temporary. This means a guarantee for Maxi that he will not do, at *Mal*, worse than **8**. For Maxi, “worse” means lower. In exploring the *Mi2* subtree we come to *Ma2*, where the value — obtained in this case from the evaluation function since *Ma2* is a leaf, but the reasoning would apply to any node — is **6**. So at node *Mi2* Minnie will not do worse (meaning, in her case, higher) than 6. But then Maxi would never, from node *Ma2*, take choice *B* leading to *Mi2*, since he already has a better result from choice *A*. Continuing to explore the subtree rooted at *Mi2*, part of choice *B*, would just be a waste of time. So as soon as it has found value **6** at *Ma2* the alpha-beta strategy discards the rest of the *Mi2* subtree.

In the figure’s example there is only one node left in the *Mi2* subtree after *Ma2* and we are at the leaf level, but of course *Ma2* could have many more right siblings with large subtrees.

Not only is this optimization an interesting insight; it also provides a good opportunity to hone our recursive programming skills. Indeed do not wait for the solution (that is to say, refrain from turning the page just now!) and try first to devise it by yourself:

Programming time!
Adding Alpha-beta to Minimax

Adapt the minimax algorithm, as given earlier, so that it will use the alpha-beta strategy to avoid exploring useless subtrees.

← Function *minimax*,
page 466.

The extension is simple. (Well, as you will have noted if you did try, it requires some care to get the details right, in particular to avoid getting our *better* comparisons upside down.) The routine needs one more argument to denote the value, if any, already guaranteed for the adversary at the level immediately above. Here is minimax updated for Alpha-beta, additions highlighted:

```

alpha_beta (p: POSITION; l: INTEGER; guarantee: INTEGER):
    TUPLE [value, choice: INTEGER]
    -- Optimal strategy (value + choice) at level l, starting from p.
    -- Even level minimizes, odd level maximizes.
    local
        next: TUPLE [value, choice: INTEGER]
    do
        if p.is_terminal (l) then
            Result := [value: p.value; choice: 0]
        else
            c := p.choices
            from
                Result := worst (l)
                c.start
            until c.after or better (guarantee, Result, l - 1) loop
                next := minimax (p.moved (c.item), l + 1), Result )
                Result := better (next, Result, l)
            end
        end
    end
end

```

← The parts not highlighted are unchanged from *minimax*, page 466 (departing from the convention of the rest of this chapter, which highlights recursive branches).

Each player now stops exploring **its** alternatives whenever it finds a result that is “*better*” for the adversary than the “*guarantee*” the adversary may already have assured.

Since *better* was defined without a precondition it will accept a zero level, so it is acceptable to pass it $l - 1$. We might equivalently pass $l + 1$. In fact a slightly simpler variant of *better (guarantee, Result, $l - 1$)* is *better (Result, guarantee, l)*; it is equivalent thanks to the symmetric nature of the strategy.

The recursive call passes as a “*guarantee*” to the next level the best **Result** obtained so far for the current level. As a consequence, alpha-beta’s trimming, which stops the traversal of a node’s children when it hits the new exit trigger *better (guarantee, Result, $l - 1$)*, will never occur when the node itself is the *first child* of its own parent; this is because the loop initializes **Result** to the *worst* value for the player, so the initial *guarantee* is useless. Only when the traversal moves on to subsequent children does it get a chance to trigger the optimization.

Minimax and alpha-beta provide a representative picture of backtracking algorithms, which have widespread applications to problems defined by large search spaces. The key to successful backtracking strategies is often — as illustrated by alpha-beta — to find insights that avoid exhaustive search.

14.6 FROM LOOPS TO RECURSION

Back to the general machinery of recursion.

We have seen that some recursive algorithms — Fibonacci numbers, search and insertion for binary search trees — have a loop equivalent. What about the other way around?

It is indeed not hard to replace *any* loop by a recursive routine. Consider an arbitrary loop, given here without its invariant and variant (although we will see their recursive counterparts later):

```
from Init until Exit loop Body end
```

We may replace it by

```
Init  
loop_equiv
```

with the procedure

```
loop_equiv  
  --Emulate a loop of exit condition Exit and body Body.  
  do  
    if not Exit then  
      Body  
      Loop_equiv  
    end  
  end
```

In **functional languages** (such as Lisp, Scheme, Haskell, ML), the recursive form is the preferred style, even if loops are available. We could use it too in our framework, replacing for example the first complete example of the discussion of loops, which animated a Metro line by moving a red dot, with

```
Line8.start  
animate_rest (Line8)
```

← “*Functional programming and functional languages*”, page 324.

← Page 168.

relying on the auxiliary routine

```

animate_rest (line: LINE)
  -- Animate stations of line from current cursor position on
  do
    if not line.after then
      show_spot (line.item.location)
      line.forth
      animate_rest (line)
    end
  end
end

```

(A more complete version should restore the cursor to its original position.)

The recursive version is elegant, but there is no particular reason in our framework to prefer it to the loop form; indeed we will continue to use loops.

The conclusion might be different if we were using functional programming languages, where systematic reliance on recursive routines is part of a distinctive style of programming.

Even if just for theoretical purposes, it is interesting to know that loops are conceptually not needed if we have routines that can be recursive. As an example, recursion gives us a more concise version of the loop-based routine *paradox* demonstrating the unsolvability of the Halting Problem:

```

recursive_paradox
  -- Terminate if and only if not.
  do
    if terminates ("C:\your_project") then
      recursive_paradox
    end
  end
end

```

← “An application: proving the undecidability of the halting problem”, page 223.

Knowing that we can easily emulate loops with recursion, it is natural to ask about the reverse transformation. Do we really need recursive routines, or could we use loops instead?

We have seen straightforward cases: *Fibonacci* as well as *has* and *put* for binary search trees. Others such as *hanoi*, *height*, *print_all* do not have an immediately obvious recursion-free equivalent. To understand what exactly can be done we must first look more closely into the meaning and properties of recursive routines.

14.7 MAKING SENSE OF RECURSION

The experience of our first few recursive schemes allows us to probe a bit deeper into the meaning of recursive definitions.

Vicious circle?

First we go back to the impolite but inevitable question: does the recursive emperor have any clothes? That is to say, does a recursive definition mean anything at all? The examples, especially those of recursive routines, should by now be sufficiently convincing to suggest a positive answer, but we should still retain a healthy dose of doubt. After all we keep venturing dangerously close to definitions that make no sense at all — vicious circles. With recursion we try to define a concept in terms of itself, but we cannot just define it *as* itself. If I say

“Computer science is the study of computer science”

I have not defined anything at all, just stated a tautology; not one of those tautologies of logic, which are things to prove and hence possibly interesting, just a platitude. If I refine this into

← “*Definition: Tautology*”, page 78.

“Computer science is the study of programming, data structures, algorithms, applications, theories and other areas of computer science”

I have added some usable elements but still not produced a satisfactory definition. Recursive routines can, similarly, be obviously useless, as:

```
p (x: INTEGER)
  -- What good is this?
  do p (x) end
```

which for any value of the argument would execute forever, never producing any result.

“Forever” in this case means, for a typical compiler’s implementation of recursion on an actual computer, “until the stack overflows and causes the program to crash”. So in practice, given the speed of computers, “forever” does not last long. — you can try the example for yourself.

→ You can see an example of the result on page 665.

How do we avoid such obvious misuses of recursion? If we attempt to understand why the recursive definitions seen so far seem intuitively to make sense, we can nail down three interesting properties:

Touch of Methodology:
Well-formed recursive definition

A useful recursive definition should ensure that:

R1 There is at least one non-recursive branch.

R2 Every recursive branch occurs in a context that differs from the original.

R3 For every recursive branch, the change of context (R2) brings it closer to at least one of the non-recursive cases (R1).

For a recursive routine, the change of “context” (R2) may be that the call uses a different argument, as will a call $r(n-1)$ in a routine $r(n: \text{INTEGER})$; that it applies to a different target, as in a call $x.r(n)$ where x is not the current object; or that it occurs after the routine has changed at least one field of at least one object.

The recursive routines seen so far satisfy these requirements:

- The body of *Hanoi* (n, \dots) is of the form **if** $n > 0$ **then** ... **end** where the recursive calls are in the **then** part, but there is no **else** part, so the routine does nothing for $n = 0$ (R1). The recursive calls are of the form *Hanoi* ($n-1, \dots$), changing the first argument and also switching the order of the others (R2). Replacing n by $n-1$ brings the context closer to the non-recursive case $n = 0$ (R3). ← Page 443.
- The recursive *has* for binary search trees has non-recursive cases for $x = \text{item}$, as well as for $x < \text{item}$ if there is no left subtree, and $x > \text{item}$ if there is no right subtree (R1). It calls itself recursively on a different target, *left* or *right* rather than the current object (R2); every such call goes to the left or right subtree, closer to the leaves, where the recursion terminates (R3). The same scheme governs other recursive routines on binary trees, such as *height*. ← Page 452.
- The recursive version of the metro line traversal, *animate_rest*, has a non-recursive branch (R1), doing nothing, for a cursor that is *after*. The recursive call does not change the argument, but it is preceded by a call *line.forth* which changes the state of the *line* list (R2), moving the cursor closer to a state satisfying *after* and hence to the non-recursive case (R3). ← Page 472.

R1, **R2** and **R3** also hold for recursive definitions of concepts other than routines:

- The mini-grammar for **Instruction** has the non-recursive case **Assignment**. ← Page 437.
- All our recursively defined data structures, such as **STOP**, are recursive through *references* (never through expanded values), and references can be void; in linked structures, void values serve as terminators. ← Page 437.

In the case of recursive routines, combining the above three rules suggests a notion of **variant** similar to the loop variants through which we guarantee that loops terminate:

← “Loop termination and the halting problem”, page 161.

Touch of Methodology: **Recursion Variant**

Every recursive routine should be declared with an associated recursion variant, an integer quantity associated with any call, such that:

- The routine’s precondition implies that the variant is non-negative.
- If an execution of the routine starts with a value v for the variant, the value v' of the variant for any recursive call satisfies $0 \leq v' < v$.

The variant may involve the arguments of the routine, as well as other parts of its environment such as attributes of the current object or of other objects. In the examples just reviewed:

- For *Hanoi* (n, \dots), the variant is n .
- For *has*, *height*, *print_all* and other recursive traversals of binary trees, the variant is *node_height*, the longest length of a path from the current node to a leaf.
- For *animate_rest*, the variant is, as for the corresponding loop, *Line8.count* – *Line8.index* + 1. ← Page 168.

There is no special syntax for recursion variants, but we will use a comment of the following form, here for *hanoi*:

```
-- variant n
```

Boutique cases of recursion

The well-formedness rules seem so reasonable that we might think they are necessary, not just sufficient, to make a recursive definition meaningful. Such is indeed the case with the first two properties:

- **R1**: if all branches of a definition are recursive, it cannot ever yield any instance we do not already know. In the case of a recursive routine, execution will not terminate, except in practice through a crash following memory exhaustion.
- **R2**: if a recursive branch applies to the original context, it cannot ever yield an instance we do not already know. For a recursive routine — say $p(x: T)$ with a branch that calls $p(x)$ for the same x with nothing else changed — this means that the branch, if taken, would lead to non-termination. For other recursive definitions, it means the branch is useless.

The story is different for **R3**, if we take this rule as requiring a clearly visible recursion variant such as the argument n for *Hanoi*. Some recursive routines which do terminate violate this property. Here are two examples. They have no practical application, but highlight general properties of which you must be aware.

McCarthy's 91 function was devised by John McCarthy, a professor at Stanford University, designer of the Lisp programming language (where recursion plays a prominent role) and one of the creators of Artificial Intelligence. We may write it as follows:

← See “*Functional programming and functional languages*”, page 324 (with photograph of McCarthy).

```

mc_carthy (n: INTEGER): INTEGER
  -- McCarthy's 91 function.
  do
    if n > 100 then
      Result := n - 10
    else
      Result := mc_carthy (mc_carthy (n + 11))
    end
  end
end

```

The value for $n > 100$ is clearly $n - 10$, but it is far less obvious — from a computation shrouded in two nested recursive calls — that for any integer up to 99, including negative values, the result will be 91, explaining the function's name. The computation indeed terminates on every possible integer value. Yet it has no obvious variant; $mc_carthy(mc_carthy(n + 11))$ actually uses as argument of the innermost recursive call a *higher* value than the original!

Here is another example, also a mathematical oddity:

```

bizarre (n: INTEGER): INTEGER
  -- A function that can yield only a 1.
  require
    positive: n >= 1
  do
    if n = 1 then
      Result := 1
    elseif even (n) then
      Result := bizarre (n // 2)
    else -- i.e. for n odd and n > 1
      Result := bizarre ((3 * n + 1) // 2)
    end
  end
end

```

This uses the operator `//` for rounded down integer division (`5 // 2` and `4 // 2` are both `2`), and a boolean expression `even (n)` to denote whether `n` is an even integer; `even (n)` can also be expressed as `n \ 2 = 0`, using the integer remainder operator `\`. The two occurrences of a `//` division in the algorithm apply to even numbers, so they are exact. *n* / 2, using the other division operator /, would give a REAL result; for example 5 / 2 is 2.5.

Clearly, if this function gives any result at all, that result can only be `1`, the value produced by the sole non-recursive branch. Less clear is whether it will give this result — that is to say, terminate — for *any* possible argument. The answer seems to be yes; if you write the program, and try it on sample values, including large ones, you will be surprised to see how fast it converges. Yet there is no obvious recursion variant; here too the change seems to go in the wrong direction: the new argument in the second recursive branch, `(3 * n + 1) // 2`, is actually larger than `n`, the previous value.

These are boutique examples, but we must take their existence into account in any general understanding of recursion. They mean that some recursive definitions exist that do *not* satisfy the seemingly reasonable methodological rules discussed above — and still yield well-defined results.

Note that such examples, if they terminate for every possible argument, do have a variant: since for any execution of the routine the number of remaining recursive calls is entirely determined by the program's state at the time of the call; it is a function of the state, and can serve as a variant. Rather, it *could* serve as a variant if we knew how to express it. If we don't, its theoretical existence does not help us much.

You will have noted that it is not possible to determine automatically — through compilers or any other program analysis tools — whether a routine has a recursive variant, even less to derive such a variant automatically: that would mean that we can solve the Halting Problem.

← “An application: proving the undecidability of the halting problem”, page 223.

In practice we dismiss such examples and limit ourselves to recursive definitions that possess properties **R1**, **R2** and **R3**, guaranteeing that they are safe. In particular, whenever you write a recursive routine, you must always — as in the examples of the rest of this chapter — explicitly list a recursive variant.

Keeping definitions non-creative

Even with well-formedness rules and recursion variants, we are not yet off the hook in our attempts to use recursion and still sleep at night. The problem is that a recursive “definition” is not a definition in the usual sense because it can be **creative**.

An *axiom* in mathematics is creative: it tells us something that we cannot deduce without it, for example (in the standard axioms for integers) that $n < n'$ holds for any integer n , where n' is the next integer. The basic *laws* of natural sciences are also creative, for example the rule that nothing can travel faster than the speed of light.

Theorems in mathematics, and specific results in physics, are not creative: they state properties that can be deduced from the axioms or laws. They are interesting on their own, and may start us on the path to new theorems; but they do not add any assumptions, only consequences of previous assumptions.

A definition too should be non-creative. It gives a new name for an object of our world, but all statements we can express with the definition could be expressed without it. We do not *want* to express them without it — otherwise we would not introduce the definition — but we trust that in principle we could. If I say

Define x^2 , for any x , as $x * x$

I have not added anything to mathematics; I am just allowing myself to use the new notation e^2 , for any expression e , in lieu of the multiplication. Any property that can be proved using the new form could also be proved — if more clumsily — using the form that serves to define it.

The symbol \triangleq , which we have taken to mean “is defined as” (starting with BNF productions), assumes this principle of non-creativity of definitions. But now consider a recursive definition, of the form

← From page 298 on.

$f \triangleq \text{some_expression}$

[1]

where *some_expression* involves *f*. It does not satisfy the principle any more! If it did we could replace any occurrence of *f* by *some_expression*; this involves *f* itself, so we would have to do it again, and so on ad infinitum. We have not really defined anything.

Until we have solved this issue — by finding a convincing, non-creative meaning for “definitions” such as [1] — we must be careful in our terminology. We will reserve the \triangleq symbol for non-recursive definitions; a property such as [1] will be expressed as an equality

$$f = \text{some_expression} \quad [2]$$

which simply states a property of the left and right sides. (We may also view it as an **equation**, of which *f* must be a solution.) To be safe when talking about recursive “definitions”, we will quarantine the second word in quotes.

The quarantine ends on page 482.

The bottom-up view of recursive definitions

To sanitize recursion and bring it out of the quarantined area, it is useful to take a **bottom-up** view of recursive routines and, more generally, recursive “definitions”. I hope this will remove any feeling of dizziness that you may still experience when seeing concepts or routines defined — apparently — in terms of themselves.

In a recursive “definition”, the recursive branches are written in a *top-down* way, defining the meaning of a concept in terms of the meaning of the same concept for a “smaller” context — smaller in the sense of the variant. For example, *Fibonacci* for *n* is expressed in terms of *Fibonacci* for *n – 1* and *n – 2*; the moves of *Hanoi* for *n* are expressed in terms of those for *n – 1*; and the syntax for *Instruction* involves a **Conditional** that contains a smaller *Instruction*.

The bottom-up view is a different interpretation of the same definition, treating it the other way around: as a mechanism that, from *known* values, gives new ones. Here is how it works, first on the example of a function. For any function *f* we may build the **graph of the function**: the set of pairs $[x, f(x)]$ for every applicable *x*. The graph of the Fibonacci function is the set

$$F \triangleq \{[0, 0], [1, 1], [2, 1], [3, 2], [4, 3], [5, 5], [6, 8], [7, 13] \dots\} \quad [3]$$

consisting of all pairs $[n, \textit{Fibonacci}(n)]$ for all non-negative integers *n*. This graph contains all information about the function. You may prefer to think of it in the following visual representation:

INTEGER	0	1	2	3	4	5	6	7	...	<i>A function graph (for the Fibonacci function)</i>
	↓	↓	↓	↓	↓	↓	↓	↓		
INTEGER	0	1	1	2	3	5	8	13	...	

The top row lists possible arguments to the function; for each of them, the bottom row gives the corresponding *fibonacci* number.

To give the function a recursive “definition” is to say that its graph F — as a set of pairs — satisfies a certain property

$$F = h(F) \quad [4]$$

for a certain function h applicable to such sets of pairs. This is like an equation that F must satisfy, and is known as a **fixpoint equation**. A fixpoint equation expresses that a certain mathematical object, here a function, remains invariant under a certain transformation, here h .

For example to “define” the Fibonacci function recursively as

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(i) &= fib(i-1) + fib(i-2) \text{-- For } i > 1 \end{aligned}$$

is to state that its graph F — the above set of pairs [3] — satisfies the fixpoint equation $F = h(F)$ [4] where h is the function that, given such a set of pairs, yields a new one containing the following pairs:

G1 Every pair already in F .

G2 $[0, 0]$. -- The pair for $n = 0$: $[0, fib(0)]$

G3 $[0, 1]$. -- The pair for $n = 0$: $[1, fib(1)]$

G4 Every pair of the form $[i, a + b]$ for some i such that F contains both a pair of the form $[i - 1, a]$ and another of the form $[i - 2, b]$.

We can use this view to give any recursive “definition” a clear meaning, free of any recursive mystery. We start from the function graph F_0 that is empty (it contains no pair). Next we define

$$F_1 \triangleq h(F_0)$$

meaning, since **G1** and **G4** are not applicable in this case (as F_0 has no pair), that F_1 is simply $\{[0, 0], [1, 1]\}$, with the two pairs given by **G2** and **G3**. Next we apply h once more to get

$$F_2 \triangleq h(F_1)$$

Here and in subsequent steps **G2** and **G3** give us nothing new, since the pairs $[0, 0]$ and $[1, 1]$ are already in F_1 , but **G4**, applied to these two pairs from F_1 , adds to F_2 the pair $[2, 1]$. Continuing like this, we define a sequence of graphs: F_0 is empty, and each F_{i+1} for $i > 0$ is defined as $h(F_i)$. Now consider the infinite union F of all the F_i for every natural integer i : $F_0 \cup F_1 \cup F_2 \cup \dots$, more concisely written

$$\bigcup_{i \in \mathbf{N}} F_i$$

where \mathbf{N} is the set of natural integers. It is easy to see that this F satisfies the property $F = h(F)$ [4].

This is the non-recursive interpretation — the semantics — we give to the recursive “definition” of Fibonacci.

In the general case, a fixpoint equation of the form [4] on function graphs, stating that F must be equal to $h(F)$, admits as a solution the function graph

$$F \triangleq \bigcup_{i \in \mathbf{N}} F_i$$

where F_i is a sequence of function graphs defined as above:

$$F_0 \triangleq \{ \} \quad \text{-- Empty set of pairs}$$

$$F_i \triangleq h(F_{i-1}) \quad \text{-- For } i > 0$$

The empty set can, of course, be written also as \emptyset . The notation $\{ \}$ emphasizes that it is a set of pairs.

This fixpoint approach is the basis of the bottom-up interpretation of recursive computations. It removes the apparent mystery from these definitions because it no longer involves defining anything “in terms of itself”: it simply views a recursive “definition” as a fixpoint equation, and admits a solution obtained as the union (similar to the limit of a sequence in mathematical analysis) of a sequence of function graphs.

This immediately justifies the requirement that any useful recursive “definition” must have a non-recursive branch: if not, the sequence, which starts with the empty set of pairs $F_0 = \{ \}$, never gets any more pairs, because all the cases in the definition of h are like **G1** and **G4** for Fibonacci, giving new pairs deduced from existing ones, but there are no pairs to begin with.

← *R1*, page 474.

This technique reduces recursive “definitions”, with all the doubts they raise as to whether they define anything at all, to the well-known, traditional notion of defining a sequence by induction.

The Fibonacci function is a good example for understanding the concepts, but perhaps not sufficient to get really excited: after all, its usual definition in mathematics textbooks already involves induction; only computer scientists look at the function in a recursive way. What we saw is that we can treat its recursive “definition” as an inductive definition — a good old definition, without the quotes — of the function’s graph. We did not learn anything about the function itself, other than a new viewpoint. Let us see whether the bottom-up view can teach us something about a few of our other examples.

This is the end of the “quarantine” decreed on page 479.

Bottom-up interpretation of a construct definition

Understood in a bottom-up spirit, the recursive definition of “type” has a clear meaning. As you will remember, it said that a **type** is either:

T1 A non-generic class, such as *INTEGER* or *STATION*.

T2 A generic derivation, of the form $C [T]$, where C is a generic class and T a **type**.

← “Definitions: Class type, generically derived, base class”, page 370.

T1 is the non-recursive case. The bottom-up perspective enables us to understand the definition as building the set of types as a succession of layers. Limiting for simplicity the number of possible generic parameters to one:

- Layer L_0 has all the types defined by non-generic classes: *INTEGER*, *STATION* and so on.
- Layer L_1 has all the types of the form $C [X]$, where C is a generic class and X is at level L_0 : *LIST [INTEGER]*, *ARRAY [STATION]* etc.
- More generally, layer L_n for any $n > 0$, has all the types of the form $C [X]$, where X is at level L_i for $i < n$.

This way we get all possible types, generically derived or not.

The towers, bottom-up

Now consider the Tower of Hanoi solution from a bottom-up perspective. We may understand the routine as recursively defining a sequence of moves. Let's denote such a sequence — move a disk from the top of needle **A** to **B**, then one from **C** to **A** and so on — as $\langle A \rightarrow B, C \rightarrow A, \dots \rangle$. The empty sequence of moves will be $\langle \rangle$ and the concatenation of sequences will use a simple “+”, so that $\langle A \rightarrow B, C \rightarrow A \rangle + \langle B \rightarrow A \rangle$ is $\langle A \rightarrow B, C \rightarrow A, B \rightarrow A \rangle$.

Then we may express the recursive solution to the Towers of Hanoi problem as a function *han* with four arguments (an integer and three needles), yielding sequences of moves, and satisfying the fixpoint equation

<i>han</i> (<i>n</i> , <i>s</i> , <i>t</i> , <i>o</i>) =	
$\langle \rangle$	-- If $n = 0$ [5]
$\textit{han}(n - 1, s, o, t) + \langle s \rightarrow t \rangle + \textit{han}(n - 1, o, t, s)$	-- If $n > 0$ [6]

defined only when the values of *s*, *t*, *o* (short for *source*, *target*, *other*) are different — we take them as before to range over 'A', 'B', 'C' — and *n* is positive.

The bottom-up construction of the function that solves this equation is simple. **[5]** lets us initialize the function's graph to all pairs for $n = 0$, each of the form

$[(0, s, t, o), \langle \rangle]$

for *s*, *t*, *o* ranging over all permutations of 'A', 'B', 'C'. Let us call H_0 this first part of the graph, made of six pairs.

Now we may use **[6]** to obtain the next part H_1 , containing all the values for $n = 1$; they are all of the form

$[(1, s, t, o), \langle s \rightarrow t \rangle]$

since for any sequence *x* the concatenation $\langle \rangle + x$ or $x + \langle \rangle$ is *x* itself. The next iteration of **[6]** gives us H_2 , whose pairs are of the form

$[(2, s, t, o), fI + \langle s \rightarrow t \rangle + gI]$

for all *s*, *t*, *o* such that H_1 contains both a pair of the form $[(1, s, o, t), fI]$ and one of the form $[(1, o, t, s), gI]$.

Iterating again will give us H_3 and subsequent elements of the graph. The complete graph — infinite of course, since it includes pairs for all possible values of n — is the set of all pairs in all elements of the sequence, $\bigcup_{i \in \mathbb{N}} H_i$.

Here I strongly suggest that you get a concrete grasp of the bottom-up view of recursive computation by writing a program that actually builds the graph:

Programming time:
Producing the graph of a function

Write a program (not using recursion) that produces successive elements $H_0, H_1, H_2 \dots$ of the function graph for the recursive Hanoi solution.

→ Details in exercise 14-E.11, page 503.

A related exercise asks you to determine (without programming) the mathematical properties of the graph. → 14-E.10, page 503.

Another important exercise directs you to apply a similar analysis to binary tree traversals. You will have to devise a model for representing the solution, similar to the one we have used here; instead of sequences of moves you will simply use sequences of nodes. → 14-E.12, page 503.

Grammars as recursively defined functions

The bottom-up view is particularly intuitive for a recursive grammar, as in our small example:

Instruction \triangleq **ast** | Conditional
Conditional \triangleq **ifc** Instruction **end**

← Actual version on page 437.

distilled even further here: **ifc** represents “**if** **Condition** **then**” and **ast** represents **Assignment**, both treated as terminals for this discussion.

It is easy to see how to generate successive sentences of the language by interpreting these productions in a bottom-up, fixpoint-equation style:

ast
ifc ast end
ifc ifc ast end end
ifc ifc ifc ast end end end

and so on. You can also look again, in light of the notion of bottom-up recursive computation, at the earlier discussion of the little **Game** language.

← “Recursive grammars”, page 307.

It is possible to generalize this approach to arbitrary grammars by taking a matrix view of a BNF description.

→ Exercise 14-E.17, page 504.

14.8 CONTRACTS FOR RECURSIVE ROUTINES

We have learned to equip our classes and their features with **contracts** stating their correctness properties: routine preconditions, routine postconditions, class invariants; the same concerns applied to algorithms gave us loop variants and loop invariants. How does recursion affect the picture?

We have already seen the notion of **recursion variant**. If a routine is recursive directly or indirectly, you should include a mention of its variant. As noted, we do not have specific language syntax for this but add a clause

← “*Touch of Methodology: Recursion Variant*”, page 475.

```
-- variant: integer_expression
```

to the routine’s header comment.

A recursive routine may have a precondition and postcondition like any other routine. Because ensuring a precondition is always the responsibility of the caller, and here the routine is its own caller, the novelty is that you must ensure that all calls within the routine (or, for indirect recursion, in associated routines) satisfy the precondition.

Here is the Towers of Hanoi routine with more complete contracts; the new clauses, expressed as comments, are highlighted.

← The original was on page 443.

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
  -- Transfer n disks from source to target, using other as intermediate
  -- storage, according to rules of Tower of Hanoi puzzle.
  -- invariant: disks on each needle are piled in decreasing size.
  -- variant: n
require
  non_negative: n >= 0
  different1: source /= target
  different2: target /= other
  different3: source /= other
  -- source has n disks; any disks on target and other are all
  -- larger than all the disks on source.
do
  if n > 0 then
    hanoi (n-1, source, other, target)
    move (source, target)
    hanoi (n-1, other, target, source)
  end
ensure
  -- Disks previously on source are now on target, in same order,
  -- on top of those previously there if any; other is as before.
end
```

A properly specified recursive routine has a **recursion invariant**: a set of properties that must hold both before and after each execution. In the absence of a specific language mechanism they will just appear twice, in the precondition as well as in the postcondition; for clarity you may also, as here, include them in the header comment under the form

```
-- invariant: integer_expression
```

This is not a language construct but relies on the following convention:

- If the recursion invariant is just pseudocode expressed as a comment, as in this example, do not repeat it in the precondition and postcondition; here this means omitting from the precondition and postcondition the property that any disks on the affected needles are piled up in decreasing size.
- Any recursion invariant clause that is formal (a boolean expression) should be included in the precondition and postcondition, since there is no other way to express it formally.

14.9 IMPLEMENTATION OF RECURSIVE ROUTINES

Recursive programming works well in certain problem areas, as illustrated by the examples in this chapter. When recursion facilitates your job you should not hesitate to use it, since in modern programming languages you can take recursion for granted.

Since there is usually no direct support for recursion in machine code, compilers for high-level languages must map a recursively expressed algorithm into a non-recursive one. The applicable techniques are obviously important for compiler writers, but even if you do not expect to become one it is useful to know the basic ideas, both to gain further insight into recursion (complementing the perspectives opened by previous sections) and to understand the potential performance cost of using recursive algorithms.

We will look at some recursive schemes and ask ourselves how, if the language did *not* permit recursion, we could devise non-recursive versions, also called **iterative**, achieving the same results.

A recursive scheme

Consider a routine r that calls itself:

```

r (x: T)
  do
    code_before
    r (y)
    code_after
  end

```

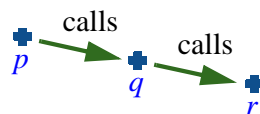
There might be several recursive calls, but we look at just one. What does it mean — if we revert to a top-down view — to execute that call?

The presence of recursion implies that neither the beginning of the routine's code nor its end are just what they pretend to be:

- When *code_before* executes, this is not necessarily the beginning of a call $a.r(y)$ or $r(y)$ executed by some client routine: it could result from an instance of r calling itself recursively.
- When *code_after* terminates, this is not necessarily the end of the r story: it may simply be the termination of one recursively called instance; execution should resume for the last instance started and not terminated.

Routines and their execution instances

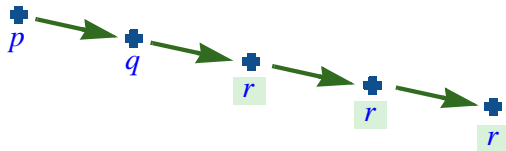
The key novelty in the last observation is the concept of **instance** (also called **activation**) of a routine. We know that classes have instances — the “objects” of object-oriented program execution — but we have not yet thought of routines in a similar way.



*A call chain,
without recursion*

At any moment during a program's execution, the state of the computation is characterized by a **call chain** as pictured above: the root procedure p has called q which has called r ... When an execution of a routine in the chain, say r , terminates, the suspended execution of the calling routine, here q , resumes just after the place where it had called r .

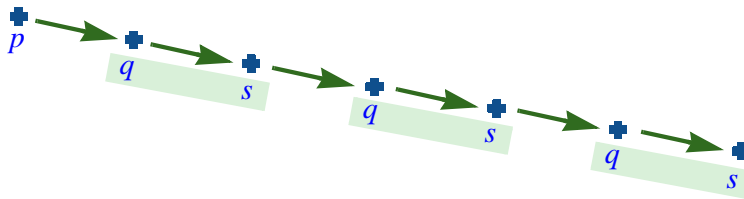
In the absence of recursion, we did not need to make the concept of routine instance explicit since any routine had, at any time, at most one active instance. With recursion, the call chain may include two or more instances of the same routine. Under *direct* recursion they will be contiguous:



Call chain with direct recursion

For example a call *hanoi* (2, *s*, *t*, *o*) immediately starts a call *hanoi* (1, *s*, *o*, *t*) which starts a call *hanoi* (0, *s*, *t*, *o*); at that stage we have three instances of the procedure in the call chain.

A similar situation arises with *indirect* recursion:



Call chain with indirect recursion

Preserving and restoring the context

All instances of a routine share their program code; what distinguishes them is their execution *context*. We have seen that in a useful case of recursion the context of every call must differ by at least one element. The context elements characterizing a routine instance (rather than object states) are: ← *R2, page 474.*

- The values of the actual routine arguments, if any, for the particular call.
- The values of the local variables, if any.
- The location of the call in the text of the calling routine, defining where execution should continue once the call completes.

As we saw when studying how stacks support the execution of programs in modern languages, a data structure representing such a routine execution context is called an **activation record**. ← *“Using stacks”, page 421.*

Assume a programming language that does *not* support recursion. Since at any time during execution there is at most one instance of any routine, the compiler-generated program can use a single activation record per routine. This is known as **static allocation**, meaning that the memory for all activation records can be allocated once and for all at the beginning of execution.

With recursion each activation of the routine needs its own context. This leaves two possibilities for implementation:

- I1 We can resort to *dynamic allocation*: whenever a routine instance starts, create a fresh activation record to hold the routine's context. Use this activation record whenever the routine execution needs to access an argument or local variable; use it too on instance termination, to determine where execution must continue in the caller's code. Resuming the caller's execution implies going back to its own activation record.
- I2 To save space, we may note that the reason for keeping context information in an activation record is to be able to *restore* it when an execution resumes after a recursive call. An alternative to saving that information is to *recompute* it. This is possible when the change performed by the recursive call is **invertible**. The recursive calls in procedure *hanoi* (n, \dots) are of the form *hanoi* ($n - 1, \dots$); rather than storing the value of n into an activation record, creating a new record holding the value $n - 1$, then restoring the previous record on return, we may use a single location for n in all recursive instances, as with static allocation: at call time, we decrease the value by one; at return time, we *increase* the value by one.

The two techniques are not exclusive: you can save space by using I2 for values whose transformation in calls (such as replacing n by $n - 1$) admits an easily implemented inverse, and retain an activation record for the rest of the context. The decision may involve a space-time tradeoff if the inverse transformation, unlike the $n := n + 1$ example, is computationally expensive.

Using an explicit call stack

Of the two strategies for handling routine contexts let us look first at I1, which relies on explicit activation records.

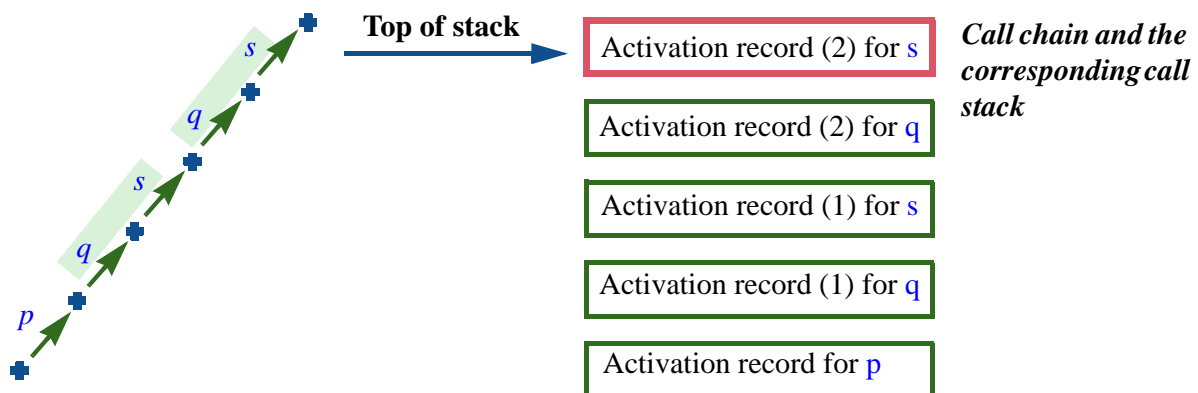
Like activation records, *objects* are created dynamically, as a result of **create** instructions. The program memory area devoted to dynamically allocated objects is known as the **heap**. But for activation records of routines we do not need to use the heap since the patterns of activation and deactivation are simple and predictable:

← “Creating simple objects”, 6.4, page 118.

- A call to a routine requires a new activation record.
- On returning from that call, we may forget this activation record (it will never be useful again, since any new call will need its own values), and we must restore the caller's activation record.

This is a *last-in, first-out* pattern for which we have a ready-made data structure: stacks. The stack of activation records will reflect the call chain, pictured here going up:

← “Stacks”, 13.11, page 420.



Call chain and the corresponding call stack

We have encountered the stack of activation records before: it is the **call stack** which keeps track of routine calls during execution. If you are programming in a language supporting recursion, the call stack is the responsibility of the code generated by the compiler. Here we are looking at how to manage it ourselves.

← “Using stacks”, page 421.

You can use an explicit stack of activation records to produce an iterative equivalent of a recursive routine:

“Iterative”, defined on page 486, means non-recursive.

- To access local variables and arguments of the current routine: always use the corresponding positions in the activation record at the top of the stack.
- Instead of a recursive call: create a new activation record; initialize it with the value of the call's arguments and the position of the call; push it on the stack; and branch back (**goto**) to the beginning of the routine's code.
- Instead of a return: return only if the stack is empty (no suspended call is pending); otherwise, restore the arguments and local variables from the activation record at the top of the stack, pop the stack, and branch to the appropriate instruction based on the call position information found in the activation record.

Note that both translation schemes involve **goto** instructions. That is fine if we are talking about the machine code to be generated by a compiler; but when it is a manual simulation of recursion in a high-level language we have learned to avoid the **goto** and in fact Eiffel has no such instruction. We will have to write **gotos** temporarily, then replace them by appropriate control structures.

← “The goto instruction”, page 183.

Recursion elimination essentials

Let us see how the scheme works for the body of *hanoi* with its two recursive calls. We use a stack of activation records, called just *stack*:

```
stack: STACK [RECORD]
```

with a small auxiliary class *RECORD* to describe activation records:

```
note
  description: "Data associated with a routine instance"
class RECORD create
  make
feature -- Initialization
  make (n: INTEGER; c: INTEGER; s, t, o: CHARACTER)
    -- Initialize from count n, call c, source s, target t, intermediary o.
  do
    count := n ; call := c ; source := s ; target := t ; other := o
  end
feature -- Access
  count: INTEGER.
    -- Number of disks.
  call: INTEGER
    -- Identifies a recursive call: 1 for the first, 2 for the second.
  source, target, other: CHARACTER -- First call
    -- Needles.
end
```

(Instead of a full-fledged class we could also just use tuples.) An instance of the class represents the context of a call: the number of disks being moved (*count*), the three needles in the order used by the call, and *call* telling us whether this execution, if coming from a recursive call, came from the first or second call in

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
  do
    if n > 0 then
      hanoi (n-1, source, other, target)  -- First call
      move (source, target)
      hanoi (n-1, other, target, source)  -- Second call
    end
  end
```

We use the *stack* of activation records to provide a non-recursive version of the procedure, temporarily relying on **gotos**, as shown on the following page.

```

iterative_hanoi (n: INTEGER; source, target, other: CHARACTER)
  local  -- We need locals representing arguments to successive calls:
    count: INTEGER
    x, y, z, t: CHARACTER
    call: INTEGER
    top: RECORD
  do  -- Initialize locals to values of arguments in original call:
    count := n; x := source; y := target; z := other

start:  if count > 0 then
        -- Translation of hanoi (n-1, source, other, target):
        stack.put (create {RECORD}. make (count, 1, x, y, z))
        count := count - 1
        t := y ; y := z ; z := t
        goto start

after_1:  move (x, y)

        -- Translation of hanoi (n-1, other, target, source):
        stack.put (create {RECORD}. make (count, 2, x, y, z))
        count := count - 1
        t := x ; x := z ; z := t
        goto start

end

        -- Translation of routine return:
after_2: if not stack.is_empty then
        top := stack.item  -- Top of stack
        count := top.count
        x := top.source ; y := top.target ; z := top.other
        call := top.call ; stack.remove
        if call = 2 then
            goto after_2
        else
            goto after_1
        end

end

        -- No else clause: the routine terminates when
        -- (and only when) the stack is empty.

end

```

Warning: because of the **goto** instructions and labels this is not legal Eiffel. The **gotos** will be removed next.

This block is referred to below as **SAVE_AND_ADAPT_1**

Referred to below as **MOVE**

Referred to below as **SAVE_AND_ADAPT_2**

Referred to below as **RETRIEVE**

The body of *iterative_hanoi* derives from *hanoi* through systematic application of recursion elimination techniques:

- D1 For every argument, introduce a local variable. The example uses a simple naming convention: *x* for *source* and so on.
- D2 Assign on entry the value of the argument to the local variable, then work exclusively on that variable. This is necessary because a routine may not change the value of its arguments ($n := \text{some_new_value}$ is invalid).
- D3 Give a label, here *start*, to the routine's original first instruction (past the local variable initializations added by D2).
- D4 Introduce another local variable, here *call*, with values identifying the different recursive calls in the body. Here there are two recursive calls, so *call* will have two possible values, arbitrarily chosen as **1** and **2**.
- D5 Give a label, here *after_1* and *after_2*, to the instructions immediately following each recursive call.
- D6 Replace every recursive call by instructions which:
- Push onto the stack an activation record containing the values of the local variables.
 - Set the values of the locals representing arguments to the values of the call's actual arguments; here the recursive call replaces *n* by *n* - 1 and swaps the values of *other* and *target*, using the local variable *swap* for that purpose.
 - Branch to the first instruction.
- D7 At the end of the routine, add instructions which terminate the routines' execution only if the stack is empty, and otherwise:
- Restore the values of all local variables from the activation record at the top of the stack.
 - Also from that record, obtain the call identification
 - Branch to the appropriate post-recursive-call label among those set in D5.

This is the general scheme applicable to the derecursification of any recursive routine, whether a programmer is carrying it out manually, as we are now doing, or — the more common situation — compilers include it in the code they generate for routine calls.

We will see next how to simplify it — including *goto* removal — with the help of some deeper understanding of the program structure; in the meantime, make sure you fully understand this example of brute-force derecursification.

If, as I hope, you do find the transformation (if not the result) simple and clear, you may enjoy, as a historical aside, an anecdote reminding us that what is standard today was not always obvious. It is told by Jim Horning, a computer scientist well known for his own contributions, in particular to the area of formal methods:



Naur & Horning
(2006)

Touch of History:
When recursion was thought impossible
(as told by Jim Horning)

In the summer of 1961 I attended a lecture in Los Angeles by a little-known Danish computer scientist. His name was Peter Naur and his topic was the new language Algol 60. In the question period, the man next to me stood up. “It seems to me that there is an error in one of your slides.”

Peter was puzzled, “No, I don’t think so. Which slide?”

“The one that shows a routine calling itself. That’s impossible to implement.”

Peter was even more puzzled: “But we have implemented the whole language, and run all the examples through our compiler.”

The man sat down, still muttering to himself, “Impossible! Impossible!”. I suspect that much of the audience agreed with him.

At the time it was fairly common practice to allocate statically the memory for a routine’s code, its local variables and its return address. The call stack had been independently invented at least twice in Europe, under different names, but was still not widely understood in America.

Slightly abridged from Jim Horning’s blog at horningtales.blogspot.com/2006/07/recursion.html. Reproduced with permission.

The reference to independent inventions of the notion of call stack is probably to Friedrich Bauer from Munich, who used the term *Keller* (cellar), and Edsger Dijkstra from Holland, when implementing his own Algol 60 compiler.

Simplifying the iterative version

The code given above looks formidable, especially against the simplicity of the original recursive version. Indeed, with a truly recursive algorithm like this one an iterative version will never reach the same elegance. But we can get close by reviewing the sources of complication:

- We may replace the **gotos** by structured programming constructs.
- By identifying invertible operations, we may limit the amount of information to be stored into and retrieved from the stack. ← See I2, page 489.
- In some cases (tail recursion) we may bypass the stack altogether.



Bauer (2005)

The last two kinds of simplification can also be important for performance, since all this pushing and popping takes time, as well as space on the stack.

On the Hanoi example let us start by getting rid of the **goto** eyesores. To abstract from the details of the code we express the body of *iterative_hanoi* as ← From page 492.

```

INIT
start:  if count > 0 then
        SAVE_AND_ADAPT_1
        goto start
after_1: MOVE
        SAVE_AND_ADAPT_2
        goto start
      end
after_2: if not stack.is_empty then
        RETRIEVE
        if call = 2 then goto after_2 else goto after_1 end
      end

```

count is an integer variable; the instructions *I0*, *I1* and *I2* can change its value.

with *SAVE_AND_ADAPT_1* representing the storing of information into the stack and change of values before the first call, *SAVE_AND_ADAPT_2* the same for the second call, *RETRIEVE* the retrieval from the stack of local variables including *call*, *MOVE* the basic move operation, and *INIT* the initialization of local variables from the arguments.

This is the example of **goto** structure that served (with abstract names for the instructions and conditions, *I1*, *C1* etc.) as illustration in the discussion of **goto** removal. The result was

```

from INIT until over loop
  from until count <= 0 loop
    SAVE_AND_ADAPT_1
  end
  from stop := stack.is_empty until stop loop
    RETRIEVE
    stop := (stack.is_empty or (call /= 2))
  end
  over := (stack.is_empty and (call = 2))
  if not over then MOVE ; SAVE_AND_ADAPT_2 end
end

```

← “Appendix: an example of goto removal”, page 205. The resulting **goto**-less structure appears on page 206. The local variable *over* is initialized to **False**.

which we can immediately simplify, getting rid in particular of the *stop* boolean variable:

```

from INIT until over loop
  from until count = 0 loop SAVE_AND_ADAPT_1 end
  from call := 2 until stack.is_empty or call = 1 loop RETRIEVE end
  over := (stack.is_empty and (call = 0))
  if not over then MOVE ; SAVE_AND_ADAPT_2 end
end

```

The simplifications result from an analysis of possible changes to the values of the variables:

- Since *count* can never become negative because of the precondition of *hanoi* and the test conditioning recursive calls, it is legitimate to replace that test, *count* ≤ 0, by *count* = 0.
- To get rid of *stop* we note that any value *call* gets out of *RETRIEVE* can only be 1 or 2, since these are the possible values stored onto the stack; so we can replace *call* ≠ 2 by *call* = 1, then set *call* to 2 the first time around so that this particular condition is only taken into account for the second and later iterations if any.

Tail recursion

A standard technique that helps reduce the overhead of stack pushing and popping relies on the observation that it is not necessary to store context information, and later retrieve it, if the algorithm does not need this information any more; this is the case in particular for a recursive call that is the *last* operation executed by an instance of the recursive routine.

This simplification applies to the *hanoi* example. The second recursive call is the last instruction executed by an activation of the routine. This means that *SAVE_AND_ADAPT_2* is not necessary, or more precisely that the only information it must preserve is *call*, since in getting back from a call you need to know whether it was an instance of the first or the second one: in the first case you need to pop the other values (*count*, *x*, *y*, *z*), in the second you don't.

A good compiler can detect tail recursion and apply this optimization to improve the performance of a recursive algorithm.

In the *hanoi* case it is superseded by another optimization, which almost entirely gets rid of the stack and which we will now see. You should, however, practice tail recursion elimination by implementing the above algorithm and removing the unneeded push operations.

→ Exercise 14-E.14,
page 504.

Taking advantage of invertible functions

Using a stack to store the values before a call and retrieve them afterwards is the default technique and always works, but we saw earlier that an alternative exists: reverting the transformation of arguments. In the *hanoi* case this turns out to be possible for *all* arguments: ← *I2, page 489.*

- The transformation of *count* prior to each call, $count := count - 1$, has an obvious inverse: $count := count + 1$.
- For the other arguments, representing needles, the transformation is $swap_{23}$ for the first call and $swap_{12}$ for the second, if we call $swap_{ij}$ the operation that swaps the variables representing the *i*-th and *j*-th needles (for example $swap_{23}$ is $t := y ; y := z ; z := t$). But every $swap_{ij}$ is its own inverse: applying it a second time restores the original values.

So we do not actually need to store any of *count*, *x*, *y* and *z* on the stack: it suffices, at the time of a *RETRIEVE*, to apply the appropriate inverse operation. Specifically, *RETRIEVE* becomes:

```

“Retrieve the value of call”
count := count + 1
if call = 1 then swap23 else swap13 end

```

A stack remains necessary, but only to record and retrieve the values of *call*. The simplification becomes even more dramatic if we notice that *call* only has two possible values, 1 and 2, which were just a convention to identify the two recursive calls. Let us instead call them 1 and 0. There is a simple representation for a stack of 0/1 (or boolean) values: if you know for certain that the stack’s height plus one cannot exceed the bit size of an integer — typically 64 on modern computers, until recently 32 —, just use a *single integer*, say *s*, for the stack. It is a matter of considering the 0s and the 1s of the binary representation, even if you do not know the details of number representation on your computer. The operations are:


```

s = 1           -- Is the stack empty?
s := 1         -- Initialize to an empty stack
s := 2 * s     -- Push a 0
s := 2 * s + 1 -- Push a 1
b := s \\ 2    -- Obtain (into b) the top of the stack (\\ is remainder)
s := s // 2    -- Pop the stack (// is integer division)

```

The first is a boolean expression, the others are instructions.

Here is the result of a typical sequence of such instructions:

<i>Instruction</i>	<i>Goal</i>	<i>Result</i>	<i>Binary representation of s (leftmost zeroes omitted)</i>															
$s := 1$	-- Start empty	$s = 1$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td></tr></table>											1				
										1								
$s := 2 * s$	-- Push a 0	$s = 2$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td></tr></table>											1	0			
										1	0							
$s := 2 * s + 1$	-- Push a 1	$s = 5$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td></tr></table>											1	0	1		
										1	0	1						
$s := 2 * s + 1$	-- Push a 1	$s = 11$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>											1	0	1	1	
										1	0	1	1					
$s := 2 * s$	-- Push a 0	$s = 22$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>											1	0	1	1	0
										1	0	1	1	0				
$s := s // 2$	-- Pop	$s = 11$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>											1	0	1	1	
										1	0	1	1					
$b := s // 2$	-- Get top	$b = 1$	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> 											1	0	1	1	
										1	0	1	1					

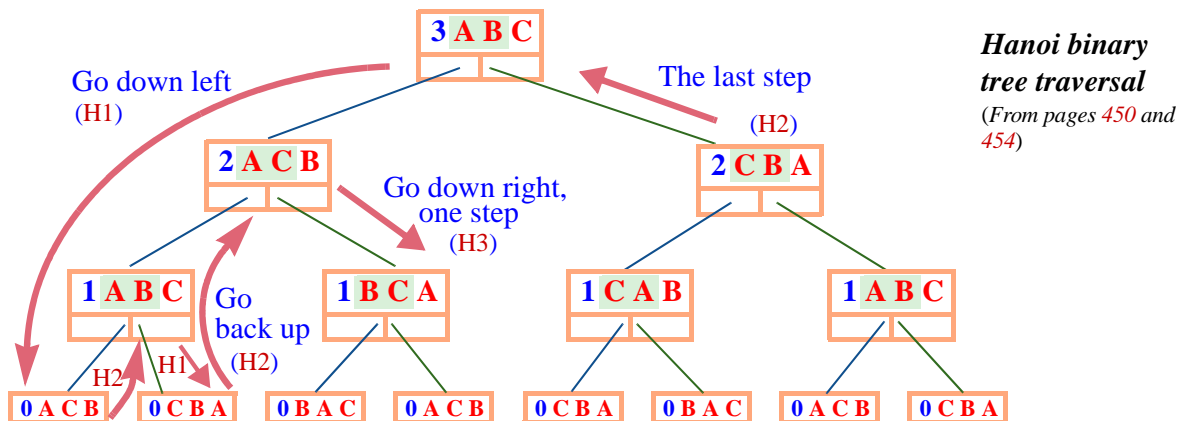
The binary representation of integers, shown in the last column, has the largest weights on the left (“big-endian” convention). The top of a non-empty stack is 0 if the number is even, 1 if it is odd.

This technique of using a single integer to represent a stack of boolean values can be used safely whenever you have a guaranteed limit on the stack size. In the *hanoi* example this is not a problem since 2^{63} or even 2^{31} are more moves than can be handled in any reasonable time.

Combining the previous observations leads to a simpler and more efficient form of the *iterative_hanoi* algorithm with arguments *n*, *source*, *target*, *other*:

<pre> from count := n ; x := source ; y := target ; z := other ; s := 1 until over loop from until count = 0 loop swap₂₃ ; s := 2 * s + 1 ; count := count - 1 end from call := 0 until s = 1 or call = 1 loop call := s // 2 ; s := s // 2 ; count := count + 1 if call = 1 then swap₂₃ else swap₁₃ end end over := ((s = 1) and (call = 0)) if not over then move (x, y) swap₁₃ ; s := 2 * s ; count := count - 1 end end </pre>	<p><i>over</i> is initialized to False as usual.</p> <p>-- Go down left -- (H1, see next page)</p> <p>-- Go back up -- (H2)</p> <p>-- Visit node, -- go down right -- (H3)</p>
---	---

Although this is the result of a systematic transformation and not the kind of program that you would normally write (recursion is simpler and clearer), it is interesting to follow the execution on this form too, relating it to the original recursive version and specifically to the binary *execution tree* from the beginning of this chapter, showing the execution as an inorder traversal:



As noted next to the algorithm, it has three components:

H1 Go down left, as far as possible, until you reach a leaf. The leaves are at $n = 0$ (*count* = 0 in this version), although earlier figures showing this tree stopped at 1 since nothing visible from the outside happens at level 0.

H2 Go back up. As long as you are coming from the right just continue going up, since this corresponds to the second recursive call and there is nothing more to do with this instance of the routine.

H3 Having gone up one left branch, perform the visiting operation (move one disk from x to y , and go down *one* right branch).

This is repeated until, coming up from the right (H2), you find the stack empty.

When going down (H1, H3), you decrement *count* and swap y and z if going left (H1), x and z if going right (H3); when coming back up (H2), you restore the original values by incrementing *count* and doing the appropriate swap depending on whether you are coming back from the left or from the right — which you find out by looking at the top of the stack, meaning the parity of s as given by *call*.

14.10 KEY CONCEPTS LEARNED IN THIS CHAPTER

- It is often convenient to define a concept recursively, meaning that the definition uses one or more other instances of the concept itself.
- For the definition to be useful, any occurrence of the concept in its definition must apply it to a smaller target, and there must be at least one case for which the definition is non-recursive, so that any application of the definition reduces in the end to a combination of elementary cases.
- Recursive definitions can be useful in particular for routines, data structures and grammars.
- Any loop can be expressed in an equivalent recursive form, through a simple transformation.
- The other way around, any recursive algorithm has a recursion-free equivalent, but the transformation is more delicate; it requires changing the control flow, and recording the value of local information prior to every recursive call so as to retrieve it later, either by using a stack or by spotting invertible transformations.

New vocabulary

Activation	Activation record	Alpha-beta
Backtracking	Binary tree	Call chain
Depth-first	Direct recursion	Indirect recursion
Inorder	Iterative	Instance (of a routine)
Minimax	Non-creative	Postorder
Preorder	Recursion	Recursive
Recursive definition	Traversal	

14-E EXERCISES

14-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

14-E.2 Too much recursion?

Is the definition of “recursive definition” a recursive definition?

← Page 435.

14-E.3 Binary search trees with repetitions

For every binary search tree routine in this chapter, rewrite the declaration (if needed) to permit multiple occurrences of a given *item* value in a tree as discussed after the initial definition.

← Page 455.

14-E.4 A programming language without program texts

This exercise addresses language processing techniques seen in an earlier chapter; the solution requires recursion.

→ See also exercises 16-E.4, page 617 to 16-E.6, page 618 in the inheritance chapter.

The goal is to write an interpreter and a compiler for an elementary programming language. To avoid dealing with concrete syntax, the tools will directly manipulate data structures rather than texts.

Our little language is called WASO (acronym for With Abstract Syntax Only) and has the following properties:

- The only data type is “integer”.
- Variables, all of integer type, do not need to be declared. A variable name is an arbitrary string.
- Integer constants can be used, such as 1.
- Integer expressions can be formed with addition, subtraction, multiplication and integer division.
- There are two kinds of instruction: assigning an expression to a variable, and printing the value of a variable.
- A WASO program consists of a sequence of assignments and a sequence of print instructions, either or both of which can be empty.
- The execution of a program consists of initializing all variables to zero, executing the assignments in sequence, and executing the print instructions in sequence.

So a typical program — written out here as if WASO had a textual representation (concrete syntax), although this is not part of the language definition — is:

```
assign
  x := 3
  y := 5
  x := 2 * (x + (y // 3))
then
  print x
  print z
end
```

This is only one possible concrete syntax.

The execution of this program prints the single value 8.

The concrete syntax is only one of many possible choices. Another would use the keyword **print** instead of **then** and in the second clause list only the variables to be printed, without repeating **print**.

The assignment:

- 1 Write a set of classes, including *PROGRAM*, *ASSIGNMENT*, *PRINT* and *EXPRESSION*, with the associated features including creation procedures, to build abstract syntax trees representing WASO programs.
- 2 Add a class with a procedure that uses these classes and features to create an abstract syntax tree representing the above example program.
- 3 Add to class *PROGRAM* a procedure *write_out* that produces a textual (concrete) representation of a WASO program, as given out for the example. Run it on the example tree from step 2 and check that the output is the above text. *Hint*: you need a recursive procedure performing a traversal, similar to those introduced for binary trees in this chapter.
- 4 Write a WASO interpreter, in the form of a procedure *interpret* in class *PROGRAM* which executes the program and produces the expected output. Run it on the example and check the result (which as noted should be the single value 8).
- 5 Write a WASO-to-Eiffel compiler, in the form of a procedure *compile* in class *PROGRAM* which produces an Eiffel system implementing the semantics of the source WASO program: a root class with an appropriate creation procedure, and any other classes needed. Run it on the example; use Eiffel Studio to Eiffel-compile the output; run it on the example and check the result.

Terminology note: the result of step 5 is an *unparser*, producing a text representation from an internal representation such as an abstract syntax tree — the reverse of what a *parser* does.

14-E.5 Non-recursive insertion

Write a version of *put* for binary search trees using a loop rather than recursion. ← Page 458.
(**Hint**: you may use for inspiration the non-recursive version of the search function *has*.)

14-E.6 Recursive reversal

Retaining the same assumptions (a list of stops is known through its first cell, of type *STOP*, giving access to the rest through repeated application of *next*), rewrite the function *reversed* from the discussion of references so that it uses recursion rather than a loop. (See also the next exercise.) ← Page 261.

14-E.7 Reversing a list, functional style

Write a recursive function that produces the reverse of a linked list (the argument and the result should be of type *LINKED_LIST[G]*, from EiffelBase). ← “Functional programming and functional languages”, page 324.
Keep pointer manipulations to a minimum and remain as close as possible to the style of the *reversed* function given as an example of Haskell programming. Analyze the time and space complexity of your solution.

14-E.8 Backtracking curtailed

Adapt the general backtracking algorithm so that it keeps track of previously explored positions and discards any path leading to such a position. You may assume that *PATH* has a query *position* defining a path's terminal position. ← Page 461.

14-E.9 Cycles despised

Adapt the general backtracking algorithm so that it does not explore paths longer than *path_cutoff*, a given integer value. ← Page 461.

14-E.10 Properties of a function graph

(This exercise calls for mathematical analysis, not a programming solution.) In the successive approximations H_i of the graph of the Towers of Hanoi function, assuming three needles 'A', 'B', 'C': ← "The towers, bottom-up", page 483.

- 1 What is the number of pairs in H_i ?
- 2 Give a mathematical formula for H_i .

14-E.11 Programming a function graph bottom-up

- 1 Devise a class of which every instance represents an arguments-result pair, of the form $[(n, s, t, o), \langle \dots \rangle]$, for the Towers of Hanoi function graph. ← "The towers, bottom-up", page 483.
- 2 Based on the preceding class, devise another to represent the function graph as a whole.
- 3 From this class and the rules [5] and [6] defining the function graph in the bottom-up interpretation of recursion, write a program that produces the i -th approximation of the graph, H_i , for any i . The algorithm may use loops, but it may not use recursion.
- 4 Use this program to print out sequences of moves (with source 'A' and target 'B') for a few values of i ; check that the results coincide with those of the recursive procedure.

14-E.12 Bottom-up view of binary tree algorithms

Consider a recursive algorithm for binary tree traversal; you may choose preorder, inorder or postorder.

- 1 Taking inspiration from the bottom-up analysis of the Towers of Hanoi solution, devise a model to interpret the traversal as a function returning a sequence of nodes. ← "The towers, bottom-up", page 483.
- 2 Write a recursive "definition" of this function.
- 3 Express this "definition" as a fixpoint equation on the function graph, using T_i as the name of the graph for binary trees of height i .
- 4 Use the definition to produce (either manually or by writing a small program) H_5 for the example binary tree, and the resulting traversal order. ← From the figure on page 447.

14-E.13 Recursion without optimization

(This exercise requires access to a compiler for a programming language such as C or C++ with support for **goto** instructions.) Implement and test the direct iterative translation of the *hanoi* procedure, in its initial version using **gotos** and a stack without optimization. ← *iterative_hanoi*, page 492.

14-E.14 Saving on stack saving

- 1 Implement and test the **goto**-free iterative, stack-based version of the Tower of Hanoi problem. ← *Algorithm on page 495.*
- 2 Improve the solution through tail recursion optimization, avoiding unnecessary saves in the second call.
- 3 (Only if you have solved the previous exercise.) Apply the same optimization to the version using **goto** instructions.

14-E.15 Traversal without a stack

We saw that implementing recursion only requires a technique to invert the transformation of arguments in recursive calls; a stack is just one possible way to satisfy this requirement. Using a suitable inversion technique, implement binary tree traversal, for example inorder, non-recursively and without any stacks except possibly a stack of boolean values (or, equivalently, a bit in every node). ← *“Implementation of recursive routines”, 14.9, page 486.*

Hint: temporarily overwrite tree links to remember where you came from.

Counter-hint: you could find a solution by running Web searches for the words *Deutsch*, *Schorr* and *Waite* (names of authors of a famous algorithm based on this idea). Don't; rather, design an algorithm, then look up existing references if you wish.

14-E.16 Transitive closure

(This exercise refers to a later chapter.) Restate the definition of transitive closure as a recursive definition. → *Page 513.*

14-E.17 Matrix algebra on BNF productions

(This exercise requires basic knowledge of linear algebra.) Consider a BNF production, such as the small example used in this chapter, or more extensive ones from earlier chapters, involving only Concatenation and Choice productions (no Repetition, as it can be replaced by combinations of the other two).

- 1 Treating concatenation of tokens as “multiplication” and alternative choices as “addition”, show that it is possible to express the grammar as a matrix equation $X = A * X + B$, where X is the vector of nonterminals, A is a matrix of terminals and nonterminals, and B is a vector.
- 2 Discuss ways of solving this equation by following the model discussed for fixpoint equations.