Einführung in die Programmierung
Introduction to Programming
Prof. Dr. Bertrand Meyer
October 2006 - February 2007

Lecture 15: Recursion

The story of the universe

*According to Édouard Lucas, Récréations mathématiques, Paris, 1883.
This is my translation; the original is on the next page.

In the great temple of Benares, under the dome that marks the center of the world, three diamond needles, a foot and a half high, stand on a copper base.

God on creation strung 64 plates of pure gold on one of the needles, the largest plate at the bottom and the others ever smaller on top of each other. That is the tower of Brahmag.

The monks must continuously move the plates until they will be set in the same configuration on another needle.

The rule of Brahmag is simple: only one plate at a time, and never a larger plate on a smaller one.

When they reach that goal, the world will crumble into dust and disappear.

The story of the universe

*According to Édouard Lucas, Récréations mathématiques, Paris, 1883

Dans le grand temple de Bénares, sous le dôme qui marque le centre du monde, repose un socle de cuivre équipé de trois aiguilles verticales en diamant de 50 cm de haut.

À la création, Dieu enfila 64 plateaux en or pur sur une des aiguilles, le plus grand en bas et les autres de plus en plus petits. C’est la tour de Brahmag.

Les moines doivent continûment déplacer les disques de manière que ceux-ci se retrouvent dans la même configuration sur une autre aiguille.

La règle de Brahmag est simple: un seul disque à la fois et jamais un grand plateau sur un plus petit.

Arrivé à ce résultat, le monde téméra en poussière et disparaîtra.

The towers of Hanoi

How many moves?

Assume \( n \) disks \( (n \geq 0) \): three needles called source, target and other.

The largest disk can only move from source to target if it’s empty; thus all the other disks must be on other.

So the minimal number of moves for any solution is:

\[
H_n = H_{n-1} + H_{n-1} + 1
\]

\[
H_n = 2H_{n-1} + 1
\]

Since \( H_1 = 1 \), this implies:

\[
H_n = 2^n - 1
\]
This reasoning gives us an algorithm!

$hanoi(n \in \text{INTEGER}, \text{source}, \text{target}, \text{other} \in \text{CHARACTER})$ is
-- Transfer $n$ disks from $\text{source}$ to $\text{target}$,
-- using $\text{other}$ as intermediate storage.

require
  non_negative: $n > 0$
  different: $\text{source} \neq \text{target}$
  different2: $\text{target} \neq \text{other}$
  different3: $\text{source} \neq \text{other}$
do
  if $n > 0$
    $hanoi(n-1, \text{source}, \text{other}, \text{target})$
    $\text{move}(\text{source}, \text{target})$
    $hanoi(n-1, \text{other}, \text{target}, \text{source})$
end

The tower of Hanoi

A possible implementation for $\text{move}$

$\text{move}(\text{source}, \text{target} \in \text{CHARACTER})$ is
-- Prescribe move from $\text{source}$ to $\text{target}$.
require
different: $\text{source} \neq \text{target}$
do
  $\text{io.put_character}(\text{source})$
  $\text{io.put_string}(' \to ')$
  $\text{io.put_character}(\text{target})$
  $\text{io.put_new_line}$
end

An example

Executing the call

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

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

$\begin{align*}
A & \to C & B & \to C & B & \to A \\
A & \to B & A & \to C & C & \to B \\
C & \to B & A & \to B & A & \to C \\
A & \to C & C & \to B & A & \to B \\
B & \to A & C & \to A & C & \to B
\end{align*}$

The general notion of recursion

A definition for a concept is recursive if it involves an instance of the concept itself.

Notes:
- The definition may use more than one "instance of the concept itself".
- Recursion is the use of a recursive definition.
### Examples

- Recursive routine
- Recursive grammar
- Recursively defined programming concept
- Recursive data structure

### Recursive routine

Direct recursion: body includes a call to the routine itself

Example: routine `hanoi` for the preceding solution of the Towers of Hanoi problem

### Recursion can be indirect

Routine `r` includes a call to routine `s`

`s` includes a call to `r`

More generally: `r_1` calls `r_2` calls ... calls `r_n` calls `r`.

### Recursive grammar

- **Instruction**: `Assignment | Conditional | Compound |` ...
- **Conditional**: `if Expression then Instruction else Instruction end`

### Defining lexicographical order

Problem: define the notion that word `w1` is "before" word `w2`, according to alphabetical order.

**Conventions:**

- A word is a sequence of zero or more letters.
- A letter is one of: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- For any two letters it is known which one is "smaller" than the other; the order is that of the preceding list.

### Examples

- `ABC` before `DEF`
- `AB` before `DEF`
- empty word before `ABC`
- `A` before `AB`
- `A` before `ABC`
A recursive definition

The word \( w_1 \) is “before” the word \( w_2 \) if and only if one of the following conditions holds:

- \( w_1 \) is empty and \( w_2 \) is not empty
- Neither \( w_1 \) nor \( w_2 \) is empty, and the first letter of \( w_1 \) is smaller than the first letter of \( w_2 \).
- Neither \( w_1 \) nor \( w_2 \) is empty, their first letters are the same, and the word obtained by removing the first letter of \( w_1 \) is (recursively) before the word obtained by removing the first letter of \( w_2 \).

Recursive data structure

Let \( G \) be some type. A binary tree over \( G \) is either:

- Empty
- A node, consisting of three elements:
  - A value of type \( G \)
  - A binary tree over \( G \), called the left child of the node
  - A binary tree over \( G \), called the right child of the node

Binary tree class skeleton

```java
class BINARY_TREE[G] feature
    item: G
    left: BINARY_TREE[G]
    right: BINARY_TREE[G]

    ... Insertion and deletion features ...
end
```

A recursive routine on a recursive data structure

```java
count: INTEGER is
    -- 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
```

Children and parents

- Parent
- Left child
- Right child

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

More binary tree properties and terminology

A node of a binary tree may have:
- Both a left child and a right child
- Only a left child
- Only a right child
- No child
More properties and terminology

Upward path:
Sequence of zero or more nodes, where any node in the sequence is the parent of the previous one if any.

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

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.

Height of a binary tree

Height:
Maximum numbers of nodes on a downward path from the root to a leaf.

\[
\text{height INTEGER = } \quad \text{Maximum number of nodes on a downward path.}
\]
\[
\text{local}
\]
\[
\text{lh, rh: INTEGER}
\]
\[
\text{do}
\]
\[
\text{if left IS Void then lh := left.height end}
\]
\[
\text{if right IS Void then rh := right.height end}
\]
\[
\text{Result := 1 + lh.max (rh)}
\]

Binary tree operations

add_left (x: a) IS
require
\[
\text{no_left_child, behind: left : Void}
\]
do
\[
\text{create left.make (x)}
\]
end

add_right (x: a): Same model.

make (x: a) IS
require
\[
\text{item IS x}
\]
do
\[
\text{item := x}
\]
end

Binary tree traversals

print_all IS
require
\[
\text{print all node values.}
\]
do
\[
\text{if left IS Void then left.print_all end}
\]
\[
\text{print (item)}
\]
\[
\text{if right IS Void then right.print_all end}
\]
end

Binary tree traversals

Inorder:

\[
\text{traverse left subtree visit root, traverse right subtree}
\]

Preorder:

\[
\text{visit root, traverse left, traverse right}
\]

Postorder:

\[
\text{traverse left, traverse right visit root}
\]

Binary search tree

A binary tree has a left subtree and a right subtree

A binary tree over a sorted set S is a binary search tree if for every node \( n \):

- For every node \( x \) of the left subtree of \( n \), \( x.item \leq n.item \)
- For every node \( x \) of the right subtree of \( n \), \( x.item \geq n.item \)
Printing elements in order

class BINARY_SEARCH_TREE [G] feature
  item: G
  left, right: BINARY_SEARCH_TREE [G]

  sort is
    -- Print element values in order
    do
      if left /= Void then left.sort end
      print(item)
      if right /= Void then right.sort end
    end
end

Searching in a binary search tree

class BINARY_SEARCH_TREE [G] feature
  item: G
  left, right: BINARY_SEARCH_TREE [G]
  has(x: G): BOOLEAN is
    -- Does x appear in any node?
    require
      argument_exists: x /= Void
    do
      if x = item then
        Result := True
      elseif x < item and left /= Void then
        Result := left.has(x)
      elseif x > item and right /= Void then
        Result := right.has(x)
      end
    end
end

Insertion into a binary search tree

Do it as an exercise!

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).

Why binary search trees?

Linear structures: insertion, search and deletion are \( O(n) \)

Binary search tree: average behavior for insertion, deletion and search is \( O(\log(n)) \)

But: worst-time behavior is \( O(n) \)!

Note measures of complexity: best case, average, worst case.

What we have seen so far

A definition is recursive if it takes advantage of the notion itself, on a smaller target
What can be recursive: a routine, the definition of a concept...

Still some mystery left: isn't there a danger of a cyclic definition?
Recursion variant

Every recursive routine should use a 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 \( 0 \leq v' < v \)

Hanoi: what is the variant?

```haskell
hanoi(n: INTEGER, source, target, other: CHARACTER) is
  -- Transfer \( n \) disks from source to target,
  -- using other as intermediate storage.
  require
  --
  do
    if \( n > 0 \) then
      hanoi(n-1, source, other, target)
      move(source, target)
      hanoi(n-1, other, target, source)
    end
end
```

Printing: what is the variant?

```haskell
class BINARY_SEARCH_TREE(G) feature
  item %
  left, right BINARY_SEARCH_TREE(G)

  sort is
    -- Print element values in order
    do
      if left /= Void then left.sort end
      print(item)
      if right /= Void then right.sort end
    end
end
```

Contracts for recursive routines

```haskell
hanoi(n: INTEGER, source, target, other: CHARACTER) is
  -- Transfer \( n \) disks from source to target,
  -- using other as intermediate storage.
  -- variant: \( n \)
  -- invariant: disks on each needle are piled in
  -- decreasing size
  require
  --
  do
    if \( n > 0 \) then
      hanoi(n-1, source, other, target)
      move(source, target)
      hanoi(n-1, other, target, source)
    end
  end
```

McCarthy’s 91 function

```plaintext
M(n) =
  \( n = 10 \) if \( n > 100 \)
  \( M(M(n + 1)) \) if \( n \leq 100 \)
```

Another function

```plaintext
bizarre(n) =
  1 if \( n = 1 \)
  \( bizarre(n / 2) \) if \( n \) is even
  \( bizarre((3 \times n + 1) / 2) \) if \( n > 1 \) and \( n \) is odd
```
Fibonacci numbers

\[ \text{fib} (1) = 0 \]
\[ \text{fib} (2) = 1 \]
\[ \text{fib} (n) = \text{fib} (n-2) + \text{fib} (n-1) \quad \text{for } n > 2 \]

Factorial function

\[ 0! = 1 \]
\[ n! = n \cdot (n-1)! \quad \text{for } n > 0 \]

Recursive definition is interesting for demonstration purposes only; practical implementation will use loop (or table)

Our original example of a loop

\[ \text{highest_name: STRING is} \quad \quad \text{-- Alphabetically greatest station name of line f} \]
\[ \text{do} \quad \quad \text{-- line f starting at current cursor position} \]
\[ \text{from} \quad \quad \text{require} \]
\[ \text{f.start Result >= } \text{"} \]
\[ \text{until} \quad \quad \text{not f.is_empty} \]
\[ \text{loop} \quad \quad \text{do} \]
\[ \text{Result := greater(Result, f.item.name)} \quad \quad \text{f.forth} \]
\[ \text{end} \quad \quad \text{after} \]

A recursive equivalent

\[ \text{highest_name: STRING is} \quad \quad \text{-- Alphabetically greatest station name} \]
\[ \quad \quad \text{-- of line f} \]
\[ \quad \quad \text{require} \]
\[ \quad \quad \text{not f.is_empty} \]
\[ \quad \quad \text{do} \]
\[ \quad \quad \text{f.start} \]
\[ \quad \quad \text{Result := f.highest_from_cursor} \]
\[ \quad \quad \text{end} \]

Auxiliary function for recursion

\[ \text{highest_from_cursor: STRING is} \quad \quad \text{-- Alphabetically greatest name of stations} \]
\[ \quad \quad \text{of} \]
\[ \quad \quad \text{require} \]
\[ \quad \quad \text{f /= Void; not f.off} \]
\[ \quad \quad \text{do} \]
\[ \quad \quad \text{Result := f.item.name} \quad \quad \text{f.forth} \]
\[ \quad \quad \text{if not f.after then} \]
\[ \quad \quad \text{Result := greater(Result, Result)} \quad \quad \text{f.back} \]
\[ \quad \quad \text{end} \]
\[ \quad \quad \text{end} \]

Loop version using arguments

\[ \text{maximum(a ARRAY [STRING]): STRING is} \quad \quad \text{-- Alphabetically greatest item in a} \]
\[ \quad \quad \text{require} \]
\[ \quad \quad \text{a.count = 1} \]
\[ \quad \quad \text{local} \]
\[ \quad \quad \text{i INTEGER} \]
\[ \quad \quad \text{do} \]
\[ \quad \quad \text{from} \quad \quad \text{do} \]
\[ \quad \quad \text{i := a.lower + 1; Result := a.item (a.lower)} \quad \quad \text{invariant} \]
\[ \quad \quad \text{i := a.lower; i := a.upper + 1} \]
\[ \quad \quad \text{-- Result is the maximum element of a[a.lower..i - 1]} \]
\[ \quad \quad \text{until} \quad \quad \text{if a.upper} \]
\[ \quad \quad \text{loop} \quad \quad \text{if a.item (i) > Result then Result := a.item (i) end} \]
\[ \quad \quad \text{i := i + 1} \]
\[ \quad \quad \text{end} \]
\[ \quad \quad \text{end} \]
### Recursive version

```haskell
maxrec (a, ARRAY, STRING), STRING is
  require
    a.count = 1
  do
    Result := max_sub_array(a, a.lower)
  end

max_sub_array(a, ARRAY, STRING, INTEGER, STRING is
  require
    i := a.lower, i := a.upper
  do
    Result := a.item(i)
    if i > a.upper then
      Result := greater(Result, max_sub_array(a, i + 1))
    end
  end
```

### Recursion elimination

Recursive calls cause (in a default implementation without optimization) a run-time penalty: need to maintain stack of preserved values.

Various optimizations are possible.

Sometimes a recursive scheme can be replaced by a loop; this is known as recursion elimination.

"Tail recursion" (last instruction of routine is recursive call) can usually be eliminated.

### Recursion elimination

```haskell
r(x) is
  x := x^'
  get start of r

... Some instructions ...

May need x!
```

... e.g. \( r(x-1) \)

... More instructions ...

After call, need to revert to previous values of arguments and other context information.

### Using a stack

Queries:
- Is the stack empty? `is_empty`
- Top element, if any: `item`

Commands:
- Push an element on top: `put`
- Pop top element, if any: `remove`

### Recursion as a problem-solving technique

Applicable if you have a way to construct a solution to the problem, for a certain input set, from solutions for one or more smaller input sets.

### What we have seen
What we have seen

- The notion of recursive definition
- Applications of recursion
- The anatomy of a recursive algorithm: the Tower of Hanoi
- What makes a recursive definition "well-behaved"
- Binary trees
- Binary search trees
- A glimpse of recursion implementation