Introduction to Programming

Bertrand Meyer

Last revised 15 December 2003
Lecture 16: Introduction to recursion
Asia-Pacific Software Engineering Conference
Lecture 16: Introduction to recursion
The story of the universe*

Dans le grand temple de Bénarès, 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.

A 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 Brahmâ.

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 Brahmâ est simple: un seul disque à la fois et jamais un grand plateau sur un plus petit.

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

*According to Édouard Lucas, Récréations mathématiques, Paris, 1883
The story of the universe*

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

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 Brahmâ.

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

The rule of Brahmâ 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.
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:

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

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

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

\[
H_n = 2^n - 1
\]
move \( (n: INTEGER; \\
source, target, other: CHARACTER) \) \textbf{is} \textbf{if} \ n \ > \ 0 \ \textbf{then} \textbf{do} \\
\textbf{end} \\
\textbf{end} \\
\textbf{-- Move} \ n \ \textbf{disks} \ \textbf{from} \ \textbf{needle} \ \textbf{source} \ \textbf{to} \ \textbf{target}, \\
\textbf{-- using} \ \textbf{other} \ \textbf{as} \ \textbf{storage}.
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 \textit{move} 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 $r_1$. 
Recursive grammar

Instruction = Assignment | Conditional | Compound | ...

Conditional = if Expression then Instruction
else Instruction end
Defining lexicographical order

Problem: define the notion that word $w_1$ is “before” word $w_2$, 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 \textit{w1} is “before” the word \textit{w2} if and only if one of the following conditions holds:

- \textit{w1} is empty and \textit{w2} is not empty

- Neither \textit{w1} nor \textit{w2} is empty, and the first letter of \textit{w1} is smaller than the first letter of \textit{w2}.

- Neither \textit{w1} nor \textit{w2} is empty, their first letters are the same, and the word obtained by removing the first letter of \textit{w1} is (recursively) before the word obtained by removing the first letter of \textit{w2}. 
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
class \texttt{BINARY\_TREE} [G] \texttt{feature}

\texttt{item}: G

\texttt{left}: \texttt{BINARY\_TREE} [G]
\texttt{right}: \texttt{BINARY\_TREE} [G]

... Insertion and deletion features ...

\texttt{end}
A binary tree has a **left subtree** and a **right subtree**

A binary tree over a sorted set $G$ 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 $n$, $x.item \geq n.item$
class BINARY_SEARCH_TREE [G ...] feature
  item: G
  left, right: BINARY_SEARCH_TREE [G]
  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
class BINARY_SEARCH_TREE [G ...] feature
  item: G
  left, right: BINARY_SEARCH_TREE [G]
  right: BINARY_SEARCH_TREE [G]

  has (x: G): BOOLEAN is
   -- Does a node in this subtree have value x?
   do
     if x < item then
       Result := ((left /= Void) and then left.has (x))
     elseif x > item then
       Result := (right /= Void) and then right.has (x)
     else
       Result := True
     end
   end
end
end
Insertion into a binary search tree

- Do it as an exercise!
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.
General soundness properties

For a recursive definition to make sense:

- There must be a non-recursive branch!

- Some non-negative number — the variant of the recursion — must decrease in each call.
Hanoi: what is the variant?

move (n: INTEGER;
       source, target, other: CHARACTER) is
    -- Move n disks from needle source to target,
    -- using other as storage.
    do
      if n > 0 then
        move (n−1, source, other, target)
        transfer (source, target)
        move (n−1, other, target, source)
      end
    end
class BINARY_SEARCH_TREE [G ...] feature
  item: G
  left, right: BINARY_SEARCH_TREE [G]
  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
McCarthy’s 91 function

\[ M(n) = \]

- \( n - 10 \) \quad \text{if} \quad n > 100
- \( M(M(n + 11)) \) \quad \text{if} \quad n \leq 100
Another function

\[ \text{bizarre} \ (n) = \]

- 1 \quad \text{if} \quad n = 1

- \text{bizarre} \ (n / 2) \quad \text{if} \quad n \text{ is even}

- \text{bizarre} \ ((3 * n + 1) / 2) \quad \text{if} \quad n > 1 \ \text{and} \ n \text{ is odd}
Fibonacci numbers

- $\text{fib} (1) = 0$
- $\text{fib} (2) = 1$

- $\text{fib} (n) = \text{fib} (n-2) + \text{fib} (n-1)$ for $n > 2$
Factorial function

- $0! = 1$

- $n! = n \times (n - 1)!$ for $n > 0$

Recursive definition is interesting for demonstration purposes only; practical implementation will use loop (or table)
Our original example of a loop

\[
\begin{align*}
\text{highest\_name: STRING is} \\
\quad \text{-- Alphabetically greatest station name of line } f \\
\quad \text{do} \\
\quad \quad \text{from} \\
\quad \quad \quad f.\text{start} \; ; \; \text{Result} \; := \; "" \\
\quad \quad \text{until} \\
\quad \quad \quad f.\text{after} \\
\quad \text{loop} \\
\quad \quad \quad \text{Result} \; := \; \text{greater} \,(\text{Result, f.item.name}) \\
\quad \quad \quad f.\text{forth} \\
\quad \text{end} \\
\text{end}
\end{align*}
\]
A recursive equivalent

\[ \text{highest\_name: STRING is} \]

\[
\quad -- \text{Alphabetically greatest station name} \\
\quad -- \text{of line } f \\
\]

\[ \text{require} \]

\[ \text{not } f.\text{is\_empty} \]

\[ \text{do} \]

\[ f.\text{start} \]

\[ \text{Result} := f.\text{highest\_from\_cursor} \]

\[ \text{end} \]
Auxiliary function for recursion

highest_from_cursor: STRING is
    -- Alphabetically greatest name of stations of
    -- line f starting at current cursor position

require
    f /= Void; not f.off

do
    Result := f.item.name
    f.forth
    if not f.after then
        Result := greater (Result, highest_from_cursor)
    end
end
Loop version using arguments

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

maxrec \(a: \text{ARRAY [STRING]}\): \text{STRING} \text{ is}
\[\text{-- Alphabetically greatest item in } a\]
\[
\text{require}\]
\[
\quad a.\text{count} \geq 1
\]
\[
\text{do}
\]
\[
\quad \text{Result} := \text{max_sub_array} (a, a.\text{lower})
\]
\[
\text{end}
\]

max_sub_array \(a: \text{ARRAY [STRING]}; i: \text{INTEGER}\): \text{STRING} \text{ is}
\[\text{-- Alphabetically greatest item in } a \text{ starting from index } i\]
\[
\text{require}\]
\[
\quad i \geq a.\text{lower} ; i \leq a.\text{upper}
\]
\[
\text{do}
\]
\[
\quad \text{Result} := a.\text{item} (i)
\]
\[
\quad \text{if } i < a.\text{upper} \text{ then}
\]
\[
\quad \quad \text{Result} := \text{greater} (\text{Result}, \text{max_sub_array} (a, i + 1))
\]
\[
\text{end}
\]
\[
\text{end}
\]
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 recursion 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

\[ r(x) \text{ is } \]
\[ \text{do} \]
\[ x := x' \]
\[ \text{goto} \text{ start\_of\_r} \]
\[ \text{end} \]

May need \( x \)!

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.
With thanks to...

- Fromageries Bel (*La Vache qui rit*)
- Yao tribes
- Swiss
- Various other elephants
- The Thai government
End of lecture 16