RECURSION

This page is intentionally blank

☺

ITERATION

- Suppose *fact n* represents the *factorial* function applied to a value *n*

- This means that

$$fact\ n = 1 \times 2 \times 3 \times 4 \times 5 \times .... \times n$$

- An algorithm to determine *fact n* might be, for a given value of *n*:

```
set Result to 1
set multiplier to 1
loop for each integer from 1 to n
      multiply Result by multiplier
      increment multiplier by 1
end_loop
output Result
```

RECURSION

- We have

  $fact\ n = 1 \times 2 \times 3 \times 4 \times 5 \times .... \times n$

  i.e. $fact\ n = 1 \times 2 \times 3 \times 4 \times 5 \times ....\times (n\text{-}1) \times n$

  Hence: $fact\ n = n \times fact\ (n\text{-}1)$ $(n > 1)$

- The latter specification of *fact n* is *recursive*: the function *fact* is defined in terms of itself

- The corresponding recursive algorithm might be, in *Eiffel*-like code:

```
fact (n : INTEGER) : INTEGER is
        - - computes factorial n for any integer n where n > 0
do
        if n = 1 then
                Result := 1
        else
                Result := n * fact(n-1)
        end - - if
end - - fact
```

RECURSION

- Another simple example of a naturally recursive process is reversing the sequence of letters in a word (e.g. producing *rats* from *star*)

- A suitable simple algorithm might be based upon:

remove first letter of word
reverse (*rest-of-word*)
append removed letter

- Allowing for termination, we might have:

```
reverse (word : STRING) : STRING is
    do
        if word has only one letter then
            append word to Result
        else
            remove first letter of word
            reverse (tail of word)
            append removed letter to Result
        end_if
    end - - reverse
```

RECURSION

- Recursion is effectively iteration in another guise: for every recursive algorithm, there is an alternative iterative version

- When compared with iterative alternatives, recursive algorithms are frequently:

  - more concise
  - easier to derive and understand

- To see this, consider the famous *Towers of Hanoi* problem:

  According to legend, a special order of Buddhist monks in Hanoi, has been charged with the following task. The monks have available three vertical golden rods and a set of 64 circular jade discs where no two discs have the same diameter. Each disc has a hole in the centre which allows it to be slid onto any of the golden rods. Initially, all the discs were on a single rod, one on top of the other in order of decreasing size with the largest on the bottom. The task that the monks have to accomplish is to transfer this original tower of discs onto one of the other rods moving only one disc at a time, using the third rod where necessary, and ensuring that no disc is ever placed on top of a smaller disc. When the complete tower of discs has been transferred, the world will end in a clap of thunder.

RECURSION

- A recursive solution to the *Towers of Hanoi* problem may be developed "naturally":
    - Suppose the tower is to move from a rod designated *source* to a rod designated *target*
    - Suppose *other* designates the third rod
    - It should be clear that to solve the problem, the largest disc must form the base of the new tower; but for this to happen, the 63 discs above it must be on the *other* rod
    - The problem constraints demand that those 63 discs on *other* are in a tower of decreasing size (bottom to top)

RECURSION

- The previous considerations lead, naturally, to the following algorithm for transferring the tower of 64 discs from *source* to *target*:

  transfer the top 63 discs from *source* to *other*
  move the bottom disc from *source* to *target*
  transfer the 63 discs from *other* to *target*

- Generalizing the algorithm for moving *n* discs:

  transfer the top *n*-1 discs from *source* to *other*
  move 1 disc from *source* to *target*
  transfer the *n*-1 discs from *other* to *target*

- Hence, we could write:

```
move_tower (n, source, target, other) is
        - - move n discs from source rod to target rod
        do
                if n = 1 then
                        move disc from source to target
                else
                        move_tower (n-1, source, other, target)
                        move 1 disc from source to target
                        move_tower (n-1, other, target, source)
                end - - if
        end - - move_tower
```

RECURSION IN FORMAL SPECIFICATION

- Suppose *del_first_occ* (*x*, *s*) is a function which returns the sequence derived by deleting from the sequence *s* the first occurrence of *x*, if any

- Given a function *first_pos_in* (*v*, *s*) which returns either
  - the index of the first occurrence of *v* in the sequence *s*, or
  - 0, if *v* is not present in s

  then a definition of *del_first_occ* (*x*, *s*) is:

$$[X]$$

$$del\_first\_occ : X \times \mathbf{seq}\ X \rightarrow \mathbf{seq}\ X$$

$$\forall x : X; s : \mathbf{seq}\ X \bullet$$
$$del\_first\_occ\ (x, s) = 1..s \setminus \{first\_pos\_in\ (x, s)\} \upharpoonleft s$$

- Alternatively, enumerating all possible *cases* in the predicate description avoids reliance on the extra function *first_pos_in* (see next)

RECURSION IN FORMAL SPECIFICATION

- Using this "case" style *del_first_occ* might be defined by:

$$
\begin{array}{|l}
\hline
[X] \\
del\_first\_occ : X \times \textbf{seq}\ X \rightarrow \textbf{seq}\ X \\
\hline
\forall\, x : X; s : \textbf{seq}\ X \bullet \\
\quad s = \langle\,\rangle \Rightarrow del\_first\_occ\ (x, s) = \langle\,\rangle \\
\forall\, x : X; s : \textbf{seq}_1\ X \bullet \\
\quad x = \textbf{head}\ s \Rightarrow del\_first\_occ\ (x, s) = \textbf{tail}\ s \quad \wedge \\
\quad x \neq \textbf{head}\ s \Rightarrow del\_first\_occ\ (x, s) = \\
\qquad\qquad \langle \textbf{head}\ s \rangle \frown del\_first\_occ\ (x, \textbf{tail}\ s) \\
\hline
\end{array}
$$

- Observe the *recursive* nature of part of the predicate description

- When using the case style of definition it is imperative that the cases cited must completely *close off* the domain of the function (i.e every object in the intended domain, but no other, must be catered for by one of the cases)

RECURSION IN FORMAL SPECIFICATION

- Note that the case style requires the inclusion of one or more axioms of the form:

$$\forall \text{ Declarations\_of } x_1, x_2, \ldots, x_k \bullet$$
$$\text{Predicate\_involving\_some\_of } x_1, x_2, \ldots, x_k \Rightarrow$$
$$\text{Function\_name } (x_1, x_2, \ldots, x_k) = \text{Term\_involving } x_1, x_2, \ldots, x_k$$

- The antecedent of the implication acts as a case-defining *guard* in the sense that the equality applies only if the antecedent is true

- It is important that the guards are always sufficiently strong to define disjoint cases

  - An example of inconsistency is:

$$k \leq 0 \Rightarrow f(k) = 0 \quad \wedge$$
$$k \geq 0 \Rightarrow f(k) = 1$$

What is the problem with this?

RECURSION IN FORMAL SPECIFICATION

- A related form of definition involves *enumerating* different argument structures or "patterns" for a function

- Instead of writing
  - $s = \langle\rangle \Rightarrow f(s \ldots) = \ldots \ldots$

    we could write, more concisely:
  - $f(\langle\rangle \ldots) = \ldots \ldots$

- The guarding antecedent has now shifted into the argument structure

- The axiom for *del_first_occ* then becomes:

$$\forall x : \mathrm{X} \bullet del\_first\_occ(x, \langle\rangle) = \langle\rangle$$

$$\forall x : \mathrm{X}; s : \mathbf{seq}\ \mathrm{X} \bullet del\_first\_occ(x, \langle x\rangle \frown s) = s$$

$$\forall x,y : \mathrm{X}; s : \mathbf{seq}\ \mathrm{X} \mid x \neq y \bullet del\_first\_occ(x, \langle y\rangle \frown s) = y \frown del\_first\_occ(x, s)$$

EXERCISES

1. Convince yourself of the correctness of the recursive algorithm given for reversing the letters of a word by changing *rats* to *star*.

2. Convince yourself of the correctness of the recursive algorithm given for the *Towers of Hanoi* problem by showing that it works for 3 discs.

3. A list can be sorted by splitting it into two approximately equal halves, sorting each half separately, and then merging the sorted halves together. Use this idea to devise a recursive algorithm for sorting a list and demonstrate the correctness of the algorithm by using it to sort a list of, say, seven names.

4. Devise algorithms to detect whether a given target element is present in a given list

   (a) if the list is *not* sorted into sequence
   (b) if the list is sorted into sequence

5. Consider the given generic definition of a function *delete_all* (x, s)which yields the sequence obtained by deleting from sequence s *all* instances of x:

$$[X]$$
$$delete\_all : X \times \mathbf{seq}\ X \rightarrow \mathbf{seq}\ X$$
$$\forall x : X; s : \mathbf{seq}\ X \bullet delete\_all(x, s) = s \upharpoonright (\mathbf{ran}\ s \setminus \{x\})$$

   (a) Provide similar generic definitions for functions:

      (i) *elems_of* (s) which yields the elements of sequence s as a set;
      (ii) *count_occs*(x, s) which counts how many times x occurs in a sequence s.

   (b) Recast, using *argument structures*, the definitions of the two functions referred to in (a) above.

   (c) Recast, using *lambda-expressions*, the definitions of all three functions referred to above.

/question continues overleaf

(d) Recast the definitions of the function *count_occs* into the equivalent **curried** versions, *curried_count_occs*

(i) as a generic form similar to the given example, and
(ii) using lambda-expressions.

6. A function *replicate,* when given input arguments n (a natural number) and s (a sequence), yields another sequence by replicating the original sequence s, n times.

e.g. *replicate* 3 ⟨p, q⟩ = ⟨p, q, p, q, p, q⟩

(a) Create a "case" style generic **curried** definition for *replicate*

(b) By considering the function for *replicate* defined above, deduce a suitable definition for a function *replicate_10_times* that replicates **any** sequence 10 times.

7. An axiomatic definition for the factorial function, *fact*, might be:

$$fact : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall n : \mathbb{N} \bullet$$
$$n = 0 \Rightarrow fact\ n = 1 \quad \wedge$$
$$n \neq 0 \Rightarrow fact\ n = n * fact\ (n - 1)$$

(a) If *sigma* is a function which, given an integer n, where $n \geq 0$, returns the sum of the integers from 0 to n (i.e. *sigma* $n = 0+1+2+3+.....+(n-1)+n$), create a similar axiomatic definition for *sigma*.

(b) If *power* is a function which, given integers x and n, where $x \neq 0$ and $n \geq 0$, returns the value of x raised to the power n, complete the following axiomatic definition for *power*:

$$power : \mathbb{N}_1 \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall x : \mathbb{N}_1 ;\ n : \mathbb{N} \bullet$$
$$n = 0 \Rightarrow power\ (x, n) = 1 \quad \wedge$$
$$n \neq 0 \Rightarrow power\ (x, n) =$$