Recursivity and ADT

Thierry Lecroq

University of Rouen
FRANCE
Outline

1. Recursivity
2. Linked lists
3. Abstract Data Types
Plan

1. Recursivity

2. Linked Lists

3. Les types de données abstraits
Recursivity

Informal Definition

An object is said to be recursive if it is used in its definition or its composition.

Example

A Russian nested doll is a doll that contains a Russian nested doll.
Recursivity can be used for defining mathematical objects such as natural numbers.

Example

0 is a natural number.
The successor of a natural number is a natural number.
Recursivity

Every recursive definition must contain a stop condition.

Example
A Russian nested doll is a doll that contains a Russian nested doll or a filled doll.
A fonction is said to be recursive if its definition contains a call to itself. The stop condition enables then to stop the recursive calls thus avoiding the program to execute indefinitely.

Most programming languages enable to use recursive sub-routines.
Factorial Computation

\[ n! = 1 \times 2 \times \cdots \times n - 1 \times n = \prod_{i=1}^{n} i \]

0! = 1

Iterative Function

function factIter(n : integer) : integer
auxiliary variables \( i, f \) : integers
begin
    \( f \leftarrow 1 \)
    \( i \leftarrow 0 \) \( \{ f = i! \} \)
    while \( i < n \) do \( \{ f = i! \text{ and } i < n \} \)
        \( i \leftarrow i + 1 \) \( \{ f = (i - 1)! \text{ and } i \leq n \} \)
        \( f \leftarrow f \times i \) \( \{ f = i! \text{ and } i \leq n \} \)
    endwhile \( \{ f = i! \text{ and } i = n \} \)
    return \( f \)
end
Factorial Computation

Example

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>24</td>
<td>120</td>
</tr>
</tbody>
</table>

Complexity

The while loop executes $n$ times, the complexity is thus in the order of $n$. 
Factorial Computation

$0! = 1$

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

Recursive Function

function factRec(n : integer) : integer
begin
    if $n = 0$ then
        return 1
    else
        return $n \times \text{factRec}(n - 1)$
    endif
end
### Memory allocated to a process

<table>
<thead>
<tr>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code area (instructions)</td>
</tr>
<tr>
<td>Static data area</td>
</tr>
<tr>
<td>Execution stack</td>
</tr>
<tr>
<td>Heap (dynamic data)</td>
</tr>
</tbody>
</table>
**Factorial Computation**

**Example**

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static data area</td>
</tr>
<tr>
<td>factRec($n = 5$)</td>
</tr>
<tr>
<td>Heap (dynamic data)</td>
</tr>
</tbody>
</table>
## Factorial Computation

### Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
</tr>
</thead>
<tbody>
<tr>
<td>factRec((n = 5))</td>
<td>5 × factRec(4)</td>
</tr>
</tbody>
</table>

Heap (dynamic data)
Factorial Computation

Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
<th>Heap (dynamic data)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>factRec($n=5$)</td>
<td>factRec($n=4$)</td>
</tr>
<tr>
<td></td>
<td>$5 \times \text{factRec}(4)$</td>
<td>$4 \times \text{factRec}(3)$</td>
</tr>
</tbody>
</table>
## Factorial Computation

### Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
</tr>
</thead>
<tbody>
<tr>
<td>factRec($n = 5$)</td>
<td>5 \times factRec(4)</td>
</tr>
<tr>
<td>factRec($n = 4$)</td>
<td>4 \times factRec(3)</td>
</tr>
<tr>
<td>factRec($n = 3$)</td>
<td>3 \times factRec(2)</td>
</tr>
</tbody>
</table>

Heap (dynamic data)
## Factorial Computation

### Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Heap (dynamic data)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static data area</strong></td>
<td></td>
</tr>
<tr>
<td>factRec(n = 5)</td>
<td>factRec(n = 0)</td>
</tr>
<tr>
<td>5 × factRec(4)</td>
<td></td>
</tr>
<tr>
<td>factRec(n = 4)</td>
<td></td>
</tr>
<tr>
<td>4 × factRec(3)</td>
<td></td>
</tr>
<tr>
<td>factRec(n = 3)</td>
<td></td>
</tr>
<tr>
<td>3 × factRec(2)</td>
<td></td>
</tr>
<tr>
<td>factRec(n = 2)</td>
<td></td>
</tr>
<tr>
<td>2 × factRec(1)</td>
<td></td>
</tr>
<tr>
<td>factRec(n = 1)</td>
<td></td>
</tr>
<tr>
<td>1 × factRec(0)</td>
<td></td>
</tr>
<tr>
<td>factRec(n = 0)</td>
<td></td>
</tr>
</tbody>
</table>

Thierry Lecroq (Univ. Rouen)

Recursivity and ADT
Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
</tr>
</thead>
<tbody>
<tr>
<td>factRec( (n = 5) )</td>
<td>5 × factRec(4)</td>
</tr>
<tr>
<td>factRec( (n = 4) )</td>
<td>4 × factRec(3)</td>
</tr>
<tr>
<td>factRec( (n = 3) )</td>
<td>3 × factRec(2)</td>
</tr>
<tr>
<td>factRec( (n = 2) )</td>
<td>2 × factRec(1)</td>
</tr>
<tr>
<td>factRec( (n = 1) )</td>
<td>1 × factRec(0)</td>
</tr>
<tr>
<td>factRec( (n = 0) )</td>
<td>← 1</td>
</tr>
</tbody>
</table>

Heap (dynamic data)
## Factorial Computation

### Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
</tr>
</thead>
<tbody>
<tr>
<td>factRec(n = 5)</td>
<td>5 × factRec(4)</td>
</tr>
<tr>
<td>factRec(n = 4)</td>
<td>4 × factRec(3)</td>
</tr>
<tr>
<td>factRec(n = 3)</td>
<td>3 × factRec(2)</td>
</tr>
<tr>
<td>factRec(n = 2)</td>
<td>2 × factRec(1)</td>
</tr>
<tr>
<td>factRec(n = 1)</td>
<td>1 × 1</td>
</tr>
</tbody>
</table>

Heap (dynamic data)
## Factorial Computation

### Example

<table>
<thead>
<tr>
<th>Code area (instructions)</th>
<th>Static data area</th>
</tr>
</thead>
<tbody>
<tr>
<td>factRec($n = 5$)</td>
<td>5 × factRec(4)</td>
</tr>
<tr>
<td>factRec($n = 4$)</td>
<td>4 × factRec(3)</td>
</tr>
<tr>
<td>factRec($n = 3$)</td>
<td>3 × factRec(2)</td>
</tr>
<tr>
<td>factRec($n = 2$)</td>
<td>2 × factRec(1)</td>
</tr>
<tr>
<td>factRec($n = 1$)</td>
<td>← 1</td>
</tr>
</tbody>
</table>

Heap (dynamic data)
Factorial Computation

Proof

- \( n = 0 \): \( \text{factRec}(0) = 1 = 0! \)
- RH: let us assume that \( \text{factRec}(k) \) computes \( k! \) \( \forall k < n \)
- \( n \): \( \text{factRec}(n) = n \times \text{factRec}(n - 1) =_{\text{RH}} n \times (n - 1)! = n! \)
Factorial Computation

Complexity

The complexity can be computing as the number of calls. For $n = 5$ there is 1 initial call and 5 recursive calls.

Proposition

The number of recursive calls of $\text{factRec}(n)$ is $n$.

Proof

By recurrence on $n$

- $n = 0$: no recursive call
- RH: $\forall k < n$ the number of recursive calls is $k$
- $n$: number of recursive calls = 1 call to $\text{factRec}(n - 1)$ + number of recursive calls of $\text{factRec}(n - 1)$ $=_{\text{RH}} 1 + n - 1 = n$

Overall the number of calls is equal to 1 initial call + $n$ recursive calls = $n + 1$ calls
Fibonacci Numbers

\[ F_0 = 0 \]
\[ F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2} \quad \forall n > 1 \]
Fibonacci Numbers

Iterative Function

function fiboIter(n : integer) : integer
auxiliary variables e, f, i : integers
begin
    if n = 0 then
        return 0
    else
        e ← 0
        f ← 1
        i ← 1 \{f = F_i\}
        while i < n do \{e = F_{i-1}, f = F_i and i < n\}
            f ← f + e \{e = F_{i-1}, f = F_{i+1} and i < n\}
            e ← f - e \{e = F_i, f = F_{i+1} and i < n\}
            i ← i + 1 \{e = F_{i-1}, f = F_i and i ≤ n\}
        endwhile \{e = F_{i-1}, f = F_i and i = n\}
        return f
    endif
end
Fibonacci Numbers

Recursive Function

function fiboRec(n : integer) : integer
begin
    if n < 2 alors
        return n
    else
        return fiboRec(n − 1) + fiboRec(n − 2)
    endif
end
Recursivity

Advantages

Once a problem is represented in a recursive manner, the construction of a recursive algorithm is immediate. Proofs of recursive programs done by recurrence are also very simple.

Drawbacks

Recursivity requires space for storing the current operations.
1. Recursivity

2. Linked Lists

3. Les types de données abstraits
Linked Lists

Let $D$ be a set of values (domain).
A linked list with values in $D$ is a sequential structure that enables to represent any sequence on $D$.
It is composed of:

- a pointer, called “head of list” giving access to the first cell of the list;
- a sequence of cells, each of them giving access to the next, except the last.

Each cell has 2 components:

- a component “value” on $D$;
- a component “pointer”.

The sequence on $D$ represented by a linked list is the sequence of components “values” taken from the head of the list.

Example

\[ \ell_1 = \langle e_1, e_3, e_4 \rangle \]
\[ \ell_2 = \langle e_2, e_5 \rangle \]
\[ \ell_3 = <> \]

The empty sequence is represented by a head pointer equal to NIL (NULL in C, $\times$ on the pictures). The component “pointer” of the last cell of a non-empty list is equal to NIL.
Linked Lists

Manipulation

type
    List = structure
        element : $D$
        next : pointer to List
    endstructure
    Plist = pointer to List

function emptyList() : PList
begin
    return NIL
end

function getNext(ℓ : Plist) : Plist
Pre-condition : ℓ is non-empty
begin
    return ↑ ℓ.next
end
Manipulation

procedure setNext(ℓ, p : Plist)
Pre-condition: ℓ is non-empty
begin
    ↑ ℓ.next ← p
end

function getElement(ℓ : Plist) : D
Pre-condition: ℓ is non-empty
begin
    return ↑ ℓ.element
end

procedure setElement(ℓ : Plist, x : D)
Pre-condition: ℓ is non-empty
début
    ↑ ℓ.element ← x
fin
Linked Lists

Manipulation

function isEmptyList(ℓ : Plist) : boolean
begin
    return ℓ = NIL
end

function addAtTheBeginning(ℓ : Plist, x : D) : Plist
auxiliary variable p : Plist
begin
    p ← allocate
    setElement(p, x)
    setNext(p, ℓ)
    return p
end
Linked Lists

Programming

```c
struct List {
    D element;
    struct List * next;
};
typedef struct List * PList;

PList emptyList() {
    return NULL;
}

PList getNext(PList ℓ) {
    // Pre-condition : ℓ is non-empty
    return ℓ → next;
}
```
void setNext(PList ℓ, PList p) {
  // Pre-condition : ℓ is non-empty
  ℓ →next = p;
}

D getElement(PList ℓ) {
  // Pre-condition : ℓ is non-empty
  return ℓ →element;
}

void setElement(PList ℓ, D x) {
  //Pre-condition : ℓ is non-empty
  ℓ →element = x;
}
Programming

```c
int isEmptyList(PList ℓ) {
    return ℓ == NULL;
}

PList addAtTheBeginning(PList ℓ, D x) {
    Plist p;

    p = (PList)malloc(sizeof(struct List));
    if (p == NULL) exit(1);
    setElement(p, x);
    setNext(p, ℓ);
    return p;
}
```
Linked Lists

### Printing
In order to print the values of a list, it must be scanned from the first cell to the last one.

### Length
Same as printing

### Reverse printing
Application of the recursivity
function directScan(ℓ : PList)
begin
    while not isEmptyList(ℓ) do
        print(getElement(ℓ))
        ℓ ← getNext(ℓ)
    endwhile
end

function length(ℓ : PList) : integer
auxiliary variable n : integer
begin
    n ← 0
    while not isEmptyList(ℓ) do
        n ← n + 1
        ℓ ← getNext(ℓ)
    endwhile
    return n
end
Linked Lists

function reverseScan(ℓ : PList)
begin
  if not isEmptyList(ℓ) do
    reverseScan(getNext(ℓ))
    print(getElement(ℓ))
  endif
end
Linked Lists: dynamic implementation versus arrays

Advantages of the dynamic implementation

- well adapted for the manipulation of sequences of variable lengths: gain of memory space;
- insertion and deletion of a cell without moving other values (insertion by “deviation” and deletion by “bypass”): gain of time.

Drawback of the dynamic implementation

sequential access (not direct contrary to arrays), in order to access to the $i$th element one needs to access to the 1st, 2nd, ... and $(i - 1)$th before: loss of time
Linked Lists

It is also possible to implement linked lists with arrays:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>$e_3$</td>
</tr>
<tr>
<td>3</td>
<td>$e_5$</td>
</tr>
<tr>
<td>4</td>
<td>$e_1$</td>
</tr>
<tr>
<td>5</td>
<td>$e_2$</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>$e_4$</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>$-1$</td>
</tr>
</tbody>
</table>

$l_1 = 4$, $l_2 = 5$, $l_3 = -1$

List of available places: available = 0
function allocate($T : \text{array of cells}$)
auxiliary variable $i : \text{integer}$
begin
    if $libre = -1$ then
        error(”no more room”)
    else
        $i \leftarrow available$
        $available \leftarrow T[available].next$
        return $i$
    endif
end
Lifetime of a dynamic variable

from the moment when it has been created till the end of the execution of the program except:

- implicit de-allocation by a garbage-collector mechanism: collect all the memory area that are not referenced any more (works in Java not in C)
- explicit de-allocation by `free(p)` that gives back the memory occupied by \( \uparrow p \) (does not change the value of the variable \( p \))

Beware to the ghost references!!
Plan

1. Recursivity
2. Linked Lists
3. Les types de données abstraites
Abstract Data Types (ADT)

An ADT is the description of an organized set of entities and of manipulation operations on this set. These operations include means for accessing and modifying the elements.

An ADT consists of:

- an interface (description of the set and of the operations);
- an implementation (description of the data structures and of their manipulation algorithms).

- interface: describes of the possible operations;
- implementation: explicits the representation of the elements.
Abstract Data Types

The concept of TDA does not depend on any programming language.

The programr decides to access the ADT through the limited set of functions defined by the interface and forbids any direct access to underlying data structures that would be possible by a detailed knowledge of the specific implementation.

An efficient realization of the ADT by data structures that implement the operations in an optimal way in terms of time and space constitute one aspect of algorithmics.
Stacks

LIFO (Last In First Out)

A stack is a linear list where insertions and deletions are all done at the same end (analogy: stack of plates).

- insertion: push
- deletion: pop

If the stack is not empty, the only accessible element is the top of the stack.

Fundamental property

Deletion inverts insertion (if one pushes an element and then pops it gets back to the initial state).
Stacks

Interface

emptyStack() : Stack  returns an empty stack

getTop(s : Stack) : D  returns the element located at the top of stack \( s \),
  \( s \) should not be empty

push(x : D, s : Stack)  inserts element \( x \) at the top of stack \( s \)

pop(s : Stack)  deletes element at the top of stack \( s \), \( s \) should not be
  empty

isEmptyStack(s : Stack) : boolean  tests is stack \( s \) is empty
Queues

**FIFO (First In First Out)**

A queue is a linear list where insertions are done at one end and deletions are done at the other end.

- insertion: enqueue
- deletion: dequeue

If the queue is not empty, the only accessible element is at the head of the queue.
## Queues

### Interface

- `emptyQueue() : Queue` returns an empty queue
- `getHead(q : Queue) : D` returns the element at the head of queue $q$, $q$ should not be empty
- `enqueue(x : D, q : Queue)` inserts element $x$ at the end of queue $q$
- `dequeue(q : Queue)` deletes element at the head of queue $q$, $q$ should not be empty
- `isEmptyQueue(q : Queue) : boolean` tests if queue $q$ is empty