Lecture 1 - Introduction in declarative programming. Recursion

Official web site: www.cs.ubbcluj.ro/~hfpop/pfl

Contents

References

Programming and programming languages

Recursion

Examples of recursion

References

Czibula, G., Pop, H.F., Elemente avansate de programare în Lisp şi Prolog. Aplicaţii în Inteligenţa Artificială., Ed. Albastră, Cluj-Napoca, 2012

Programming and programming languages

Algorithms, programs and data structures:

- **Algorithm**. An algorithm is the outline, the essence of a computational procedure, a sequence of instructions, given as a text.
- **Program**. A program is an implementation of an algorithm in some programming language.
- **Data structure**. Many data structures are needed to write a program that solves a problem. They are used to separate various structures and operations on these structures. A correct use of data structures leads to increased clarity and reduced complexity.

Algorithm - American Heritage Dictionary of the English Language, 5th Edition:

- A finite set of unambiguous instructions that, given some set of initial conditions, can be performed in a prescribed sequence to achieve a certain goal and that has a recognizable set of end conditions.

- A precise rule (or set of rules) specifying how to solve some problem; a set of procedures guaranteed to find the solution to a problem.

Concerning the meaning of an algorithm, the effect of its execution:

- each algorithm defines a mathematical function
- an algorithm is written to solve a specific problem
- more algorithms are available for solving a certain problem.

The main characteristics of an algorithm are:

- **generality**: the algorithm solves a problem for all possible values of the input data, not only in particular cases;
- **uniqueness**: the repeated execution of an algorithm for the same input data provides the same results;
- **finitude**: the algorithm is a finite text in a description language;
- **correctness** .

We would also like to note the following:

- An algorithm describes actions on the input instance.
- There are usually more than one correct algorithms for the same algorithmic problem.

The declarative view

Declarative programming means writing code in such a way that it describes **what** you want to do, and not **how** you want to do it.

Programming rules

R1. Know and follow the meaning of each variable.

R2. Use different names for variables with different meaning. Or, otherwise stated, do not use the same name for variables of different meaning.

R3. Do not use uninitialized variables.

R4. Completely know the problem to be solved.

R5. Do not use uninitialized variables.

R6. Do not reevaluate the limits and do not modify the loop variable inside a FOR repetitive structure.

R7. Choose suggestive names for variables.

R8. Postpone for later the insignificant details; concentrate your attention to the important decisions of the moment.

R9. Avoid reading and printing in a subalgorithm.

R10. Create a subalgorithm independent of the context where it will be used.

➢ LANGUAGES

- **Procedural (imperative) - high level languages**

    - Fortran, Cobol, Algol, Pascal, C, ...
    - program - sequence of instructions
    - the assignment statement, control structures - for the control of sequential execution, branching and cycling
    - the role of the programmer - “what” and “how”
        1. to describe what is to be calculated
        2. to organize the calculation
        3. to organize memory management

        HOW
    - !!! it is argued that the assignment instruction is dangerous in high-level languages, just as the GO TO instruction was considered dangerous for structured programming in the '68s.

- **Declarative (descriptive, applied) - very high level languages**

    - based on expressions
    - expressive, easy to understand (have a simple basis), extensible
    - programs can be seen as descriptions that state information about values, rather than instructions to determine values or effects.
    - they give up instructions
        1. thus they protect users from making too many mistakes
        2. they are generated from mathematical principles - analysis, design, specification, implementation, abstraction and reasoning (deductions of consequences and properties) become more and more formal activities.
    - the role of the programmer - “what” (not “how”)
    - two classes of declarative languages
        1. **functional languages** (eg Lisp, ML, Scheme, Haskell, Erlang)
            - ➢ focus on values of data described by expressions (built through applications of functions and definitions of functions), with automatic evaluation of expressions
        2. **logical languages** (e.g. Prolog, Datalog, Parlog),
            - ➢ focus on logical assertions that describe the relationships between data values and automatic derivations of answers to questions, starting from these assertions.
    - applications in Artificial Intelligence – automated proofs, natural language processing and speech understanding, expert systems, machine learning, intelligent agents, etc.

- Multiparadigm languages: **F#, Python, Scala** (imperative, functional, object oriented)
- Interactions between declarative and imperative languages - declarative languages that provide interfaces with imperative languages (eg C, Java): SWI Prolog, GNU Prolog, etc.
- **Logtalk** – integrates logic and object-oriented programming
- Logic programming in **Python**:
    - **Karen**
    - **SymPy** – library for simbolic computations

Recursion

- general mechanism to elaborate programs
- recursion arose from practical necessities (direct transcription of recursive mathematical formulas; see Ackermann's function)
- recursion is the mechanism by which a subprogram (function, procedure) calls itself
  - two types of recursion: **direct** or **indirect**
- **!!! Result**
  - any calculable function can therefore be expressed and programmed) in terms of recursive functions
- two things to consider in describing a recursive algorithm: **the recursive rule** and **the termination condition**
- **advantage** of recursion: source text that is extremely short and very clear.
- **disadvantage** of recursion: filling the stack segment if the number of recursive calls, respectively of the formal and local parameters of the recursive subprograms is high enough.
    - declarative languages have specific mechanisms to optimize the recursion (see the mechanism of tail recursion in Prolog).

Important results

Function

f : X → Y

Partial function

f : X → Y is a function from a subset S of X to Y

Total function

A partial function where S = X

Computable function

If there exists an algorithm that can to the job of the function

General recursive function
Partial recursive function

A partial function N → N that is computable

Total recursive function
Recursive function

If the function is total recursive function

Primitive recursive function

A function that can be computed by a computer program whose loops are all "for" loops, i.e. an upper bound of all iterations of every loop can be determined before entering the loop

Theorems:

All primitive recursive functions are total and computable.

Not all the total computable functions are primitive recursive.

Example: The Ackermann function

One of the simplest and earliest-discovered functions:

- total computable function
- not primitive recursive

Version 1

$F(m,n,0) = m+n$
$F(m,0,1) = 0$
$F(m,0,2) = 1$
$F(m,0,p) = m \qquad p>2$
$F(m,n,p) = F(m, F(m,n-1,p),p-1) \qquad n,p>0$

Version 2

$A(0,n) = n+1$
$A(m+1,0) = A(m,1)$
$A(m+1, n+1) = A(m, A(m+1,n))$

Version 3

$A_0(n) = n+1$
$A_{m+1}(n) = A_m^{n+1}(1) \qquad$ where $f^{n+1}(x) = f(f^n(x))$

Examples of recursion

Remarks

- a list is a sequence of items ($l_1 l_2 \dots l_n$)
- the empty list (with 0 elements) is denoted by $\emptyset$
- adding an item to a list is denoted by $\oplus$

1. Create list (1,2,3, ... n)

a) directly recursive

$$creareLista(n) = \begin{cases} \emptyset & daca\ n = 0 \\ creareLista(n-1) \oplus n & altfel \end{cases}$$

b) using a recursive auxiliary function to create the sublist (i, i + 1, ..., n)

// create the list consisting of the elements i, i + 1,…, n

Recursive mathematical model

$$creare(i,n) = \begin{cases} \phi & daca\ \ i > n \\ i \oplus creare(i+1,n) & altfel \end{cases}$$

// create the list consisting of elements 1, 2,…, n

$$creareLista(n) = creare(1,n)$$

Pseudocode

Data representation : singly linked list with dynamic allocation of nodes.

NodeLSI

e: TElement // useful information of node

urm: ^NodeLSI // address the following node is stored

LSI

prim: ^NodeLSI // address of the first node in the list

Function createNodeLSI(e)

{pre: e: TElement}

{post: return a ^NodeLSI having e as useful information }

{allocates a storing space for a NodeLSI }

{p: ^NodLSI}

allocate(p)

[p].e ← e

[p].urm ← NIL

{result returned by the function }

createNodeLSI ← p

EndFunction

Function create(i, n)

{post: return a ^NodLSI, pointer towards the head of the linked list formed by }

{ elements i, i+1,…, n }

If i > n **then**

create ← NIL

else

{ allocate a storage space for a NodeLSI with usefun information e }

q ← **createNodeLSI**(i)

{ create the link between node q and the head of the linked list formed }

{ by elements i+1,…, n }

[q].urm ← **create**(i+1, n)

create ← q

EndIf

EndFunction

Function createList(n)

{post: return a ^NodeLSI, pointer towards the head of the linked list formed by }

{ elements 1, 2,…, n }

createList ← **create**(1, n)

EndFunction

2. Given a natural number n, calculate the sum 1 + 2 + 3 +… + n.

a) directly recursive

$$suma(n) = \begin{cases} 0 & daca\ n = 0 \\ n + suma(n-1) & altfel \end{cases}$$

b) using a recursive auxiliary function for calculating the sum i + (i + 1) +… + n

$$suma\_aux(n, i) = \begin{cases} 0 & daca\ i > n \\ i + suma(n, i+1) & altfel \end{cases}$$

$$suma(n) = suma\_aux(n, 0)$$

3. Add an item at the end of a list.

// build the list (l1, l2,…, ln, e)

$$adaug(e, l_1 l_2 \ldots l_n) = \begin{cases} (e) & daca \quad l\ e\ vida \\ l_1 \oplus adaug(e, l_2 \ldots l_n) & altfel \end{cases}$$

4. Search for an element in a list.

$$apare(E, l_1 l_2 \ldots l_n) = \begin{cases} fals & daca\ \ l\ e\ vida \\ adevarat & daca \quad l_1 = E \\ apare(E, l_2 \ldots l_n) & altfel \end{cases}$$

5. Count the number of occurrences of an item in the list.

$$nrap(E, l_1 l_2 \ldots l_n) = \begin{cases} 0 & daca\ \ l\ e\ vida \\ 1 + nrap(E, l_2 \ldots l_n) & daca\ \ \ l_1 = E \\ nrap(E, l_2 \ldots l_n) & altfel \end{cases}$$

6. Check if a numeric list is set.

$$eMultime(l_1 l_2 \ldots l_n) = \begin{cases} adevarat & daca\ \ l\ e\ vida \\ fals & daca\ \ \ l_1 \in (l_2 ... l_n) \\ eMultime(l_2 \ldots l_n) & altfel \end{cases}$$

7. Transform a numeric list into a set.

$$multime(l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ multime(l_2 ... l_n) & daca\ \ \ l_1 \in (l_2 ... l_n) \\ l_1 \oplus multime(l_2 \ldots l_n) & altfel \end{cases}$$

8. Return the inverse of a list.

$$invers(l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ & \\ invers(l_2 ... l_n) \oplus l_1 & altfel \end{cases}$$

9. Remove all occurrences of an item from a list.

$$stergere(E, l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ l\ e\ vida \\ l_1 \oplus stergere(E, l_2 \ldots l_n) & daca\ \ \ l_1 \neq E \\ stergere(E, l_2 \ldots l_n) & altfel \end{cases}$$

10. Return the k-th element of a list (k> = 1).

$$element(l_1 l_2 \ldots l_n, k) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ l_1 & daca\ k = 1 \\ element(l_2, \ldots l_n, k-1) & altfel \end{cases}$$

11. Return the difference between two sets represented as lists.

$$diferenta(l_1 l_2 \ldots l_n, p_1 p_2 \ldots p_m) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ diferenta(l_2 \ldots l_n, p_1 p_2 \ldots p_m) \quad daca & l_1 \in (p_1 p_2 \ldots p_m) \\ l_1 \oplus diferenta(l_2 \ldots l_n, p_1 p_2 \ldots p_m) & altfel \end{cases}$$

<u>Homework</u>

1. Verify whether a natural number is prime.
2. Calculate the sum of the first k elements in a numeric list ($l_1 l_2 \ldots l_n$)
3. Remove the first k even numbers from a numeric list.