LECTURE 1

Introducere în programarea declarativă. Recursivitate

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

Contents

References

Chapter 1, 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

➢ 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 HOW
        3. to organize memory management
    - !!! 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 protects 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), which 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).

Examples of recursion

<u>Remarks</u>

- a list is a sequence of items ($l_1 l_2 \ldots l_n$)
- 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 amount *and*+ (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.

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

6. Check if a numeric list is set.

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

7. Transform a numeric list into a set.

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

8. Return the inverse of a list.

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

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

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

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.