**LECTURE 7. Functional programming. Introduction in LISP**

**Contents**



## 1. Introduction

### 1.1. A new programming paradigm

Functional programming is primarily a new programming paradigm, comparable in importance to the promotion of structured programming. It is important to emphasize that functional programming brings with it a new programming discipline, usable even in the absence of programming languages.

A complete approach to the issue of functional programming must contain all three hypostases under which it can be used, namely:

1. method for developing reliable programs;
2. means of analyzing the efficiency of programs;
3. method of analyzing the correctness of programs.

As a working principle, the functional programming methodology can be used with the same efficiency even if the implementation language is not a specific functional one. That is why we say that functional programming is a new programming paradigm (principle) and not just a new programming method.

Functional programming is also known as application programming or value-oriented programming. Its importance for the current moment results from the following reasons (which also represent defining features of functional programming):

- functional programming gives up the assignment instruction present as a basic element in imperative languages; more correctly, this attribution is present only at a lower level of abstraction (analogous to the comparison between goto and the structured control structures while, repeat and for);
- functional programming encourages thinking at a higher level of abstraction, allowing through higher order functions (so-called functional forms) to change the behavior of existing programs and combine them (practice called programming in the large - working with larger units than individual instructions ).
- the working principles of functional programming match the requirements of massively parallel programming: the absence of attribution, the independence of the final results from the evaluation order and the ability to operate at whole structures are three reasons why it seems that functional languages will prevail over those. imperative at the time of mass transition to parallel programming;
- the study of functional programming is closely related to the denotational semantics of programming languages; the essence of this semantics is the translation of conventional programs into equivalent functional programs.

Artificial intelligence has been the basis for promoting functional programming. The object of this field consists in the study of the way in which behaviors can be achieved with the help of the computer that are usually qualified as intelligent. Artificial intelligence, through the problems of specific applications (simulation of human cognitive processes, automatic translation, retrieval of information) requires, first of all, symbolic processing and less numerical calculations. This increased "power" of expressing (symbolic) calculations could be obtained on the basis of formal mathematical logical models such as lambda calculus (Lisp) or Markov algorithms (Snobol). These models were developed in 1941 and 1954, respectively, regardless of the existence of computers, as working tools in the theoretical study of calculability.

The characteristic of symbolic data processing languages consists in the possibility of manipulating data structures, no matter how complex, structures that are built dynamically (during the execution of the program). The information provided by this data mainly highlights the properties of the objects as well as the relationships between them. Data is usually strings, lists, or binary trees.

Not much emphasis will be placed on issues of functional language implementation, for the following reasons:

- on conventional architectures the functional languages are implemented using the techniques used for the implementation of functions in structured languages that admit recursion (Pascal, Modula, C);
- the implementation of functional languages on unconventional architectures is changing very fast nowadays, being another field of research. The information available now would soon become obsolete, making it unnecessary to address it in a basic course.

## 1.2. Brief history of functional programming

Georg Cantor (1845-1918) and Leopold Kronecker (1823-1891) stated that a mathematical object exists only if it can be constructed (at least in principle). The question thus arose "what does it mean that a number or a mathematical object is constructible?".

Giuseppe Peano (1858-1932) showed that natural numbers can be constructed by a large number of applications of the successor function. Since 1923, Thoralf Skolem (1887-1963) has shown that almost all natural number theory can be developed constructively based on the extensive use of recursive definitions. Thus, the beginning of the 20th century brings with it considerable experience in the field of recursive definitions of natural number functions.

To avoid using infinity, the buildable object was defined as an object that can be built in a finite number of steps, each step requiring a finite amount of effort. Attempts to formalize such a definition began in the 1930s, leading to the term calculability.

The first attempt was Turing's definition of the class of abstract machines (then known as Turing machines), machines that perform simple reads and writes on a finite portion of tape. Other attempts:

- the introduction of general recursive functions by Kurt Godel (1906-1978);
- the development of lambda calculus by Church and Kleene;
- Markov algorithms;
- Post production systems.

It is very interesting to note that all these independently formulated notions of computability have proved to be equivalent (Church, Kleene and Turing have proved it). This equivalence led Church to propose the thesis that bears his name (Church thesis),

namely that the notion of calculable function be identified with the notion of general recursive function; the idea of the latter belongs to the French mathematician Jacques Herbrand (1908-1931).

Thus, in the period immediately following the invention of the first electronic computers, it was shown that any calculable function can be expressed (and therefore programmed) in terms of recursive functions.

The next major event in the history of functional programming was the publication in 1960 by John McCarthy of an article on the Lisp language. In 1958 McCarthy investigated the use of list operations in implementing a symbolic differentiation program. As differentiation is a recursive process, McCarthy was led to use recursive functions. Moreover, it needed the ability to transmit functions as arguments to other functions. Lambda calculus helped him in this, and so McCarthy came to use Church's notations for his programs.

This is the motivation for starting a project in 1958 at MIT that aimed to implement a language that incorporated such ideas. The result was Lisp 1, described by McCarthy in the article "Recursive Functions of Symbolic Expressions and Their Computation by Machine" (1960). This article (which shows how a very large number of important programs can be expressed as pure functions operating on list-type structures) is considered as a starting point for functional programming.

Towards the end of the 1960s Peter Landin tried to define the non-functional language Algol 60 through lambda calculus. This approach was continued by Strachey and Scott and led to a method of defining the semantics of programming languages known as denotational semantics. In essence, denotational semantics defines the meaning of a program in terms of an equivalent functional program.

Extensive research in functional programming was started following an article by John Backus, published in CACM in 1978: "Can Programming Be Eliberated of the Von Neumann Style?". In this article, Fortran's inventor brought a severe critique of conventional programming languages, calling for the development of a new programming paradigm, which he called functional programming. Most of the functional forms proposed by Backus were inspired by the LPA language, an imperative language developed in the 1960s and which provided powerful operators to work on entire data structures.

### 1.3. Functional languages

Any programming language can be divided into two components:

- component independent of the chosen field of application, called the framework, which includes the basic syntactic (linguistic) mechanisms used in the construction of programs. For an imperative language the frame contains the control structures, the procedure call mechanism and the assignment operation. The framework of an application language includes the function definition and the function application mechanism.
- component dependent on the chosen application domain, which contains components useful to that domain. If the domain is, for example, numerical programming, the frame contains real numbers, operations (+, -), and relations (=, <, etc.).

So the framework provides the form and the components the content on the basis of which the programs are built. The choice of framework determines the type of language (applied or imperative). The choice of components orients the language to a class of problems (for example, numerical or symbolic).

A data type is called abstract if it is specified in terms of a (abstract) set of values

and operations and not in terms of a concrete implementation.

A component found in most application languages (and lacking in most imperative languages) is the sequence date structure. Sequence is an abstract type of data. Moreover, it is also a generic type (integer sequences, string sequences, are particular types). It is implemented as a simple chained linear list. In order to decide which should be the primitive operations on the sequence, recursion helps us (taking into account the fact that we want as few operations as possible). The prefix (cons) operation was preferred to the postfix one due to the way the simple chained linear list is implemented.

Regarding the operations we need to define an abstract type of data, they must be:

(i) **constructors** - operations that construct an abstract type of data in a finite number of steps based on the supply of components;
(ii) **selectors** - element selection operations according to desired criteria;
(iii) **discriminators** - operations capable of comparing different classes and instances of the abstract data type to verify that selectors can be applied to them.

As for current functional programming languages, they suffer from a lack of standard syntactic notations. This is because most of these languages are still experimental languages, one of the purposes being the experimentation of notations. On the other hand, the basis of these languages is the same foundation, namely lambda calculus. Most functional programming systems accept commands of three types:

(1) Definitions of functions
(2) Data definitions
(3) Expression evaluations

Functional programs are closer to the formal specifications of program systems. Unlike conventional languages, functional languages treat the function as a fundamental object (the principle of regularity or uniformity) that can be transmitted as a parameter, returned as a result of a processing, constituted as part of a data structure, etc. This increases the abstracting power of a language.

### 1.4. Brief history of the Lisp language

In 1956 John McCarthy organized a meeting in Dartmouth with artificial intelligence researchers. The need for a programming language specifically for artificial intelligence research was agreed upon.

Until 1958, McCarthy developed a first form of a language (Lisp - LISt Processing) for list processing, a form based on the idea of transcribing into that language the programming of algebraic expressions.

However, the imposition of Lisp as the basic language for programming artificial intelligence applications was made only after some concessions from the author, concessions intended to obtain more efficient implementations as well as more concise expressions. The last version developed by the author was Lisp 1.5 (1962). Subsequent subsequent implementations have remedied some of the shortcomings of that version, creating a number of new versions, most extensions. There is no standard version yet, a standardization has only been attempted in recent years. Lisp programming environments are large programs (usually also written in Lisp) that provide facilities for the interactive creation and debugging of Lisp programs and the management of all functions of a program. In general, a Lisp environment is designed based on an interpreter,

The best known Lisp environments are

- InterLisp (MIT, 1978);
- MacLisp (MIT, 1970, DEC10);
- Franz Lisp (Berkeley, under UNIX).

MacLisp allowed the development of one of the first real utility applications: the MACSYMA program, intended for symbolic processing in the mathematical field.

Here is a brief presentation of Lisp, made by the author himself:

- calculations with symbolic expressions instead of numbers;
- representation of symbolic expressions and other information through the list structure;
- composing functions as a tool for forming more complex functions;
- the use of recursion in the definition of functions;
- representation of Lisp programs as Lisp data;
- the Lisp eval function which serves both as a formal definition of language and as an interpreter;
- garbage collection as a means of dealing with the problem of allocation;
- Lisp instructions are interpreted as commands when a conversational environment is used.

The field of use of the language Lisp is that of symbolic calculation, ie of the processing performed on a series of symbols grouped in expressions. One of the causes of the force of Lisp language is the possibility of manipulating objects with a hierarchical structure.

Areas of Artificial Intelligence the Lisp language has found a wide field of applications are the demonstration of theorems, learning, the study of natural language, speech comprehension, image interpretation, expert systems, robot planning, etc.

## 2. Basic elements of the Lisp language

A Lisp program processes symbolic expressions (S-expressions). Even the program is such an S-expression.

The usual way of working of a Lisp system is the conversational one (interactive), the interpreter alternating the data processing with the user intervention.

The basic objects in Lisp are **atoms** and **lists**. The primary data (atoms) are **numbers** and **symbols** (the symbol is the Lisp equivalent of the variable concept in the other languages). Syntactically, the symbol appears as a **string** (the first being a letter); semantically, it designates an **S-expression**.

Atoms are used to build **lists** (most S-expressions are lists). A list is a sequence of atoms and / or lists.

In order to establish a rule for distinguishing argument operations (both being syntactically designated by symbols), Lisp adopted the prefixed notation (notation that also suggests the interpretation of operations as functions), the symbol on the first position of a list being the name of the function that applies .

Evaluating an S-expression means extracting (determining) its value. The evaluation of the value of the function takes place, naturally, after the evaluation of its arguments.

We will synthesize in the form of rules the way of formation (language syntax) and evaluation (language semantics) of S-expressions:

**3. Dynamic data structures**

The need to use dynamic data structures (DDS) arises due to the numerous applications in which data processed on the computer are not predictable either quantitatively or in terms of their structure. In such cases a more flexible memory management can only be ensured by using DDS.

Symbolic processing has two main features:

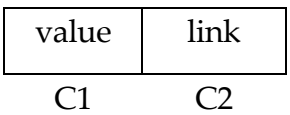
(a) The processed values are symbolic expressions (each variable is associated with a symbolic expression and not a numerical value);

(b) The main operations lead to the formation of new expressions from existing expressions.

Linear memory and the static memory allocation model prevent efficient operations on symbolic expressions. It was necessary to adopt internal representations of the data as general and flexible as possible, easily modified by the respective operations.

One of the best known and simplest DDS is the Singly Linked Linear List (SLLL), which is also the implementation model for an ADT required in symbolic processing: the sequence.

An item in the list consists of two fields: the value and the link to the next item. The links give us information of a structural nature that establishes order relations between elements):




(C1 . C2)

Variations in the content of these two fields generate other kinds of structures: if C1 contains a pointer then binary trees are generated, and if C2 can be anything other than a pointer then the so-called **dotted pairs** appear (two-field data elements).

Thus, elements can be formed whose fields can be equally occupied by atomic information (numbers or symbols) and structural information (references to other elements). The graphical representation of such a structure is that of a binary tree (tree structure), a structure that generalizes the notion of ordered pair in mathematics, allowing each element to be in turn a pair.

**Remark**: Any linear list can be represented as a tree (being a particular case of it), but not every tree can be represented as a list!

**Any list has an equivalent in dotted pair notation**, but not every dotted pair has an equivalent in list notation (generally only dotted notations where NIL is enclosed in parentheses can be represented in list notation). ). In Lisp, as in Pascal, NIL has the meaning of a null pointer.

The recursive definition of equivalence between lists and dotted pairs in Lisp:

(a) If A is atom, then the list (A) is equivalent to the dotted pair (A . NIL).

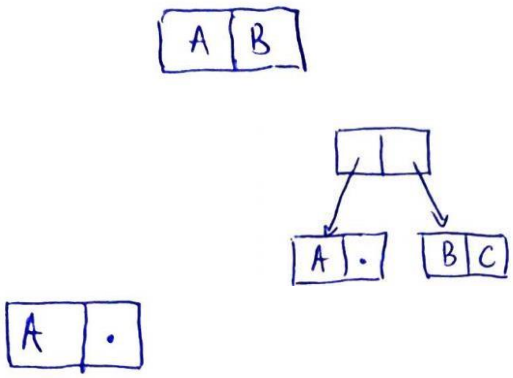
(b) If the list (l1 l2…ln) is equivalent to the dotted pair **<p>** then the list

(l l1 l2 … ln) is equivalent to the dotted pair (l . <**p**>).

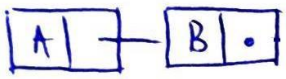
### 3.1. Examples

Here are some examples.

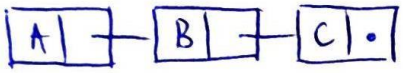
- (A . B) has no list equivalent;
- ((A . NIL) . (B . C)) has no list equivalent;
- (A) = (A . NIL)



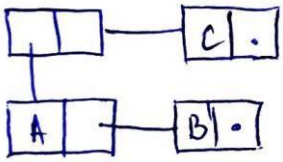

- (A B) = (A . (B))



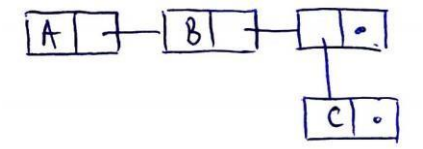

- ((A . NIL) . ((B . NIL) . NIL)) = ((A) (B))
- (A B C) = (A . (B . (C . NIL)))



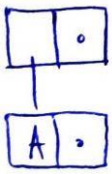

- ((A B) C) = ((A . (B . NIL)) . (C . NIL))



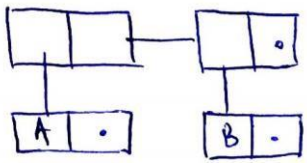
- (A B (C)) = (A . (B . ((C . NIL) . NIL))




- ((A)) = ((A . NIL) . NIL)




- ((NIL)) = ((NIL . NIL) . NIL)
- (()) = (NIL . NIL)
- (A (B . C)) = (A . ((B . C) . NIL))
- ((A) (B)) = ((A . NIL) . ((B . NIL) . NIL))



**4. Syntactic rules**

(a) a **numerical atom** is a string of digits whether or not followed by the dot character and preceded or not by a + or -;

(b) a **string atom** is a quoted string;

(c) a **symbol** is a string other than delimiters: space, (,), ', ",., comma, =, [,];
delimiters can appear in a symbol only if they are avoided (via the backslash "\");

(d) an **atom** is

a symbol,

a numerical atom,

or a string;

(e) a **list** is a construction of the form

() or

(e) or

(e1 e2 ... en)

with n> 1, where e, e1, ..., en are S-expressions;

(f) a **dotted pair** is a construction of the form (e1 . e2) where e1 and e2 are S-expressions;

(g) an **S-expression** is

an atom;

a list;

a dotted pair;

(h) a **form** is an evaluable S-expression;

(i) a **Lisp program** is a sequence of evaluable S-expressions.

## 5. Evaluation rules

The following rules for evaluating S-expressions apply:

(a) a numerical atom is evaluated by that number;

(b) a string is evaluated by its text itself (including quotation marks);

(c) a list is evaluable (ie it is a form) only if its first element is the name of a function, in which case all the arguments are evaluated first, after which the function is applied to these values and the result is determined.

**Remark:**The QUOTE function even returns the S-expression argument, which is equivalent to stopping the attempt to evaluate the argument. The character '(apostrophe) can be used instead of QUOTE.

The essence of Lisp consists in the processing of S-expressions. The data has the same form as the programs, which allows the launch as programs of some data structures as well as the modification of some programs as if they were ordinary data. As such, there is the possibility of writing self-modifying programs.

### 5.1. Examples

| | |
|---|---|
| > 'A<br>A | > (quote A)<br>A |
| > (A)<br>undefined function A | > (NIL)<br>undefined function NIL |
| > ' (A)<br>(A) | > ' (NIL)<br>(NIL) |
| > ()<br>NIL | > (())<br>undefined function NIL |
| > NIL<br>NIL | > ' (())<br>(()) |
| > (A.B)<br>undefined function A.B | > ' (A.B)<br>(A.B) |

## 6. Classification of Lisp functions

The functions of the Lisp system are of four categories, depending on the number of parameters - fixed or variable - and the moment of evaluating the arguments:

1. **subr functions** - have a fixed number of arguments; Lisp evaluates each argument first and then proceeds to evaluate the function;
2. **nsubr functions** - have a fixed number of arguments; argument evaluation is part of the function evaluation process;
3. **lsubr functions** - have a variable number of arguments; Lisp evaluates each argument first and then proceeds to evaluate the function;
4. **fsubr functions** - have a variable number of arguments; argument evaluation is part of the function evaluation process.

In the description of the Lisp functions we used the following notation from literature:

**FUNCTION category number-of-params (type-of-params): type-of-result**

## 7. Primitive Lisp functions

Due to the simplicity of the SLLL structure, three primitive functions were established: CONS (constructor), CAR and CDR (selectors).

**CONS subr 2 (e1 e2): l or pp**

The CONS function performs the operation of forming a dotted pair. It is applied on two S-expressions and has the effect of forming a new CONS element having the representations of S-expressions in the two fields (side effect). The return value of the function is the pair of dots or the list that has as representation in the computer the constructed CONS element. The CONS element built in this way is called the CONS cell. Argument values are not affected.

Here are some examples:

- (CONS 'A 'B) = (A . B)
- (CONS 'A '(B)) = (A B)
- (CONS '(A B) '(C)) = ((A B) C)
- (CONS '(A B) '(C D)) = ((A B) C D)
- (CONS 'A '(B C)) = (A B C)
- (CONS 'A (CONS 'B '(C))) = (A B C)

**CAR subr 1 (l or pp): e**

**CDR subr 1 (1 or pp): e**

Extraction of subexpressions from an S-expression can be done with the primitive CAR and CDR functions. Thus, CAR extracts the first element of a list or the left side of a dotted pair. Applied to an atom, the value is undefined (some systems return NIL).

The CDR function is complementary to CAR and extracts the rest of the list or the right side of a dot pair, respectively.

Here are some examples:

- (CAR '(A B C)) = A
- (CAR '(A . B)) = A
- (CAR '((A B) C D)) = (A B)
- (CAR (CONS '(B C) '(D E)) = (B C)
- (CDR '(A B C)) = (B C)
- (CDR '(A . B)) = B
- (CDR '((A B) C D)) = (C D)
- (CDR (CONS '(B C) '(D E)) = (D E)

CONS recreates a list from the fragments that CAR and CDR have created. It should be noted, however, that the object obtained as a result of the application of the CAR, CDR and CONS functions is not the same as the object from which it started, because CONS creates one new CONS cell:

- (CONS (CAR '(A B C)) (CDR '(A B C))) = (A B C)
- (CAR (CONS 'A '(B C))) = A
- (CDR (CONS 'A '(B C))) = (B C)

When repeatedly using the selection functions, the abbreviation Cx1x2 ... xnR can be used, equivalent to Cx1R o Cx2R o ... o CxnR, where the characters xi are either 'A' or 'D'. Depending on the implementations, it will be possible to use in this composition at most three or four CxiR:

- (CAADDR '((A B) C (D E))) = D
- (CDAAAR '((((A) B) C) (D E))) = NIL

- (CAR '(CAR (A B C))) = CAR

**SETQ fsubr 2, ... (s1 f1 ... sn fn): e**

**SET lsubr 2, ... (s1 e1 ... sn en): e**

The SET and SETQ side effects are used to give values to the symbols in Lisp.

**Definition**. The action by which a function, in addition to calculating its value, makes changes to the data structures in memory is called **a side effect**.

The SETQ function evaluates only the forms f1, ..., fn, while the SET function evaluates all its arguments. The values of even rank arguments become the values of the odd rank arguments (previously evaluated at symbols for SET and not evaluated for SETQ) and the result returned by functions is the value of the last argument form.

- (SET 'X 'A) = A — X is evaluated at A
- (SET X 'B) = B — A is evaluated at B
- (SET 'X (CONS (SET 'Y X) '(B))) = (A B) — X: = (A B) and Y: = A
- (SET 'X 'A 'L (CDR L)) — X: = A and L: = (CDR L)
- (SETQ X 'A) = A
- (SETQ A '(B C)) = (B C)
- (CDR A) = (C)
- (SETQ X (CONS X A)) = (A B C) — X is evaluated at (A B C)

The empty list is the only list case that has a symbolic name, NIL, and NIL is the only atom that is treated as the list, () and NIL representing the same object (the NIL element has in the CDR field a pointer to itself, and in CAR has NIL).

- (CDR NIL) = NIL

- (CAR NIL) = NIL
- NIL = NIL

**Remark**. Note the difference between () = NIL, the empty list, and (()) = (NIL), the list that has NIL as the only element.

- (CONS NIL NIL) = (())

**SETF macro 2, ... (p1 e1 ... pn en): e**

Note that in addition to the SET and SETQ functions, Lisp offers the SETF macro. The parameters p1, ..., pn are forms that when evaluating the macro access a Lisp object, and e1, ..., en are forms whose values will be related to the locations designated by the parameters p1, ..., pn corresponding. The result returned by the SETF is the value of the last evaluated expression, en.

To find out more about how SETF works, let's look at the following example:

- (SETQ A '(B C D)) A evaluated as (B C D)
- (SETF (CADR A) 'X) replace (CADR A) with X
  A evaluated as (B X D)
- (SET (CADR A) 'Y) evaluate (CADR A) as X
  initialize symbol X to value Y

- (SETQ X '(A B)) X is evaluated as (A B)
- (SETQ Y '(A B)) Y is evaluated as (A B)
- (SETF (CAR X) 'C) X is evaluated as (C B), and Y is evaluated as (A B)

In principle, primitive functions are sufficient for any kind of list-level operations, but the procedures become much more complicated as their structure increases. Therefore, any Lisp system provides a set of more powerful list processing functions, functions defined on the basis of primitives and on the basis of the already defined set. This set is called the system function set. The diversity of this set underlies the diversity of Lisp systems and the relative lack of language portability.