LECTURE 7. PART I. Functional programming

Contents

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 aassignment, the independence of the final results to the evaluation order and the ability to operate with whole structures are three reasons why it seems that functional languages will prevail over the imperative languages 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.

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.

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 (+, -), 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 data sequence 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.

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.