LECTURE 12. PART 2.
Other aspects of functional programming

Contents

1. Lisp parameters in "Dynamic Scoping" context

As we have seen so far, all occurrences of formal parameters are replaced by the values of the corresponding arguments. So, we have a variant of name calling, text calling, because in Lisp we are dealing with the dynamic determination of the field of visibility (dynamic scoping) simultaneously with the textual replacement of the evaluated values.

Although the parameter is related to the argument (in the sense of the reference call in imperative languages, ie the name of the argument and that of the parameter become synonymous), it is not a reference call, because changes in the values of formal parameters do not affect the value of the arguments.).

E.g:

- (DEFUN FCT (X) (SETQ X 1) Y) is assessed at FCT
- (SETQ Y 0) is evaluated at 0
- (FCT Y) is evaluated at 0

In the above sequence Y is evaluated at 0, and X binds to 0. The FCT call is thus equivalent to (FCT 0). X is a formal parameter, Y is the current parameter, and as such the modification of X does not affect Y, so we are NOT dealing with a reference call.

This demonstrates that the transmission of parameters in LISP is not done by reference. But how do we justify that there is no appeal by value? However, there are situations in which the value of the current parameter can be changed, for example by the action of the destructive effect functions RPLACA and RPLACD.

Note also that despite the "Dynamic scoping" characteristic, the variable Y is not related to the execution of the formal parameter just replaced by Y, ie the Y "internal" of the function, but retains its characteristic of global variable for FCT.
The body of a LISP function is the field of view of its formal parameters, which are called related variables in relation to that function. The other variables that appear in the function definition are called free variables.

The current execution context consists of all the variables in the program together with the values to which they are linked at that time. This concept is necessary to establish the value of free variables that intervene in the evaluation of a function, the evaluation in LISP being done in a dynamic context (dynamic scoping), by the call order of the functions, and not static, ie relative to the place of definition.

Let us recall for this purpose the difference between the static and the dynamic determination of the field of visibility within the imperative languages. Let Pascal be the program:

```
cousin
        a: integer;

procedure P;
begin
        writeln (a);
end;

procedure Q;
cousin
        a: integer;
begin
        a: = 7;
        P;
end;

begin
        a: = 5;
        Q;
end.
```

Static determination of the visibility domain (SDVD, static scoping) assumes that the P procedure acts in the definition environment and as a result it will always "see" the global variable. This is also the case of the Pascal language, a situation in which P called in Q will print 5 and not 7.

On the other hand, if we were dealing with dynamic determination of the visibility domain (DDVD), the P procedure will always print the last (in order of calls) variable defined (or linked, in Lisp terminology). In this case 7 will be printed, because the last binding of a is relative to the value 7.

The Lisp language has DDVD, this approach being more natural than the static one for the case of interpreter-based systems. A Lisp variant of the above program that highlights DDVD is:

```
(DEFUN P () A)
(DEFUN Q ()
        (SETQ A 7)
        (P)
)

(SETQ A 5)
(SETQ Y (LIST (P) (Q)))
(PRINT Y)
```

The list (5 7) will be printed, the value returned by P being the last A bound each time.

The advantage of dynamic binding is adaptability, ie increased flexibility. In the case of functional arguments, however, unwanted interactions may occur. To illustrate the kind of problems that can occur, let's take the example of the function form twice, which applies the function argument to a value twice:

```
(DEFUN twice (func val) (funcall func (funcall func val)))
```

Application examples:

- (twice 'add1 5) returns 7
- (twice '(lambda (x) (* 2 x)) 3) returns 12

However, if we happen to use the wave identifier and within its func, taking into account DDVD a name collision occurs, with the following effect:

- (setq val 2) returns 2
- (twice '(lambda (x) (* val x)) 3) returns 27, not 12

(as we would probably like to get). This is because the last binding (dynamic) of val was performed as a formal parameter of the function twice, so val became 3.

This problem has been called the FUNARG problem (functional argument). It is a matter of conflict between SDVD and DDVD. Its solution did not consist in giving up DDVD for LISP but in introducing a special form called function, which connects a lambda expression to its definition environment (thus treating that case statically). So,

- (twice (function (lambda (x) (* val x))) 3) is evaluated at 12

The conclusion is therefore that LISP has two DDV rules: DDVD by default and SDVD by means of the function construction.

2. The universe of expressions and the universe of instructions

Any programming language can be divided into two so-called universes (domains):

1. the universe of expressions;
2. the universe of instructions.

The universe of expressions includes all constructions of the programming language whose purpose is to produce a value through the evaluation process.

The universe of instructions includes instructions for a programming language. These are of two kinds:

(i) instructions that influence the control flow of the program:
- conditional instructions;
- jump instructions;
- cycling instructions;
- calls for procedures and functions.

(ii) instructions that alter the memory status:
- assignment (alters internal, primary memory);
- input / output instructions (alters external, secondary memory)

As a resemblance of these two universes, we can say that both alter something. But there are important differences. In the universe of instructions, the order in which the instructions are executed is usually essential. That is, the instructions

```
i: = i + 1; a: = a * i;
```

they have a different effect from the instructions

```
a: = a * i; i: = i + 1;
```

Whether

```
z: = (2 * a * y + b) * (2 * a * y + c);
```

Many compilers eliminate the redundant evaluation of the common subexpression 2 * a * y by the following substitution:

t: = 2 * a * y; z: = (t + b) * (t + c);

This substitution could be made in the universe of expressions because within an expression a subexpression will always have the same value.

Within the universe of instructions, the situation changes, due to possible side effects. For example, for the sequence

y: = 2 * a * y + b; z: = 2 * a * y + c;

factorization of the common subexpression provides the non-equivalent sequence

t: = 2 * a * y; y: = t + b; z: = t + c;

Although a compiler can analyze a program to determine for each subexpression whether it can be factorized or not, this requires sophisticated global flow analysis techniques. Such analyzes are expensive and difficult to implement. We conclude, therefore, that the universe of expressions has an advantage over the universe of instructions, at least in relation to this aspect of the elimination of common subexpressions (there are, however, other advantages that will be highlighted below).

The purpose of functional programming is to extend to the entire programming language the advantages that the universe of expressions promotes over the universe of instructions.

3. Independence of the evaluation order

Evaluating an expression means extracting its value. The evaluation of the expression 6 * 2 + 2 provides the value 14. The evaluation of the expression E = (2ax + b) (2ax + c) cannot be done until we specify the values of the names a, b, c and x. So the value of this expression depends on the context of the evaluation. Once this is stated, it is also important to emphasize that the value of the expression E does not depend on the order of evaluation (ie the replacement of names, first the first factor or the second, etc.). Even parallel evaluation is possible. This is because in pure expressions (such as mathematical ones) the evaluation of a subexpression cannot affect the value of any other subexpression.

Definition. A pure expression is an expression free of side effects, ie it contains neither explicit (type C) nor implicit (function calls) assignments.

This property of pure expressions, namely the independence of the evaluation order, is called the Church-Rosser property. It allows the construction of compilers that choose evaluation orders that make the most efficient use of machine resources. Let us emphasize again the potential for parallelism displayed by this property.

The referential transparency property assumes that in a fixed context the replacement of the subexpression with its value is completely independent of the surrounding expression. So, once evaluated, a subexpression will not be evaluated again because its value will not change. Referential transparency results from the fact that arithmetic operators have no memory, so any call from an operator with the same inputs will produce the same result.

One of the characteristics of mathematical notation is the manifest interface, ie, the input-output connections between a subexpression and the surrounding expression are visually obvious (for example in the 3 + 8 expression there are no "hidden" inputs) and the outputs depend only on the inputs. The producers of side effects, and therefore the functions in general, do not have a manifest interface but a non-manifest interface (hidden interface).

Let's review the properties of pure expressions:

- the value is independent of the order of valuation;
- expressions can be evaluated in parallel;
- referential transparency;
- lack of side effects;
- the inputs and effects of an operation are obvious.

The purpose of functional programming is to extend all these properties to the entire programming process.

Applied programming has a single fundamental syntactic construction, namely the application of a function to its arguments. The ways to define the functions are the following:

1. **Enumerative definition**. It is only possible when the function has a small finite range.
2. **Definition by composing already defined functions.**If it is the case of a definition in terms of an infinite or unspecified number of compositions, such a method is useful only if a principle of regularity can be extracted, a principle that allows us to generate cases not yet listed based on those specified. If such a principle exists we are in the case of a recursive

definition. In functional programming, recursion is the basic method for describing an iterative process.

Another distinction to be made regarding the definition of functions refers to explicit and implicit definitions. An explicit definition tells us what a thing is. An implicit definition states certain properties that a certain object has.

In an explicit definition the defined variable appears only in the left limb of the equation (for example y = 2ax). Explicit definitions have the advantage that they can be interpreted as rewriting rules (rules that specify that one class of expressions can be replaced by another).

A variable is defined by default if it is defined by an equation in which it appears in both members (for example 2a = a + 3). To find its value, the equation must be solved.

Repeated application of the rewriting rules always ends. However, some default definitions may not terminate (ie they do not define anything). For example, the equation without solution a = a + 1 leads to an infinite substitution process.

An advantage of functional programming is that, as with elementary algebra, it simplifies the transformation of implicit definitions into explicit definitions. This is very important because the formal specifications of software systems usually take the form of implicit definitions, but the conversion of specifications into programs is easier in the case of explicit definitions.

Let us also emphasize that recursive definitions are by their nature implicit. However, due to their regular form of

specification (ie the left limb consists only of the name and arguments of the function) they can be used as rewriting rules. The stop condition ensures the completion of the substitution process.

Compared to conventional languages, functional languages also differ in the operational models used to describe the execution of programs. It is not proper, for example, for functional languages to follow step-by-step execution, because the order of evaluation of subexpressions does not matter. So we don't need a troubleshooter in the true sense of the word.

However, the Lisp language allows you to track the execution of a Lisp form. For this there is the TRACE macro-definition which receives as argument the name of the desired function, the result returned by TRACE being T if the respective function exists and NIL otherwise.