LECTURE 7. PART II. Introduction in LISP

Contents

1. 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 S-expressions. are created (language syntax) and evaluated (language semantics).

2. 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 creation 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):

value	link
C1	C2

(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

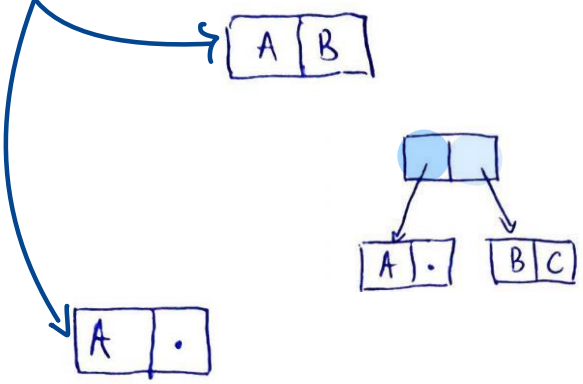
(l l1 l2 … ln) is equivalent to the dotted pair (l . <**p**>).

* NIL is both an atom and a list, it means empty list a false

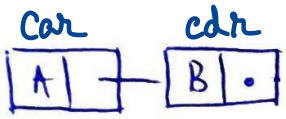
2.1. Examples

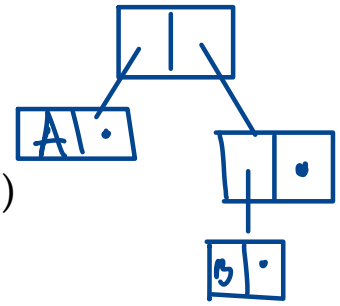
Here are some examples.

- (A . B) has no list equivalent; B is data, not an address
- ((A . NIL) . (B . C)) has no list equivalent;
- (A) = (A . NIL)



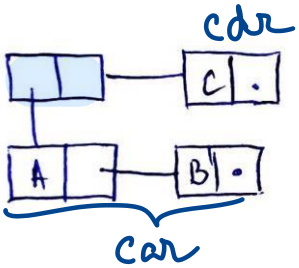
- (A B) = (A . (B))



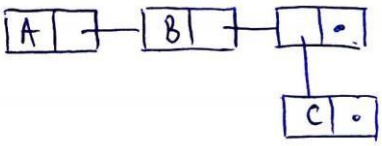


- ((A . NIL) . ((B . NIL) . NIL)) = ((A) (B))
- (A B C) = (A . (B . (C . NIL)))

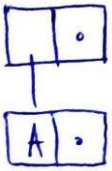
- ((A B) C) = ((A . (B . NIL)) . (C . NIL))

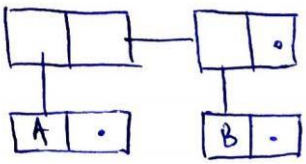


- (A B (C)) = (A . (B . ((C . NIL) . NIL))



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





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

3. 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.

4. 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 returns the S-expression argument itself, which is equivalent to stopping the attempt to evaluate the argument. The character '(apostrophe) can be used instead of QUOTE. return the expression as is, not evaluated

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.

4.1. Examples

> 'A A	> (quote A) A
> (A) → function with 0 params undefined function A	> (NIL) undefined function NIL
> ' (A) (A)	> ' (NIL) (NIL)
> () NIL	> (()) undefined function NIL
> NIL NIL	> ' (()) (())
> (A.B) undefined function A.B	> ' (A.B) (A.B)

5. 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 fusnction;
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

6. 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 :) rethink if you had to do this
- (CAR '(CAR (A B C))) = CAR

7. Assignment: SET, SETQ, SETF

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:

A := B C D
A := B X D
X := Y

• (SETQ A '(B C D))	A evaluated as (B C D)
• (SETF (CADR A) 'X)	replace (CADR A) with X
↳ changes the structure of A !!	A evaluated as (B X D)
• (SET (CADR A) 'Y)	evaluate (CADR A) as X
↳ changes the value	initialize symbol X to value Y

X := A B
Y := A B
X := C B

• (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)

8. Other list constructors

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.

LIST lsubr 0,1, ... (... e ...): l

LIST is a function with a variable number of S-expression parameters. Returns the list of argument values to the surface.

- (LIST '(a b) 'c) = ((a b) c)
- (LIST 'a) = (CONS 'a NIL) = (a)
- (LIST '(a b) '(c d)) = ((a b) (c d))
- (LIST '(a b c) '()) = ((a b c) NIL) = (LIST '(a b c) ())

It is indicated for constructing lists. Suppose we start from the symbols X, Y and Z which have as values A, B and C respectively and that we want the list of values of the symbols X, Y and Z. Then,

- (X Y Z) is not correct because X will be interpreted as function;
- '(X Y Z) it is not correct because it thus prevents evaluation;
- (LIST X Y Z) correct and provides the list (A B C).

APPEND lsubr 0,1, ... (... l ...): l

APPEND is a function with variable number of parameter lists (the last can be an object of any type). Copies the list structure of each argument, replacing the CDR of each last CONS with the argument on the right. Returns the resulting list. Note that all atomic parameters except the last parameter are ignored.

- (APPEND '(A B) '(C D)) = (A B C D)
- (APPEND 'A) = A
- (APPEND '(A B) 'C) = (A B . C)
- (APPEND '(A B C) '()) = (A B C)
- (APPEND 'A 'B '(C D) 'E 'F) = (C D . F)

because they don't have a next param

Remarks

- the APPEND function is strongly memory consuming; the first argument list is copied before being linked to the next;
- due to the fact that the last argument of the APPEND function is not copied, it is worth noting what happens when the last argument of the function is destructively modified (for example with SETF):

X := (A)
Y := (B)
Z := (A Y)
X := (D)
Y := (E)

- (SETQ X '(A)) — X is evaluated at (A)
- (SETQ Y '(B)) — Y is evaluated at (B)
- (SETF Z (APPEND X Y)) — Z is evaluated at (A B)
- (SETF (CAR X) 'D) — X is evaluated at (D)
- (SETF (CAR Y) 'E) — Y is evaluated at (E) but Z is evaluated at (A E)

this is a copy of X
⇒ changes cannot be seen!
hence the evaluation