Programming: Common Lisp/Tutorial

From Wikibooks, the open-content textbooks collection

http://en.wikibooks.org/wiki/Programming:Common_Lisp/Tutorial

navigation

- Main Page
- Help
- Wikijunior
- Wikistudy
- Wikiversity
- Wikiprofessional
- Books by subject
- Books alphabetically
- Books near completion

toolbox

- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this article

[edit]

Basic Stuff

This chapter gives some theoretical basics about structure of Lisp programs. You need to read it to understand the code.

[edit]

Syntax

Lisp operates on forms. Each Lisp form is either an atom or a list of forms. Atoms are numbers, strings, symbols and some other structures. Lisp symbols are actually quite interesting - I'll talk about them in another section.

When Lisp is forced to evaluate the form it looks whether it's an atom or a list. If it's an atom then its value is returned (numbers, strings and other data return themselves, symbols return their value). If the form is a list Lisp looks at the first element of the list - its **car** (yep, some Lisp terminology is quite strange and dates back into the olden times). The car of that list should be a symbol or a lambda-expression (lambda-expressions would be discussed later). If it's a symbol Lisp takes its function (the function associated to that symbol - NOT its value) and executes that function with the arguments taken from the rest of the list (if it contains forms they're evaluated as well).

Example: (+ 1 2 3) returns 6. Symbol "+" is associated with the function **+** that performs the addition of its arguments. (+ 1 (+ 2 3) 4) returns 10. The second argument contains a form and is evaluated before being passed to the outer **+**.

[edit]

Some interesting functions

+, **-**, *****, / are basic operations on numbers. They can accept multiple arguments. Note that (/ 1 2) is 1/2, not 0.5 - Lisp knows rational numbers (as well as complex ones...). **<**, **<=**, **=** and so on are used for number comparison. Note that **=**, **<**, **<=** and others are ***polyadic*** :

```
(= 1 1 1) => t
(= 1 1 2) => nil
(< 1 2 3) => t
(< 1 3 2) => nil
```

list as the name suggests creates a list.

```
(list 1 2 3) => (1 2 3)
```

cons creates a pair (pair is NOT the list of 2 elements).

```
(cons 1 2) => (1 . 2) ;;note the dot.
```

car or first return the first element of the cons (pair). **cdr** or **rest** return the second element of the cons.

```
(car (cons 1 2)) => 1  (cdr (cons 1 2)) => 2
```

[edit]

Lists and conses

Since lists are so prominent in Lisp it's good to know what they exactly are. The truth is: lists consist of conses. Except one special list called **nil** aka **()**. **nil** is a ***self-evaluating*** symbol that is used both as a falsehood constant and an empty list. **nil** is the only false value in Lisp - everything else is true for the purpose of **if** and similar constructs. The opposite of **nil** is **t**, which is also self-evaluating and represents the truth. **t** is however not a list. Let's return to the lists then... The definition of a ***proper*** list (I won't explain improper lists in this document) is that it's either **nil** or a **cons** whose **cdr** is a proper list. (Note that since proper lists have to start from somewhere, (cdr (cdr (cdr... (cdr x)))...) is **nil** for some finite number of **cdr**s.)

So (1 2 3) is really (1 . (2 . (3 . nil))). What follows from that is that (car (list 1 2 3)) is 1 and (cdr (list 1 2 3)) is (2 3). What doesn't follow from that is that (car nil) and (cdr nil) are **nil**.

[edit]

Symbols

Symbols play the same role as variable names in other programming languages. Basically a symbol is a string associated with some values. The string can consist of any characters, including spaces and control characters. However you probably wouldn't use symbols with weird names because they're awkward to type. By default Lisp converts what you type to uppercase, so in some sense it's a case-insensitive language.

As was said you can use almost any characters for your symbols. Avoid using " ", """, "(", ")", "#", "\", ".", "|" and ";" because the Lisp reader is likely to misunderstand you at the sight of them.

Symbols are created as you use them. For example when you type (setf x 1) symbol called "X" is created (remember that Lisp uppercases your input) and its value is set to 1. However it's a good style to define your symbols before you use them. **defvar** and **defparameter** are used for that purpose.

(defparameter x 1) ;;defines symbol "X" and sets its value to 1.

Symbols also have other stuff associated with them - functions, classes and so on. To get a function associated with a symbol, a special operator (these are discussed in the next chapter) **function** is used.

[edit]

Other things besides functions

There are some things in Lisp that look like functions but behave slightly different. These are macros and special operators. The difference from the functions is that the arguments are not always evaluated and that makes the things much more interesting.

The first special operator is **quote**. It returns its only argument, ***unevaluated*** . This is impossible with functions as they always evaluate their argument. Quote is used quite often so it has a shortcut ' . (quote x) is equivalent to 'x. **quote** may be used to quickly conjure up lists: '(1 2 3) returns (1 2 3), '(x y z) returns (X Y Z) - compare that to (list x y z) which would create a list of ***values*** of x, y and z, or signal an error if no values were assigned. In fact, '(x y z) is the same value as (list 'x 'y 'z).

You can also notice that such construct as **if** cannot be implemented as a function because only one of its arguments should be evaluated and not the other. In Lisp **if** is a special operator, which takes three (or less) arguments: the first one is always evaluated: if it is not **nil**, second argument is evaluated, if it is **nil**, third argument is evaluated. For example (if t 1 2) returns 1 and (if nil 1 2) returns 2.

Macros are like special operators except they're not hardcoded in Lisp implementation but are defined in Lisp code. A lot of Lisp constructs you will be using are actually macros. Only very essential stuff is hardcoded. Of course, to the user there is no difference.

[edit]

Doing things

In this chapter I'll explain how to do simple things in Lisp. Most of the useful constructs would be introduced. After reading this chapter you should be able to write simple programs.

[edit]

Storing values

While storing values in variables is an important thing in many programming languages, in Lisp it is used much less often. As you may have noticed I never stored values anywhere in the previous chapter... except in that "Symbols" section. That's because symbols are for storing values. Macro **setf** stores a value into a symbol:

```
(setf x 1) => 1
x => 1
```

Maybe it's quite awkward compared to C's x=1; but you'll be using setf less often than in C. That's because there is another way in Lisp to remembering values: binding them.

[edit]

Binding values

With **let** and **let*** you can bind some values to some variables within some part of your program.

```
(let ((x 1) (y 2) (z 3)) (+ x y z))
 => 6
(let* ((x 1) (y (+ x 1)) (z (+ y
1))) (+ x y z)) => 6
```

Inside the body of **let** you can use variables you defined as though they're real symbols - outside of **let** these symbols can be unbound or have completely different values. If you call or define functions within the **let** body the bindings stick around, thus some interesting interactions are made possible (which are outside of the scope of this manual). You can even use **setf** on these variables and the new value would be stored temporarily. As soon as the last form in **let** body is executed its result is returned, and the variables are restored back to their original values. The difference between **let** and **let*** is that **let** initialises its variables in parallel and **let*** does it sequentially.

In general you should prefer binding over setfing just because most Lisp programmers do.

[edit]

Control flow

The **if** operator was already explained before, but you probably are not able to use it at this point. This is because **if** allows only one form for each branch, which makes it hard to use in most situations. Fortunately while C has curly braces and Pascal has begin/end, Lisp gives you more freedom to define your blocks. **progn** creates a very simple block of code, executing its arguments one by one and returning the result of the last one. **let** and **let*** can also be used for that purpose especially if you want some temporary variables inside the branches. **block** creates a named block, from which you can return with **return-from**:

```
(block aaa
  (return-from aaa 1)
  (+ 1 2 3)) => 1 ;;The form (+ 1
2 3) is not evaluated.
```

...and there are also **the**, **locally**, **prog1**, **tagbody** and so on. Fortunately, if you're not into writing Lisp macros, **if** is probably the only construct where you'll have to use blocks.

As **if** is quite ugly there are some convenient macros in place so you won't have to use it all that often. **when** takes its first argument and if it's not nil it evaluates the rest of its arguments, returning the last result. **unless** does the same if its first argument is **nil**. Otherwise they both return **nil**.

cond is slightly more complicated. It tests the conditions until one of them is not nil and then evaluates the associated code. The syntax of **cond** is as follows:

```
(cond (condition1 some-forms1)
      (condition2 some-forms2)
      ...and so on...)
```

case is similar to **cond** except it branches after examining the value of some expression:

```
(case expression
  (values1 some-forms1) ;values is
 either one value
```

```
  (values2 some-forms2) ;or a list
 of values.
  ...
  (t some-forms-t)) ;executed if no
 values match
```

or evaluates its arguments until one of them is not nil, returning its value, or the value of last argument.

and evaluates its arguments until one of them is nil, returning its value (nil, that is) - otherwise the value of the last argument is returned.

You may notice that **or** and **and** could be used as logical operations as well - remember that everything non-nil is true.

[edit]

Loops

As in every area of Lisp programming there are plenty of tools for evaluating something many times. The most useful tool is **loop** - in its simplest form it simply executes its body until the **return** operator is called, in which case it returns the specified value:

```
(setf x 0)
(loop (setf x (+ x 1)) (when (> x
10) (return x))) => 11
```

More complicated forms of **loop** are better learned through examples. While **loop** should be enough for all kind of loops, there are other constructs for those who don't bother to learn its full syntax.

dotimes executes some code a fixed number of times. Its syntax is:

```
(dotimes (var number result)
  forms)
```

dotimes increments var from 0 to number-1 and executes forms that number of times, returning result in the end.

dolist iterates through a list: its syntax is the same as of **dotimes** except there is a list instead of number.

mapcar applies a function to different sets of arguments and returns a list of results, for example:

```
(mapcar #'+ '(1 2 3) '(3 4 5) '(6
7 8)) => (10 13 16)
```

#'+ is a shortcut for (function +). The function + is first applied to the list of arguments (1 3 6), then to (2 4 7) and then to (3 5 8).

[edit]

Defining functions

Functions are defined using macro **defun**:

```
(defun function-name (arguments)
  body)
```

The created function is associated with the symbol function-name and can be called like any other function. It's worth mentioning that functions defined that way can be recursive or can call each other.

The special operator **lambda** creates an anonymous function, which you can use for a one-time purpose. Its syntax is the same except instead of "defun function-name" you write "lambda". These functions cannot be recursive.

You can also bind functions temporarily with **flet** and **labels**. They are very similar to **let** and **let***. The difference between them is that in **labels** a function can refer to itself, while in **flet** it refers to a former function with the same name instead.

[edit]

Calling functions

To call a function f with arguments a1, a2 and a3 simply type

```
(f a1 a2 a3)
```

Sometimes the function which should be called is stored in a variable and you don't know its name beforehand. Or maybe you don't know how many arguments are to be passed. In those cases functions **funcall** and **apply** become handy. Like all functions they evaluate their arguments - the first one should produce a function to be called, the rest should produce arguments. **funcall** just calls the function with whatever arguments supplied and **apply** checks if its last argument is a list and if it is, it treats it as a list of arguments. Compare:

```
(funcall #'list '(1 2) '(3 4))  =>
 ((1 2) (3 4))
(apply #'list '(1 2) '(3 4)) => ((1
 2) 3 4)
```

[edit]

User interaction

For this tutorial I cover only simple tasks of input and output. To read a value from the user use the **read** function. It will attempt to read an arbitrary Lisp expression from the input, and the value returned by read is that expression.

```
>(read)  ;Run read function
(+ 1 x)  ;That's what the user
types
(+ 1 X)  ;that's what is returned
```

In that example the returned value is a list of three elements: symbol +, number 1 and symbol X (note how it got uppercased). **read** is one of the three functions that comprise ***read-eval-print*** loop, the core element of Lisp. This is the same function that is used to read the expressions you type at the Lisp prompt.

While **read** is handy for receiving numbers and lists from the user, other data-types are expected by most users to be fed to computer in a different manner. Take strings for example. To make **read** recognise a string one must add double quotes around it `"like this"`. But normal user expects to just type `like this` without the quotes and press Enter. So, there is a different general-purpose function used for input: **read-line**. **read-line** returns a string that contains what user typed before pressing Enter. You can then process that string to extract the information you need.

For output there is also quite a number of functions available. One that is interesting to us is **princ**, which simply prints a supplied value, and also returns it. This may be confusing when using it in the Lisp console:

```
>(princ "aaa")
aaa
"aaa"
```

The first aaa (without the quotes) is what is printed, the second one is a returned value (which is printed by **print** function and it is not as nice looking). Another function that could be useful is **newline**, which prints a new line to the output.