FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ

7. LISP: MAKRA, PRINCIPY FUNKCIONÁLNÍHO PROGRAMOVÁNÍ V COMMON LISPU

2011 Jan Janoušek
MI-FLP

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

WHEN TO USE MACROS

FUNCTION OR MACRO?

- can be both:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

Here, it is better to use the function

- only macro:

```
(defmacro while (test &body body)
      `(do ()
           ((not ,test))
           ,@body))
```

No function could do that; in a function call, all the arguments are evaluated beforethe function is even invoked.

Argument transformation

- e.g. the SETF macro, which picks apart its arguments before evaluation

The converse of `car` is `rplaca`, of `cdr`, `rplacd`, and so on.

`(setf (car x) 'a)` expands into

```
(progn (rplaca x 'a) 'a)
```

To perform this trick, setf has to look inside its first argument and to transform it.

Conditional evaluation

- like IF, WHEN, COND, etc.

```
(defmacro when (condition &rest body)
   `(if ,condition (progn ,@body)))
```

Multiple evaluation of arguments

- like DO, WHILE, etc.

```
(defmacro while (test &body body)
      `(do ()
           ((not ,test))
           ,@body))
```

Using the calling environment

A macro expansion replaces the macro call in the lexical scope of the call; hence it can use and change lexical bindings in ways that functions can't.

```
(defmacro foo (x)
      `(+ ,x y))
```

depends on the binding of y where foo is called.

Using the calling environment

A macro expansion replaces the macro call in the lexical scope of the call; hence it can use and change lexical bindings in ways that functions can't.

```
(defmacro foo (x)
      `(+ ,x y))
```

depends on the binding of y where foo is called.

Wrapping a new environment

Within the body of an expression like

```
(let ((y 2))
     (+ x y)),
```

y will refer to a new variable.

Others

- Saving function calls

There is no overhead associated with macro calls. By runtime, the macro call has been replaced by its expansion.

- Integration with Lisp.

Sometimes you can write macros that transform problems, in a higher-level language of your own design, into simple Lisp.

Defining a new syntax…

CONS

- Functions are data - they can be passed as arguments, returned from other functions, etc. None of this is possible with macros.

- Clarity at runtime - macros can be harder to debug, you can't trace them (because they're gone by runtime) and stack backtraces are less informative when you use lots of macros.

CL AND FUNCTIONAL PROGRAMMING

COMMON LISP

- We have discussed that Common Lisp gathers three paradigms (styles) of programming:
  - **Functional programming**, which originated from the lambda calculus and is also the origin of Lisp. It is represented by functions, macros, symbolic rewritting and reduction style, recursion.
  - **Imperative programming**, which is represented by variables, blocks of a sequence of consecutive statements, iterative cycles.
  - **Object-oriented programming**, which is represented by OOP features, such as operators `defclass` and `defmethod` for example.
- This lecture considers a good style of **functional programming** in Lisp.

Lisp – a collection of functions

- Lisp is a collection of Lisp functions(, except for a small number of operators called *special forms*).
- If you think of something you wish Lisp could do, you can write it yourself, and your new function will be treated just like the built-in ones.
- Functions are Lisp objects.

Functional design

- Programs should evolve instead of being developed by the old plan-and-implement method.
- *Functional programming* means writing programs which work by **returning values** instead of by performing side-effects. Side-effects of functions are to be minimised.
- The following example will show how functional programming differs from what you might do in another, imperative, language.

Example - a function to reverse lists

```
(defun bad-reverse (lst)
 (let* ((len (length lst))
        (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
    (j (1- len) (1- j)))
     ((>= i ilimit))
     (rotatef (nth i lst) (nth j lst)))))
```

- The list is reversed by side-effects:

```
> (setq lst '(a b c))
(A B C)
> (bad-reverse lst)
NIL
> lst
(C B A)
```

Example, contd.

- Function `bad-reverse` is far from good Lisp style and draws its callers away from the functional ideal.

- The desired behaviour should be:

```
> (setq lst '(a b c))
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

Example, contd.

```
(defun good-reverse (lst)
 (labels ((rev (lst acc)
           (if (null lst)
               acc
               rev (cdr lst) (cons (car lst)
acc)))))
  (rev lst nil)))
```

- Generally, functional code looks fluid on the page, like definitions; imperative code looks solid and blockish, like C.

Example – built-in `reverse` function

- Like `good-reverse`, the built-in `reverse` works by returning a value — it doesn't touch its arguments. Operators like `reverse` are intended to be called for return values, not side-effects:

Instead of

```
(reverse lst)
```

we need to write

```
(setq lst (reverse lst)).
```

Example, contd.

- Instead of building new list structure, `bad-reverse` operates on the original list. However, for efficiency operating on the original data structure is sometimes necessary.

- For such cases, Lisp provides an O(n) **destructive** reversing function called `nreverse`.

- Even destructive functions usually work by returning values!

- You still can't write `(nreverse lst)` in the middle of a function and assume that afterwards `lst` will be reversed.

Example, contd.

- This is what happens in most Lisp implementations:

```
> (setq lst '(a b c))
(A B C)
> (nreverse lst)
(C B A)
> lst
(A)
```

- To reverse `lst`, you have would have to set `lst` to the return value, as with plain reverse.

General rules on side-effects

- Only a few Lisp operators are intended to be called for side-effects. In general, the built-in operators are meant to be called for their return values. (Don't be misled by names like sort, remove, or substitute.)
- If you want side-effects, use `setf` on the return value.
- Some side-effects are inevitable in general. However, having functional programming as an ideal doesn't imply that programs should never have side effects. It just means that they should have no more than necessary.

Multiple return values

- To make functional programming easier Lisp functions can return more than one value.

```
> (truncate 26.21875)
26
0.21875
```

- The first return value is catched implicitly, catching all return values is done by using a `multiple-value-bind`.

```
> (= (truncate 26.21875) 26)
T
```

```
> (multiple-value-bind (int frac) (truncate
26.21875) (list int frac))
(26 0.21875)
```

Multiple return values, contd.

- To return multiple values, we use the `values` operator:

```
> (defun powers (x)
    (values x (sqrt x) (expt x 2)))
POWERS
> (multiple-value-bind (base root square) (powers
4)(list base root square))
(4 2.0 16)
```

Another explanation of the difference

- A functional program tells you what it wants; an imperative program tells you what to do.
- A functional program says “Return a list of a and the square of the first element of x:”

```
(defun fun (x)
    (list 'a (expt (car x) 2)))
```

Another explanation of the difference

- An imperative programs says "Get the first element of x, then square it, then return a list of a and the square:"

```
(defun imp (x)
    (let (y sqr)
          (setq y (car x))
          (setq sqr (expt y 2))
          (list 'a sqr)))
```

- Note: In fact, the definition of `imp` is similar in form to the machine language code that most Lisp compilers would generate for `fun`.

Note on the difference

- A programmer used to programming in an imperative language often conceives of programs in imperative terms, and can find it easier to write imperative programs than functional ones in the beginning.
- This habit of mind is worth overcoming if you have a language that will let you.
- Functional programs can be written with unusual speed, and at the same time, can be unusually reliable.

Informal parables of the difference in Lisp

- Beginning to use Lisp may be like stepping onto a skating rink for the first time. It's actually much easier to get around on ice than it is on dry land — if you use skates. Till then you will be left wondering what people see in this sport.

- What skates are to ice, functional programming is to Lisp. Together the two allow you to travel more gracefully, with less effort.

From imperative to functional style

- There is a trick for transforming imperative programs into functional ones.
- You can begin by applying this trick to finished code. Soon you will begin to anticipate yourself, and transform your code as you write it.
- Soon after that, you will begin to conceive of programs in functional terms from the start.

From imperative to functional style

- An imperative program can be seen as a functional program turned inside-out.
- Vice versa, to find the functional program implicit in our imperative one, we just turn it outside-in.
- We show this method on the example of `imp` and `fun` functions. (Note. The first thing we can notice in `imp` is the creation of variables `y` and `sqr` in the initial `let`. This is a sign that bad things are to follow - uninitialized variables are so rarely needed that they should generally be treated as a symptom of illness in the program.)

From imperative to functional style

The method:

- Let us go straight to the end of the function. What occurs last in an imperative program occurs outermost in a functional one.
- So our first step is to grab the final call to list and begin stuffing the rest of the program inside it.
- We continue by applying the same transformation repeatedly.

Example

- Starting at the end, we replace: `sqr` with `(expt y 2)`, yielding:

```
(list 'a (expt y 2)))
```

- Then we replace `y` by `(car x)`:

```
(list 'a (expt (car x) 2))
```

- Now we can throw away the rest of the code, having stuffed it all into the last expression. In the process we removed the need for the variables `y` and `sqr`, so we can discard the `let` as well.