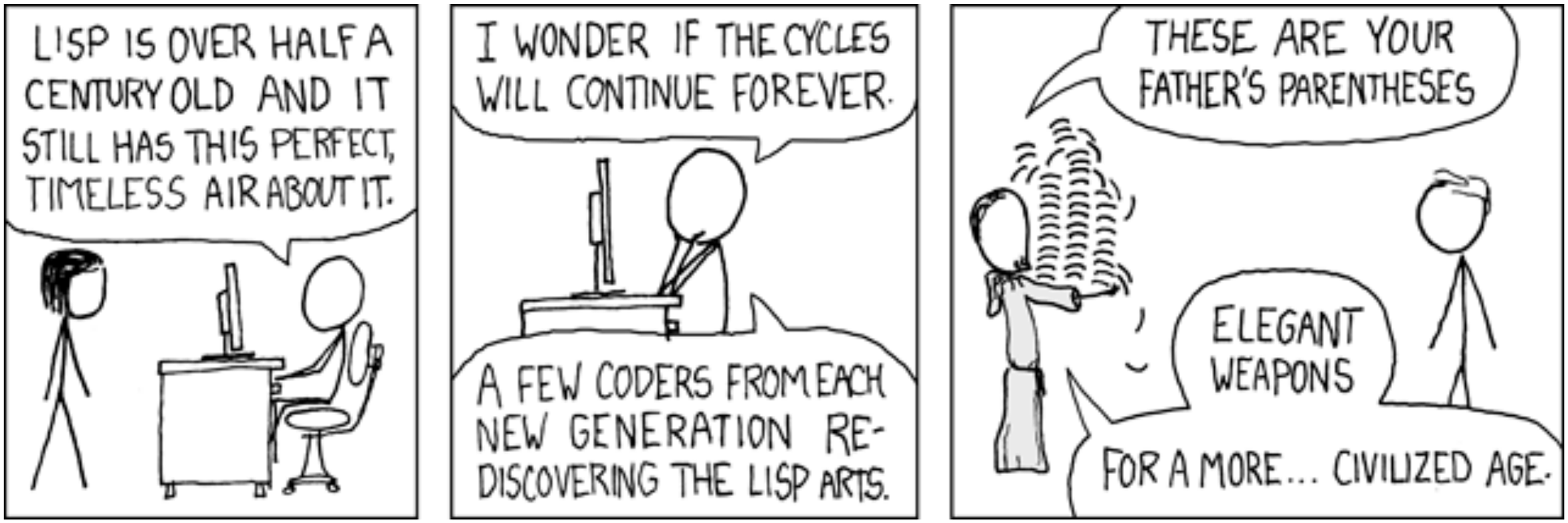
Advanced Functional Programming, 1DL450

Lecture 4, 2012-11-07

Cons T Åhs

Common Lisp



http://xkcd.com/297/

A Short History of Lisp

- The only language that is truly beautiful.
- Devised by John McCarthy (Sept 4, 1927 - Oct 24, 2011)
- First implementation as early as 1958
- First widespread implementation (Lisp 1.5) in 1961 (or so)
- Several competing implementations; MacLisp, InterLisp, T, NIL, ZetaLisp ..
  - Brought “together” in a standard; Common Lisp
- Inspired languages such as EmacsLisp, Scheme, Clojure
- Today, there is a large number of implementations of Common Lisp, both open source and commercial versions (often multi platform)
  - They all have the goal of implementing the standard
  - Differences are in quality, speed and non standard features, e.g., frameworks for GUIs
- Special purpose hardware - *Lisp Machines* - built for efficient execution of Lisp.

Large language

- Common Lisp is a very large language from the start
- The result of standardisation meant including many libraries in the standard
- The set of types is very rich, probably larger than any other language
- Very early effort to abstract away, e.g., the external file system making a cross platform development easier
  - Interestingly enough, file systems at that time were more diverse and capable than what we have today
  - Take a look at the handling of logical pathnames..
- It takes a lot of time to really master the language due to the sheer size of it
- It can probably be considered “too large” because the large number of constructs might lead to local dialects, while still being valid Common Lisp

Small core

- Originates from λ-calculus
- A minimal usable core has
  - S-expressions
    - *atoms* - numbers and symbols
    - *cons cells* - holds two S-expressions
  - Five functions
    - `atom` - determines if argument is an atom
    - `eq` - determines if the two arguments denote the same atom
    - `cons` - returns a cons cell of the two arguments
    - `car` - returns first S-expression of a cons cell
    - `cdr` - return second S-expression of a cons cell
  - Two (or three) special forms
    - `if` - conditional computation; evaluate first argument, depending on result evaluate second or third
    - `quote` - return argument without evaluation; denote a constant
    - (`lambda` - denotes a function, returns a closure)

Syntax

- Syntax is extremely simple, which makes it beautiful
- S-expressions (which is our data) have a written syntax
- atoms are written as expected, but the allowed character set is probably larger
  - `foo, foo-bar, *current-package*, +max-buf-size+, %internal, 42, 1.23, 2.71E6`
- cons cells are written within parentheses with a dot separating car and cdr
  - `(foo . bar)`
  - `(t . (42 . 17))`
  - `(a . (b . (c . nil)))`
- This is simplified by
  - if cdr is a cons cell, skip the dot and enclosing parentheses
  - if cdr is `nil`, skip the dot and don't write the nil
  - `(t 42 . 17)`
  - `(a b c)`
- This is list syntax - the dot is only needed when the last element is not `nil`
- `nil` denotes the empty list

Syntax for Programs & Data

- Easy to read for humans and easy to parse by a program
- List syntax is used for programs as well as data
  - might be confusing before you get used to it
  - strongly beneficial when you write macros, DSLs, language implementations, programs for analysing programs etc
- For a program (or expression) the first element in the list is the function *name* and the rest of the elements (which are expression) are the arguments to the function.
  - No infix operators, no precedence, no mess
  - Polish prefix notation

Lispisms

- Lisp has two very special atoms, which are constants
  - `T` - the eternal positive truth value
  - `NIL` - the eternal negative truth value
    - also the empty list (as is `()`)
- In a conditional expression NIL is “false” and everything else is “true”
- The original, and most often used, conditional is `COND`, not `IF`
  - Test each pair in order 0, 1 .. n; if C_k is non `NIL` evaluate E_k
  - If no condition is non `NIL`, `E` is valuated (since `t` is non `NIL`)
  - Given one of `COND` and `IF` the other can be defined by macro

```
(cond (C0 E0)
      (C1 E1)
       ...
      (Cn En)
      (t E))
```

```
(if C0 E0
    (if C1 E1
        (if
            ...
            (if Cn En
                E)...)))
```

Equality of S-expressions

- Observation:
  - S-expressions are binary trees without any info in the nodes
  - The leaves consist of atoms
- Two S-expressions are equal if
  - both are the same atom
  - their respective car and cdr parts are equal

```
(defun equal (s1 s2)
   (cond ((atom s1) (if (atom s2)(eq s1 s2) nil))
         ((equal (car s1) (car s2))
          (equal (cdr s1) (cdr s2)))
         (t nil)))
```

idiomatic version

```
(defun equal (s1 s2)
   (cond ((atom s1) (eq s1 s2))
         ((atom s2) nil)
         ((equal (car s1) (car s2))
          (equal (cdr s1) (cdr s2)))
         (t nil)))
```

- `eq` actually works on any two S-expressions, making ok to apply it an atom and cons cell.
- `eq` tests for *identity* - how can this be implemented?

List concatenation

```
(defun concat (list-1 list-2)
   (cond ((eq list-1 nil) list-2)
         (t (cons (first list-1)
                  (concat (rest list-1) list-2)))))
```

- Testing against `nil` is common so there is function for it - `null`
- A list is an abstraction built on S-expression
- Being strict and abstract, `car` and `cdr` are for S-expressions
- For lists, there are `first`, `second`, `third` .. and `rest`
- Traditionally, for accessing known parts of an S-expression there are combinations of `car` and `cdr`, e.g, `cadr`, `cddr`, `cadar` etc
- The predefined function `append` does list concatenation.

```
(defun cadr (sexpr)
   (car (cdr sexpr)))

(defun cadaar (sexpr)
   (car (cdr (car (caar sexpr)))))
```

Simple use

- To test the `equal` and `concat` functions we need to supply arguments to the functions.
- Trying the “obvious” does not work as code and data are represented the same way.
- Lisp will try to evaluate the arguments as function call and will fail
  - 1 is not a function
  - there is (probably) no function named `a`

```
CL-USER 34 > (concat (1 2 3) (a b c))

Error: Illegal argument in functor position: 1 in (1 2 3).
  1 (continue) Evaluate 1 and ignore the rest.
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :?
for other options.
```

Simple use

- Prevent evaluation by marking the argument as a literal
- Special form `quote` to the rescue!

```
CL-USER 46 > (quote (1 2 3))
(1 2 3)

CL-USER 47 > '(a b c)
(A B C)

CL-USER 48 > (concat '(x 1 2 3) '(a b c))
(X 1 2 3 A B C)

CL-USER 49 > (equal 'a 'c)
NIL

CL-USER 50 > (equal '((foo . a) . b) '((foo . a) . b))
T

CL-USER 51 > (equal '(foo bar) '(bar foo))
NIL
```

- Is `quote` a function?
- Instead of writing out `quote` one can write the quote symbol before a form
  - A *reader macro*, i.e., it is expanded at read time.

Types (some of them)

- Symbols (or atoms)
- Strings is a real data type
- Vectors (arrays) zero of more dimensions
- Hash tables
- Sequences (lists, strings, vectors)
- Numbers; ints, floats, bignums, but also rational numbers and complex numbers

```
CL-USER 54 > (sqrt -1)
#C(0.0 1.0)

CL-USER 55 > (/ 765 87)
255/29
```

Identity and equality

- There are several ways of comparing entities for equality
- `EQ`
  - identity, originally only for determining identity for atoms
  - “small” integers might be eq, but bignums are not
- `=`
  - numerical identity; arguments must be numbers
- `EQL`
  - same type and compares equal with `eq` or `=`
- `EQUAL`
  - structural equality
- What is the difference between these?
- Tip:
  - Use = when you know you have numbers
  - Use `EQL` for other “simple” objects.
  - Use `EQUAL` for structures only when needed - try to avoid it

Tuples?

- Lisp does not have tuples, but there are alternatives
  - lists
  - vectors
  - structs
  - classes (from CLOS)
- One use of tuples is to return multiple values from a function
  - The tuple is then deconstructed and the parts used.
  - The tuple is only a transport object
- Common Lisp supports *multiple value return*, i.e., a function actually returning several values which can then be used directly by the calling function
  - No transport object needed

Quirk - two namespaces!

```
(defun foo (defun null cons cond)
  (cond ((null defun) (cons cons cond))
        ((< (first defun) null)
         (foo (rest defun) null
              (cons (first defun) cons) cond))
        (t (foo (rest defun) null cons
                (cons (first defun) cond)))))
```

- What does this function do?
  - apart from making your eyes hurt
- Lisp has two namespaces so the value of a symbol depends on where it is
  - function position (first in an expression, unquoted)
  - variable position (anywhere else, unquoted)
- Immediate consequence is that higher order functions cannot be called in the "natural" manner.
- See lisp-1 vs lisp-2

Packages

- Each symbol belongs to a package (called module in some languages)
- There is no strict mapping between files and packages
  - Several files can introduce symbols belonging to the same package
  - It is even possible to have to introduce symbols to different packages from the same file
- Symbols are exported from a package, not only functions
- It is possible to access both external (exported) and internal symbols
- For two symbols to be EQ they have to belong to the same package

```
;; accessing exported symbols
image:dimensions
vector:dimensions

;; accessing internal (unexported) symbols
image::smooth-p
vector::polar-to-cartesian

;; special keyword package (for constants, no quote needed)
:to
:end
```

Anonymous functions

```
CL-USER 92 > (lambda (x) (+ x 3))
#<anonymous interpreted function 200FABFA>

CL-USER 93 > (defun mk-adder (n)
               (lambda (x) (+ x n)))
MK-ADDER

CL-USER 94 > (mk-adder 4)
#<anonymous interpreted function 200DCDE2>

CL-USER 95 > (lambda (x y) (append y x))
#<anonymous interpreted function 200DDE22>

CL-USER 96 > ((lambda (x) (* x x)) 7)
49
```

- An anonymous function is created with `LAMBDA`
- An anonymous function can be written in the function position, but not used in the normal and expected way when passed as an argument

Anonymous functions

```
CL-USER 97 > (defun call (f x) (f x))
CALL

CL-USER 98 > (call (mk-adder 3) 2)

Error: Undefined operator F in form (F X).
  1 (continue) Try invoking F again.
  2 Return some values from the form (F X).
  3 Try invoking something other than F with the same arguments.
  4 Set the symbol-function of F to another function.
  5 Set the macro-function of F to another function.
  6 (abort) Return to level 0.
  7 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for
other options.
```

- Due to the different namespaces `f` in the body is not the same as `f` in the argument list
- Use the function `funcall` to call a function

Anonymous functions

```
CL-USER 97 > (defun call (f x) (funcall f x))
CALL

CL-USER 98 > (call (mk-adder 3) 2)
5

CL-USER 105 > (call #'length '(1 2 3))
3

CL-USER 106 > (function length)
#<Function LENGTH 203A397A>

CL-USER 107 > #'length
#<Function LENGTH 203A397A>

CL-USER 108 > length

Error: The variable LENGTH is unbound.
  1 (continue) Try evaluating LENGTH again.
  2 Return the value of :LENGTH instead.
  3 ...
```

- Use special form `FUNCTION` or reader macro `#'` to obtain the function associated with a name.

Higher order functions

```
CL-USER 126 > (mapcar #'(lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)

CL-USER 127 > (map 'list #'(lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)

CL-USER 128 > (map 'vector #'(lambda (x) (* x x)) '(1 2 3 4))
#(1 4 9 16)
```

- A great number of predefined functions make use of functional arguments
- There are generalised functions working on sequences (lists, vectors, strings)

Local variables

```
(defun foo (x y)
  (let ((sum (+ x y))
        (diff (- x y)))
    (list sum diff)))

(defun oof (x y)
  (let ((x (+ x y 3))
        (y (- y x 4)))
    (list x y)))
```

```
CL-USER 137 > (foo 10 8)
(18 2)

CL-USER 138 > (oof 10 8)
(21 -6)
```

- Use `LET` to introduce local variables
- Scope is to the end of the `LET`
- Binding is parallel and free names refer to outer context
- Construct the equivalent application of `LAMBDA`

Local variables

```
(defun oof* (x y)
  (let* ((x (+ x y 3))
         (y (- y x 4)))
    (list x y)))

CL-USER 139 > (oof* 10 8)
(21 -17)
```

- `LET*` has sequential binding and free names refer to the context of the previous binding
- Construct the equivalent application of `LAMBDA`

Local functions

```
(defun bar (x y)
  (flet ((sqr (x) (* x x))
         (cube (x) (* x x x)))
    (+ 3 (sqr x) (cube x))))

(defun concat (x y)
  (labels ((app (l)
             (if (null l) y
               (cons (first l) (app (rest l))))))
    (app x)))
```

- You can also construct local functions
  - FLET behaves like LET
  - LABELS allows you to define locally recursive functions
- Useful for making helper functions
  - no pollution of namespace
  - they can use non local variables to decrease number of arguments

Function definitions

```
(defun decorated (x &optional (y 10 yp) &key to from)
  (list x y yp to from rest))

(defun varargs (x &rest y)
  (cons x (reverse y)))

CL-USER 160 > (decorated 2)
(2 10 NIL NIL NIL)

CL-USER 161 > (decorated 2 4 :from 4 :to 19)
(2 4 T 19 4)

CL-USER 162 > (varargs 1 2 3 4 5)
(1 5 4 3 2)
```

- Decorating the argument list of a function definition (even a `LAMBDA`) makes it possible to
  - have variable number of arguments
  - use keyword arguments (named arguments)
  - provide default values for optional arguments