Introduction to Functional Programming and basic Lisp

Based on Slides by
Yves Lespérance & Peter Roosen-Runge

Functional vs Declarative Programming

- *declarative* programming uses logical statements to describe objects.
  - Prolog is an example of this kind of language.
- *functional* programming uses mathematical functions and functional expressions to describe objects.
  - Functional programming has its roots in lambda calculus
  - Lisp is a functional language.

Functional Programming

- Functional programming is not based on assignments that change the state.
- Functions specify other values in terms of existing data without changing it.
- This allows all sorts of clever implementations e.g. on parallel hardware.

Background on LISP

- Acronym for LISt Processing
- created by the AI pioneer John McCarthy
- widely used in research on AI for over 40 years.
  - Used in industry to develop expert systems & other AI applications
- found inside applications like Emacs and AutoCAD as an *embedded* language
  - makes the embedding application easily extensible

Lisp as an *extensional* language

- embedding Lisp makes it easy to extend an application
- creation of new languages built "on" Lisp
- industrial-strength versions are usually standardized on Common Lisp

LISP Interpreter

- An interactive environment, always **evaluating input**
- To run LISP at prism labs, type: "clisp”
  To exit: Ctrl+D
- To load the file lp.lsp, in C:\MyFolder, use the following command:
  (load “C://MyFolder/lp.lsp”)

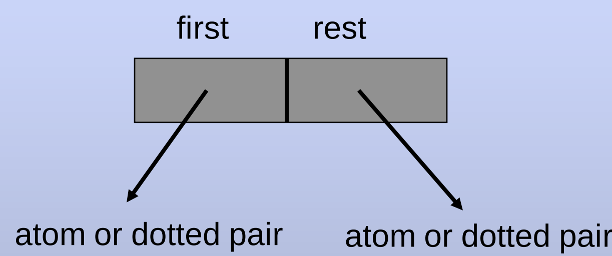
S-expressions

- A symbolic expression (s-expression) is defined inductively as
  - an atom (number or word), *or*
  - dotted pair of s-expressions (e.g. (x.y) where x and y are s-expressions)
- lists are the main kind of s-expressions
  - Example: (a b c) or a. (b . (c . nil)) - nil is the same as () - the only atom that is a list

disassembly

- functions which extract the two parts of a dotted pair:
  - first extracts the first part,
    - also called car
  - rest extracts the second part,
    - also called cdr

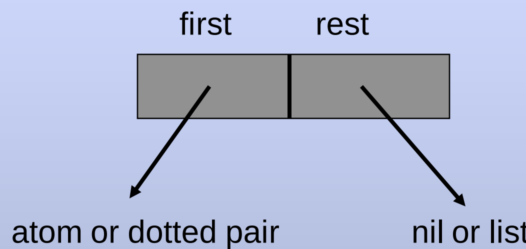
contents of a dotted-pair



new names for old

- car = function which returns the content referenced by address-register
- cdr = function which returns content referenced by decrement register
- many books cling to car and cdr for 'backwards compatibility'.
  - can use first and rest
  - you can use either - the Lisp interpreter doesn’t care.

list structure



functional expressions

- terms (functional expressions) are represented as lists
    - write f(x, y) as (f x y).
    - (a b c) represents the term a(b, c).
- already we see a bit of the power of symbolic computing:
    - expressions have same form as data
    - a function name is just an atom (a symbol)
    - could itself be computed

evaluating functions

- evaluating (function arg1 arg2 . .)
  - applies the system function eval to each argument
    - then applies the function to the results
  - (+ 2 (+ 3 5))

    eval(2) = 2

    eval((+ 3 5)) = +(eval(3), eval(5)) = +(3,5) = 8

    + (2, 8) = 10

blocking evaluation

- try evaluating
  - (reverse (a b)) --> error
    - no function a defined
  - (reverse (+ 1 2)) --> error
    - "3" is not a list
- how to block evaluation of an s-expression?
  - quote it

the uses of quotation

- try evaluating
  - (reverse (quote (a b))) --> (B A)
    - quote returns its argument **unevaluated**
  - (reverse '(+ 1 2)) --> (2 1 +)
    - '<*s-expression*> = (quote <*s-expression*>)
- (equal '(reverse (a b)) '(b a))) ?
  - NIL -- why?
- (equal (reverse '(reverse (a b))) '((a b) reverse)))
  - T

List Processing Functions

> (car '(a b c)) --> A
> (cdr '(a b c)) --> (B C)
> (car(cdr(car '((a b))))) --> (B)

- Other built-in functions for list processing include: cons, append, list, …

> (cons 'a '(b c)) --> (A B C)
> (append '(a) '(b) '(c)) --> (A B C)
> (list 'a 'b 'c) --> (A B C)

oceans of functions

- the basis for Lisp programs is the concept of a **function**
  - and variants with special properties
- Common Lisp has several hundred built-in functions
- many are redundant - could be replaced by expressions involving other functions

Function definitions

- Written as (defun *function-name* *arg-list* *result-spec*).
- *function-name* is a symbol.
- *arg-list* is a list of symbols, the parameters.
- *result-spec* is an expression whose value is the result of the function.
- When a function application is evaluated, substitute actual arguments for parameters in result-spec and return its value.

Function Example 1

> (**defun avg (x y) (/ (+ x y) 2.0))**

AVG

> (avg 1 2)

1.5

Function Example 2

- Rewrite (John P Doe) as (Doe John P)
- (defun last_name_first (name_list)
  (cons (third name_list)
  (cons (first name_list)
  (cons (second name_list)
  nil))))

Functions in LISP

- functions are considered as first class objects in Lisp.
- A function can take another function as an argument and a function can return a function as a value.
- This is what make functional programming very powerful.
- In a sense you can define your own control structures and manipulate programs!

More ….

- 'Practical Common Lisp' Book online
  http://www.gigamonkeys.com/book/
- Notes on Lambda Calculus
  http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf