A little bit of Lisp

Introduction to Artificial Intelligence

CSCE 476-876, Fall 2020

`www.cse.unl.edu/~choueiry/F20-476-876`

Read LWH: Chapters 1, 2, 3, and 4.

Every recitation (Monday): ask your questions on Lisp/xemacs.

Berthe Y. Choueiry (Shu-we-ri)

(402)472-5444

Features of Lisp

1. Interactive: interpreted and compiled

2. Symbolic

3. Functional

4. Second oldest language but still 'widely' used
   (Emacs, AutoCad, MacSyma, Yahoo Store, Orbitz, etc.)

Software/Hardware

- We have Allegro Common Lisp (by Franc Inc.): alisp and mlisp

- There are many old and new dialects (CormanLisp, Kyoto CL, LeLisp, CMU CL, SBCL, ECL, OpenMCL, CLISP, etc.)

- There have also been Lisp machines (Symbolics, Connection Machine, IT Explorer, others?)

Lisp as a functional language
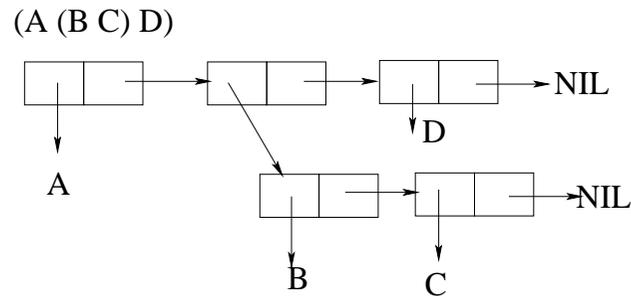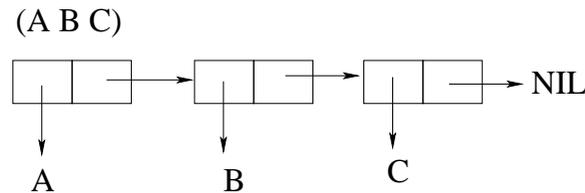
`(function-name arg1 arg2 `*etc*`)`

1. Evaluate arguments

2. evaluate function with arguments

3. return the result

Functions as arguments to other functions:

`(name2 (name1 arg1 arg2 `*etc*`) arg3 arg2 `*etc*`)`

Symbolic language

- Atoms: numeric atoms (numbers), symbolic atoms (symbols)
  Each symbol has: print-name, plist, package, symbol-value,
  symbol-function

- Lists:

(A B C)

(A (B C) D)

Symbolic expressions: symbols and lists

More constructs

- Data types:

  atoms and lists, packages, strings, structures, vectors,

  bit-vectors, arrays, streams, hash-tables, classes (CLOS), etc.

  NIL, T, numbers, strings: special symbols, evaluate to self

- Basic functions:

  `first (car)`, `rest (cdr)`, `second`, `tenth`

  `setf`: does not evaluate first argument

  `cons`, `append`, `equal`, operations on sets, *etc.*

- Basic macros:

  `defun`, `defmacro`, `defstruct`, `defclass`, `defmethod`,

  `defvar`, `defparameter`

- Special forms:
  `let`, `let*`, `flet`, `labels`, `progn`,

- Predicates:
  `listp`, `endp`, `atom`, `numberp`, `symbolp`, `evenp`, `oddp`, *etc.*

- Conditionals:
  `if <test> <then form> <else form>`,
  `when <test> <then form>`,
  `unless <test> <else form>`,
  `cond`,
  `case`

- Looping constructs:
  `dolist`, `dotimes`, `do`, `mapcar`, `loop`,

- Lambda functions

A really functional language

(defun <function-name> <arg1> <arg2> <arg3> ...
  (flet ((local-function-name <arg a> <arg b> ....
       ....
     <return some value>)
)

  (.............. #'(lambda (x) ....)..)

  <some-value>))

Regular function

Anonymous function

Local function

```
defun, flet/labels, lambda
```

What makes Lisp different?

Paradigms of AI Programming, Norvig

- Built-in support for lists

- Dynamic storage management (garbage collection!)

- Dynamic typing

- First-class functions (dynamically created, anonymous)

- Uniform syntax

- Interactive environment

- Extensibility

Allegro Common Lisp

- Free download: `www.franz.com/downloads/`

- Available on SunOS (csce.unl.edu), and Linux.

- Great integration with emacs
  Check `www.franz.com/emacs/` Check commands distributed
  by instructor

- Great development environment
  Composer: debugger, inspector, time/space profiler, etc.
  `(require 'composer)`

```
;;; -*- Package: USER; Mode: LISP; Base: 10; Syntax: Common-Lisp -*-


(in-package "USER")



;;;; +============================================================+
;;;; | Source code for the farmer, wolf, goat, cabbage problem |
;;;; |      from Luger's "Artificial Intelligence, 4th Ed."      |
;;;; |   In order to execute, run the function CROSS-THE-RIVER   |
;;;; +============================================================+
```

```
;;; +===============================================+
;;; | State definitions and associated predicates |
;;; +===============================================+
(defun make-state (f w g c)
  (list f w g c))


(defun farmer-side (state)
  (nth 0 state))


(defun wolf-side (state)
  (nth 1 state))


(defun goat-side (state)
  (nth 2 state))


(defun cabbage-side (state)
  (nth 3 state))
```

```lisp
;;; +=====================+
;;; | Operator definitions |
;;; +=====================+


(defun farmer-takes-self (state)
  (make-state (opposite (farmer-side state))
              (wolf-side state)
              (goat-side state)
              (cabbage-side state)))


(defun farmer-takes-wolf (state)
  (cond ((equal (farmer-side state) (wolf-side state))
         (safe (make-state (opposite (farmer-side state))
                           (opposite (wolf-side state))
                           (goat-side state)
                           (cabbage-side state))))
        (t nil)))
```

```lisp
(defun farmer-takes-goat (state)
  (cond ((equal (farmer-side state) (goat-side state))
          (safe (make-state (opposite (farmer-side state))
                            (wolf-side state)
                            (opposite (goat-side state))
                            (cabbage-side state))))
        (t nil)))


(defun farmer-takes-cabbage (state)
  (cond ((equal (farmer-side state) (cabbage-side state))
          (safe (make-state (opposite (farmer-side state))
                            (wolf-side state)
                            (goat-side state)
                            (opposite (cabbage-side state)))))
        (t nil)))
```

```
;;; +==================+
;;; | Utility functions |
;;; +==================+


(defun opposite (side)
  (cond ((equal side 'e) 'w)
        ((equal side 'w) 'e)))


(defun safe (state)
  (cond ((and (equal (goat-side state) (wolf-side state))
              (not (equal (farmer-side state) (wolf-side state))))
          nil)
        ((and (equal (goat-side state) (cabbage-side state))
              (not (equal (farmer-side state) (goat-side state))))
          nil)
        (t state)))
```

```
;;; +========+
;;; | Search |
;;; +========+

(defun path (state goal &optional (been-list nil))
  (cond
    ((null state) nil)
    ((equal state goal) (reverse (cons state been-list)))
    ((not (member state been-list :test #'equal))
     (or (path (farmer-takes-self state) goal (cons state been-list))
         (path (farmer-takes-wolf state) goal (cons state been-list))
         (path (farmer-takes-goat state) goal (cons state been-list))
         (path (farmer-takes-cabbage state) goal (cons state been-list)
     )))
```

```
;;; +==================+
;;; | Canned Execution |
;;; +==================+


(defun cross-the-river ()
  (let ((start (make-state 'e 'e 'e 'e))
        (goal (make-state 'w 'w 'w 'w)))
    (path start goal)))
```