CS 4100
LISP

April 18, 2012

Based on slides by Istvan Jonyer

Book by MacLennan

Chapters 9, 10, 11

LISP Is Interpreted

- Most LISP systems provide interactive interpreters
  - One can enter commands into the interpreter, and the system will respond

```
> (plus 2 3)
5
> (eq (plus 2 3) (difference 9 4))
t           (means ‘true’)
```

Pure vs Pseudo-Functions

- Pure functions
  - plus, eq, …
  - Only effect is the computation of a value
- Pseudo-functions
  - Has *side-effect*; more like a procedure
  - set
    - (set ‘text ‘(to be or not to be))
    - Side effect:
      - Sets the value of *text* to (to be or not to be)
    - Return value:
      - (to be or not to be)

Data Structures

- Primitives
  - Numbers
    - Operations: plus, minus, times, eq, etc.
  - Non-numeric atoms
    - Strings of characters used as symbols
      - Much like enumerated types in Pascal
      - Not used as strings
    - Operations: eq
    - Special atoms
      - t: true
      - nil: false; non-existent atom; empty list

Data Constructor

- The data constructor is the list
- Lists can have 0, 1 or more elements
  - Observes the Zero-One-Infinity principle
  - Empty list: ‘`() or nil`
- All lists are non-atomic (except empty list)

```
> (atom ‘())   or  (atom nil)  or  (atom 5)
t
> (atom ‘(to be)) or (atom ‘(()))
nil
```

Car and Cdr

- Accessing parts of a list
  - Car
    - Accesses first element of the list

```
>(car ‘(to be or not to be))
to
>(car ‘((to be) or (not to be)))
(to be)
```

    - Returns an element
  - cdr
    - Accesses rest of the list (list without first element)

```
>(cdr ‘(to be or not to be))
(be or not to be)
```

    - Returns a list

Combining *car* and *cdr*

- How do we select the second element?

```
>(car (cdr ‘(to be or not to be)))
be
```

- Third?

```
>(car (cdr (cdr ‘(to be or not to be))))
or
```

- How about this?

```
(set ‘DS ‘( (Don Smith) 45 30000 (Aug 4 80)))
```

  – Select day of hire

```
>(car (cdr (car (cdr (cdr (cdr DS))))))
4
```

- This can be simplified:

```
>(cadadddr DS)
4
```

Defining Functions

```
(set ‘DS ‘( (Don Smith) 45 30000 (Aug 4 80)))
```

- Define functions to replace cadadddr

```
(defun hire-date (r) (cadddr r))
(defun day (d) (cadr d))
```

  – Now we can select the day of the hire date as

```
(day (hire-date DS))
```

- This is more readable and more maintainable

Property Lists

- List like this are hard to maintain and read:
  ((Don Smith) 45 30000 (Aug 4 80))
  – We don’t know what elements mean
  – Hard to change the structure of the list
- A better way is to use property lists:
  (name (Don Smith) age 45 salary 30000 hire-date (Aug 4 80))
  – This way we can search for property name we want (age) and return value (45)
  – Order of properties becomes immaterial
  – General form (p_1 v_1 p_2 v_2 … p_n v_n)

Accessing Property Lists

(name (Don Smith) age 45 salary 30000 hire-date (Aug 4 80))

- How do we find the property?
  – If property we want is the first one, return second element of list
  – else skip first 2 elements, and start over
- In LISP (get property *p* of list *l*)

```
(defun getprop (p l)
    (if     (eq (car l) p)
            (cadr l)
            (getprop p (cddr l)) ))
```

Association Lists

- What if the property does not have a value? (e.g. “retired”)
- What is the property has more than a single value?
  – Of course, these can be solved using the property list, if we understand the properties of each property…
  – A better, more foolproof way is to use association-lists:

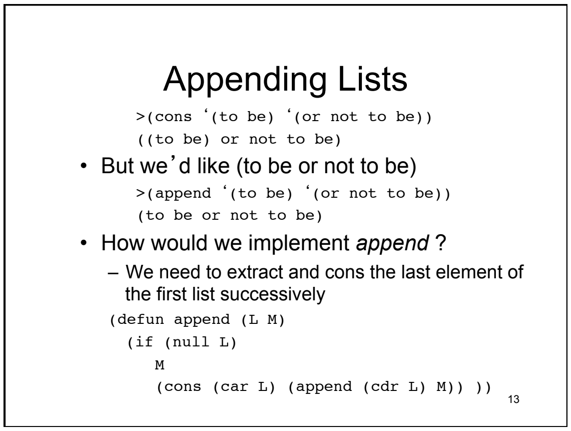
```
( (name (Don Smith))
  (age 45)
  (salary 30000)
  (hire-date (Aug 4 80)) )
```

Constructing Lists

- Need inverse of car and cdr
  – car: get first of list
  – cdr: get rest of list
- Inverse:
  – cons: append first of list to rest of list

```
>(cons ‘to ‘(be or not to be))
(to be or not to be)
>(cons ‘(to be) ‘(or not to be))
((to be) or not to be)
```

  – Returns a list



Appending Lists

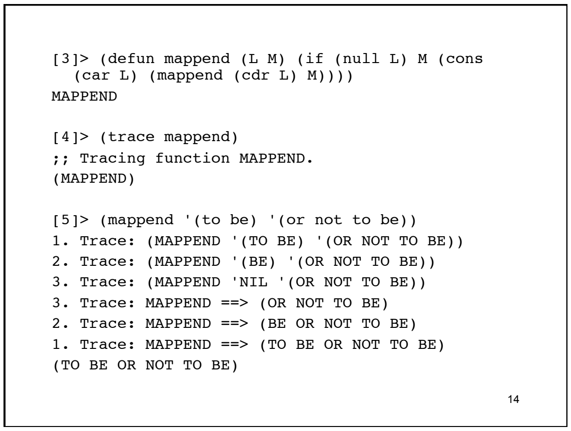
```
>(cons ‘(to be) ‘(or not to be))
((to be) or not to be)
```

- But we’d like (to be or not to be)

```
>(append ‘(to be) ‘(or not to be))
(to be or not to be)
```

- How would we implement *append* ?
  - We need to extract and cons the last element of the first list successively

```
(defun append (L M)
  (if (null L)
     M
     (cons (car L) (append (cdr L) M)) ))
```



```
[3]> (defun mappend (L M) (if (null L) M (cons
   (car L) (mappend (cdr L) M))))
MAPPEND

[4]> (trace mappend)
;; Tracing function MAPPEND.
(MAPPEND)

[5]> (mappend '(to be) '(or not to be))
1. Trace: (MAPPEND '(TO BE) '(OR NOT TO BE))
2. Trace: (MAPPEND '(BE) '(OR NOT TO BE))
3. Trace: (MAPPEND 'NIL '(OR NOT TO BE))
3. Trace: MAPPEND ==> (OR NOT TO BE)
2. Trace: MAPPEND ==> (BE OR NOT TO BE)
1. Trace: MAPPEND ==> (TO BE OR NOT TO BE)
(TO BE OR NOT TO BE)
```