Lisp Programming

Boolean functions, free and bound variables, logical operators, conditionals, recursion

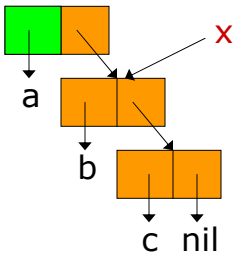
Shakil M. Khan
adapted from Gunnar Gotshalks

last week

- denotational programming/symbolic computing
- Lisp data structures
- Lisp functions
  - function invocation (list)
  - function definition (defun)
  - some basic functions (car, cdr, cons, quote, setq, set)

recap : setq & set (example)

- (setq x 'y)
  - x has the "value" y
- (setq l '(a b c))
  (setq x (cdr l))
  - x "points to" (b c)
  - this is because cdr returns the right pointer of (a b c)
  - so if you modify cdr of l, x will also change
- how to verify this in Lisp?



equal and eq

- (equal x y)
  - T if x and y are equal, otherwise nil
- (eq x y)
  - T if x and y represent the same object, else nil
- example:
  - (setq x '(a b c))
  - (setq y '(b c))
  - (setq z (cdr x))
  - (equal (cdr x) y), (equal (cdr x) z) → **T**
  - (eq (cdr x) y) → **NIL**
  - (eq (cdr x) z) → **T**

Boolean functions (predicates)

- returns **T** for true and **nil** for false
  - (atom item)
    is item an atom, i.e. not a cons cell
  - (listp item)
    is item a list, i.e. a cons cell
  - (null item)
    is item the empty list nil
  - (consp item)
    is item a non-empty list
  - note, in Lisp anything other than nil is true

predicates (cont'd)

- in general have a predicate for every type
  - e.g. numberp, evenp, oddp, zerop, …
- typep - generic type predicate
  - general form: (typep item 'type)
  - type can be one of: list, atom, symbol, number, …
  - e.g. : (typep 5 'number)
- (member item list)
  - is item a member of list
  - (member 'a '((a) b c)) → NIL
  - (member 'b '((a) b c)) → (b c)

other useful functions

- append
  - returns a list; the concatenation of its args
  - (append '(a b) '(c) '((c))) → (a b c (c))
- list
  - returns a list concatenating its args
  - (list '(a b) '(c)) → ((a b) (c))
- 1+, 1-
  - (setq x 5)
  - (1+ x) → 6, (1- x) → 4
- reverse
  - (reverse '(a b c)) → (c b a)

variables

- global or free variables
- local or bound variables

free variables

- global variables are defined at the upper level (outside of function definitions)
- within a function consistently changing the name of a free variable will normally alter the semantics of the function

(defun f1 (a) (+ a y))

is **NOT** the same as

(defun f1 (a) (+ a z))

bound variables

- local variables are defined in the parameter list of a function definition
- within a function consistently changing the name of a bound variable will not alter the semantics of the function

(defun f1 (y) (+ a y))

is the same as

(defun f1 (z) (+ a z))

scope of variables

- different variables can be denoted by the same symbol – Lisp carefully avoids conflicts using scoping rules
- static or lexical scoping
- dynamic scoping

static and dynamic scoping (example)

- consider the following:

```
( defun f1 ( v1 ) ( f2 v1 ) ) ;; v1 defined – argument
( defun f2 ( v2 ) ( 1+ v1 ) ) ;; is v1 defined?
( f1 7 )
```

- under static (lexical) scoping invoking (f1 7) produces an error as v1 is undefined in f2
- under dynamic scoping v1 in f2 is defined because f2 is executed in the environment of f1 in which v1 is defined
  - dynamic scoping leads to the funarg problem as function arguments can shadow (hide) global variables

execution environment

- an environment consists of binding between a set of symbols and their values
  ((A 1) (B 5) ... (D (a b c)))
- at the interpreter level global symbols are created, using setq or defun, giving a global environment
- the value of a symbol is looked up in the environment
- evaluating a function causes the parameters to be prepended to the appropriate environment
  - evaluating (f1 3) defined as (defun f1 (v1) (f2 v1))
  - creates the environment:
    ((v1 3) (A 1) (B 5) ... (D (a b c)))

execution environment (cont'd)

- we evaluate (f2 v1) in the context
  ((v1 3) (A 1) (B 5) ... (D (a b c)))
- v1 has the value 3 – passed as an argument to f2
- f2 is defined as
- (defun f2 (v2) (1+ v1))
- what environment does f2 use?
  - we have two choices
    - dynamic scoping
    - static scoping

execution environment (dynamic scoping)

- passes the existing environment
  ((v1 3) (A 1) (B 5) ... (D (a b c)))
- after prepending (v2 3)
  v2 is the parameter of f2 and 3 is the argument from f1
- the following is passed
  ((v2 3) (v1 3) (A 1) (B 5) ... (D (a b c)))
- so v1 has a definition in the environment
- the environment grows and shrinks on entry and exit from functions
  - a different environment for every function

execution environment (static scoping)

- passes the top-level environment in the context of the definition of f2
  - the same environment passed to f1
  - after prepending (v2 3)
    - the following environment is passed
      ((v2 3) (A 1) (B 5) ... (D (a b c)))
    - so v1 has NO definition in the environment
- environment on entry is fixed by the static structure
  - the same environment for every function

CLisp scoping rules

- is CLisp dynamically or statically scoped?
- in general, what happens to a formal parameter does not effect code outside a call to that function
  - local variables are **truly** local
  - e.g.: (defun f1 (**x**) (setq **x** 5))
    (defun f2 (**x**) (f1 x) **x**)
    (f2 3) → ?
  - may use same symbol as a parameter to different functions

CLisp scoping rules (cont'd)

- changes to a global variable inside a function persist
  - e.g.: (setq **x** 2)
    x → 2
    (defun f1 (y) (setq **x** y))
    (f1 3)
    x → ?
  - can be used to communicate between functions
  - hard to understand what a program with a global var. (*name*) is doing; use with care

dynamic scoping – funarg problem

- dynamic scoping leads to the funarg problem as function arguments can shadow global variables
  - (defun funarg (func arg) (funcall func arg))
  - (defun timesArg (x) (* arg x))
  - (setq arg 2)
  - (funarg 'timesArg 3)

  = 3 (both)

  = 3 (dynamic)
  = 2 (static)

  in a static environment the result is 6

  in a dynamic environment the result is 9

  → see Wilensky chapter 8 (8.3.1)

funarg problem (cont'd)

- lambda functions have dynamic scoping in PowerLisp on the Mac
  - (defun funarg (func val) (funcall func val))
  - (setq val 2)
  - (funarg '(lambda (x) (* val x)) 3)

  the result is 9
- use the following
  - (setq notVal 2)
  - (funarg '(lambda (x) (* notVal x)) 3)

  the result is 6

logical operators

- reverse a Boolean result
  - (**not** (atom item))
- combine predicates – lazy evaluation
  - (**and** (listp a) (listp b))
    stop evaluating as soon as false is found
  - (**or** (listp a) (listp b))
    stop evaluating as soon as true is found
  - (and (listp l) (listp (car l)))
    if l is not a list then (car l) would fail
    lazy evaluation prevents this

generic conditional – cond

- general template
  - (cond (p.1 s.1-1 s.1-2 ... s.1-p)
    (p.2 s.2-1 s.2-2 ... s.2-q)
    ...
    (p.n s.n-1 s.n-2 ... s.n-r)
)
- p.i are predicates
- s.i-k is the k'th S-expression for predicate p.i
  - usually only one per predicate

conditional (cont'd)

- uses lazy evaluation
  - evaulate p.i in turn, for i : 1 .. n
  - for the first true p.i evaluate s.i-1 ... s.i-r
  - value of cond is value of s.i-r
  - if all p.i are false, value of cond is nil
- example
  - note use of **T** to handle the otherwise case
  - (cond ((atom a) a)
    ((atom (car a)) (print (car a)) (cdr a))
    (**T** (process (car a)))
)

recursion (brief into.)

- only looping method in **pure** Lisp is recursion
  - involves reducing a problem to some computation that involves a smaller version of the same problem
  - e.g. length_of List can be defined as
    (1+ (length_of (cdr List))
  - consider (length_of (a b)); after 2 recursive calls to length_of, it'll try to compute (length_of nil)…will run forever!
  - so need a "grounding condition" (base case) to indicate when to stop
  - thus, also need to specify: if List is nil then return 0

recursion in Lisp

- in general, recursion involves
  - taking a list apart
    (car theList) (cdr theList)
  - doing recursion on the parts
    (recursiveCall (car theList))
    (recursiveCall (cdr theList))
  - rebuilding a new list from the parts of the old list
    (cons (recursiveCall (car theList))
    (recursiveCall (cdr theList)))
  - empty list is often used used for termination
    (cond ((null theList) (do base case)) ...)

recursion in Lisp – a general template

- (defun recursive (theList otherParams)
  (cond ((null theList) (first base case))
  ...
  ((pred1) (last base case))
  ((pred2) (first recursive case))
  ...
  (t (last recursive case))
)
)

recursion example 1

remove **item** from **list** only at the top level

e.g.: (removetop '(a (a b) c a) 'a) → ((a b) c)

```
(defun removeTop (list item)
  (cond     ((null list) nil)
            ((equal (car list) item)
                  (removeTop (cdr list) item))
            ( t
                (cons (car list) (removeTop (cdr list) item)))
  )
)
```

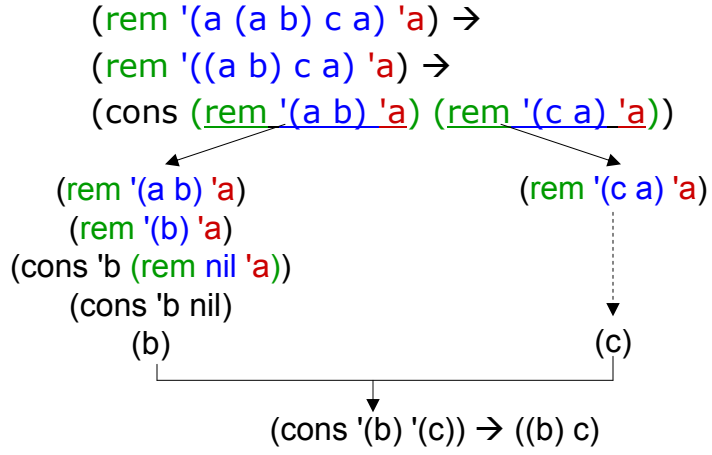

recursion example 2

remove **item** from **list** from all levels

e.g.: (rem '(a (a b) c a) 'a) → ((b) c)

```
(defun rem (list item)
  (cond ((null list) nil)
        ((equal (car list) item)
            (rem (cdr list) item))
        ((atom (car list))
            (cons (car list) (rem (cdr list) item)))
        (t
            (cons (rem (car list) item) (rem (cdr list) item)))
    )
  )
```

example 2 trace

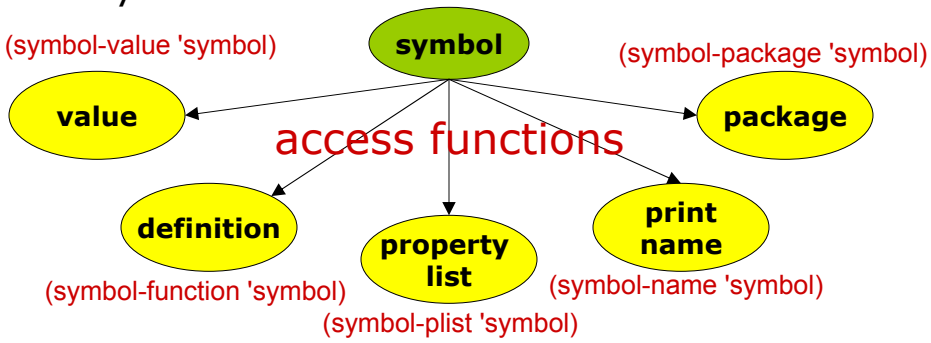


Lisp Symbols

Shakil M. Khan
adapted from Gunnar Gotshalks

symbols are more than variables

they have a complex structure – see notes on symbols



what is a property list?

- programs model the world as we see it
- entities with attributes (properties in Lisp) populate the world
  - entity book with attributes author, publisher, num of pages
  - attributes have values
    author Asimov publisher ACE books pages 412
  - Lisp models the above with
    symbol book
    property list
    (author Asimov publisher ACE books pages 412**)**

EBNF definition of property lists

- property lists are attribute-value pairs all at the same list level
- EBNF for property Lists

```
PropertyList ::= (nil, PropName PropValue
                       PropertyList) ;
PropName ::= any symbol ; // name of the property
PropValue ::= any s-expression ; // its value
```

- examples
  - (colour red size large)
  - (colour white
    change (penny 3 dime 4 looney 6 toonie 10))
- values can themselves be property lists

accessing properties

- use (**get** 'symbol 'propName) to access a property value for a symbol
  - assume the symbol purse has the property list
    (color white
    change (penny 3 dime 4 looney 6 toonie 10))
- then
  - (get 'purse 'change)
    returns the s-expression
    (penny 3 dime 4 looney 6 toonie 10)

setf

- (setf place form)
  - the s-expression for place is evaluated and results in a pointer to a location in memory
  - the s-expression for form is evaluated
  - the result is stored in the memory location indicated by place
  - more general than set and setq
  - if place is a symbol, then it is like setq
  - e.g.: (setq x '(a b c))
    (setf (car x) 20)
    x → (20 b c)

accessing properties (cont'd)

- use (**setf** (get 'symbol 'propName) value) to set a property value
  - (setf (get 'purse 'color) 'pink)
    changes the color of the purse to pink
- get returns the address of where the attribute value is or would be
  - hence new attribute-value pairs can be stored

getf and property lists

- (getf prop-list 'propName) accesses properties as well; the first argument is a property list
  - (setf (get 'purse 'color) 'pink)
  - (getf (symbol-plist 'purse) 'color) → returns pink
- property lists do not need to be associated with a symbol's plist
  - any list of attribute-value pairs will do
    - (setf x '(color blue change (penny 4 dime 5)))
      (getf x 'change) returns (penny 4 dime 5)
  - even just a property list structure will do
    - (getf '(color blue change (penny 4 dime 5)) 'change)
  - returns (penny 4 dime 5)

Lisp symbols vs. variables

- a change to a property is NOT local to a function definition
- properties are associated with symbols, not variables (recall that the same symbol can represent multiple variables in Lisp)

property list ambiguity

- if a property does not exist **get** returns **nil**
- what if a property value is nil?
- one solution: use an *optional* third argument to get
  - (get 'purse 'change 'unknown)
  - returns unknown if the property change does not exist
  - returns nil if the value of change is nil

property list ambiguity (another solution)

- good mathematical and programming practice is to give a special name for nil property values
  - could use (change none) in place of (change nil) or (color unknown)…
  - then nil would mean the property does not exists as opposed to the value of an existing property is nil

association lists

- like property lists
- associate attributes with values
- uses lists of lists
  - ((color black) (size large))
- first item of each sub-list is the property (key) and the second is the value

Control Over Evaluation and Function Application

functions as arguments (funcall, apply, eval), mapping functions (mapcar, maplist, reduce)

Shakil M. Khan
adapted from Gunnar Gotshalks

the problem

- suppose you want to write the following functions
  - **sum of the first n integers**

```
(defun sumN (n )
  (cond ( ( equal n 0 ) 0)
        ( t (+ n ( sumN (1- n )) )) ))
```

  - **sum of the squares first n integers**

```
(defun sumN2 ( n )
  (cond ( ( equal n 0 ) 0)
        ( t (+ (* n n ) ( sumN2 (1- n )) )) ))
```

  - **sum of the cubes first n integers**

```
(defun sumN3 ( n )
  (cond ( ( equal n 0 ) 0)
        ( t (+ (* n n n ) ( sumN3 (1- n )) )) ))
```

abstract the commonalities and only supply the variations

write one function for all cases

abstraction requires

- passing a unary function
  - identity for sum of integers
  - square for sum of squares
  - cube for sum of cubes

first attempt

- (defun square (n) (* n n))
  (defun sumInt (func n)
  (cond ((equal n 0) 0)
  (t (+ (func n)
  (sumInt func (1- n))))
  ))
  (sumInt 'square 5)
- but (func n) applies the function "func" on the argument n, not its value (i.e. square)

abstraction requires (cont'd)

- ability to evaluate the function
  - (funcall '+ 1 2 3) --> 6
    requires the parameters of the function to evaluate to appear at the level of the function
  - (apply '+ (1 2 3)) --> 6
    requires the parameters of the function to evaluate to be a list
  - note: these can't be used on special functions such as setq and defun

the abstraction

- using funcall
  - (defun sumInt (func n)
    (cond ((equal n 0) 0)
    (t (+ (funcall func n)
    (sumInt func (1- n))))
    ))
- using apply
  - (defun sumInt (func n)
    (cond ((equal n 0) 0)
    (t (+ (apply func (list n))
    (sumInt func (1- n))))
    ))

using the abstraction

- now we can define or use any unary function to obtain the sum of that function applied to the first N integers; eg.:
  - (defun double (int) (+ int int))
  - (sumInt 'double 10) --> 110
  - (defun square (int) (* int int))
  - (sumInt 'square 10) --> 385
  - (defun identity (int) int)
    - i.e. do nothing with the integer before summing
  - (sumInt 'identity 10) --> 55

more abstraction

- here is the original abstraction
  - (defun sumInt (func n)
    (cond ((equal n 0) 0)
    (t (+ (funcall func n)
    (sumInt func (1- n))))
    ))
- can abstract further
  - (defun funInt (binaryFunc unaryFunc n base)
    (cond ((equal n 0) base)
    (t (funcall binaryFunc
    (funcall unaryFunc n)
    (funInt binaryFunc
    unaryFunc (1- n) base)))
    ))

more abstraction (cont'd)

- now can do the following
  - (funInt '+ 'double 10 0) --> 110
  - (funInt '* 'double 10 1) --> 3715891200
  - (funint '* 'double 10 0) --> 0 ; why?
- too much abstraction make functions too complex
  - judgment and experience dictate when abstraction has gone too far

evaluating an s-expression

- want to write a function called "our-if"
  - (our-if test then-part else-part)
  - first attempt

```
(defun our-if (test then-part else-part)
          (cond (test then-part)
                    (t else-part)
 ))
(our-if (> n 50) (setq x 100) (setq x 0))
```

  - problem:
    - arguments to our-if are always evaluated – may have lasting effects
    - quoting them or using funcall doesn't work either

eval

- **eval** evaluates an s-expression
  - (setq x (cons '+ '(2 3))) → (+ 2 3)
  - (eval (eval x)) → 5
- funcall and apply are based on the function eval
- note: eval makes it possible to execute s-expressions created by a Lisp program
- provides the essentials for Artificial Intelligence
  - dynamically construct s-expressions which represent functions
  - dynamically execute them

our-if using eval

- (defun our-if (test then-part else-part)
  (cond (test (eval then-part))
  (t (eval else-part))
  ))
- (our-if (> n 50) '(setq x 100) '(setq x 0))

context problems with eval

- (defun our-if (test then-part else-part)
  (setq *n* 1)
  (cond (test (eval then-part))
  (t (eval else-part))
  ))
  (setq *n* 5)
  (our-if (> *n* 3) '(- *n* 3) '(+ *n* 3))
  x → ?
- need to be careful with eval and ensure that we are evaluating an expression in the intended context

mapping functions

- we saw how to pass a function to another function
- suppose we want to apply functions to a list of arguments, rather than just one argument
- Lisp provides useful abstractions
- these functions differ from one another in how they sequence through the argument list and what value they return

mapcar

- (mapcar function arg1 arg2 ... argN)
  - apply the function to the first of each of a list of arguments
  - recursively apply to the second of each argument, etc
  - collect all the results in a list
  - e.g.: (mapcar '+ '(1 2 3 4) '(10 20 30 40))
    → (11 22 33 44)
- ((function (car arg1) (car arg2) ... (car argN))
  (function (cadr arg1) (cadr arg2) ... (cadr argN))
  (function (caddr arg1) (caddr arg2) ... (caddr argN))
  (function (cadddr arg1) (cadddr arg2) ... (cadddr argN))
  ...)

maplist

- (maplist function arg1 arg2 ... argN)
  - apply the function to the arguments
  - remove the first item from each argument
  - collect all the results in a list
  - e.g.: (maplist 'list '(a b) '(x y))
    → (((a b) (x y)) ((b) (y)))
- ((function arg1 arg2 argN)
  (function (cdr arg1) (cdr arg2) ... (cdr argN))
  (function (cddr arg1) (cddr arg2) ... (cddr argN))
  (function (cdddr arg1) (cdddr arg2) ... (cdddr argN))
  ...)

reduce

- (reduce function list)
  - apply function to the first two items in the list
  - recursively apply to the result and the next item on the list
  - e.g.: (reduce '+ '(1 2 3 4)) → 10
  - computes (+ (+ (+ 1 2) 3) 4)
- (function
  ...
  (function
  (function
  (function (car list) (cadr list))
  (caddr list))
  (cadddr list))
  ...
  (cad...dr list))

reminder

- although I used (and the text uses) setq inside function definitions for illustration purpose…
- **do not** use setq within function definitions
- setq creates global symbols **NOT** local symbols
- **very poor** programming practice

additional course info

- my office hours (CSEB-2017):
  - Wednesday 3:00 PM – 4:30 PM
  - Friday 2:00 PM – 3:30 PM
- assignment 1
  - check course page after Friday
  - due in approx. 2 weeks