CSE-3401
Functional and Logic Programming

Course overview

Shakil M. Khan
http://www.cse.yorku.ca/~skhan
skhan@cse.yorku.ca
adapted from Yves Lespérance

purpose

- expand your understanding of key concepts in the evolution of programming languages
- learn to adapt to a new mind-set

  (actually two new mindsets!)
  - pure functional programming
  - declarative programming

"the main stream"

- Java, C, scripting languages are all built on the same idea:
  procedural, or operational semantics
- program describes operation of a machine as a sequence of steps:
  e. g. x = x + 1;

a road less travelled

- Lisp, J, Haskell, ML / Prolog, Goedel
- *denotational* languages
  describe what rather than how
  - program = set of mathematical expressions or statements
  - meaning (denotation) = mathematical object
    - example: y <= 2*y
    - denotation: complex relationship between x and y

denotational vs. procedural

- (defun equal (l1 l2)
  (and (eq (first l1) (first l2))
  (equal (rest l1) (rest l2))))
- while (n <= s1.length() & s1[n] == s2[n])
  n++;

another perspective: symbolic computing

- computing on non-numbers, non-character strings.
  - analogy to numbers and characters: *atoms* from atoms we build *symbolic structures*:
    lists, trees, terms, propositions, etc.
  - Symbols are used to describe,
    so symbolic programming has come to be programming that uses and creates *descriptions*.
  - Symbolic computing is often reflexive:
    a program is computed from a description.

functional vs. declarative programming

- *functional* programming uses mathematical functions and functional expressions to describe objects.
  - E.g.: Lisp
- *declarative* programming uses logical statements to describe objects.
  - E.g. Prolog

functional expressions

- examples

  x + 1
  - x + 1 denotes a numeric value

  y = x + 1
  - result is a logical value (true, false).
- Lisp notation:

  (+ x 1)

  (EQ y (+ x 1))

timelessness vs change of state

- denotational semantics uses mathematical language:
  - timeless propositions
  - nothing changes: *'x = x +1'* is false.
- operational semantics uses the language of state (memory) and change of state:
  - *'x = x + 1'* describes a change of state

in a nutshell

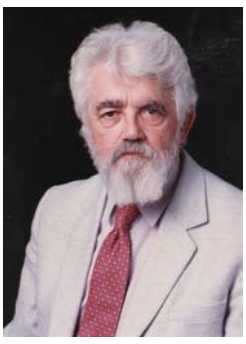
- we will investigate symbolic computation
- by looking at programming which manipulates symbols to specify what is to be computed, rather than how.
- using two "classic" languages
  - Lisp (MIT, 1959)
  - Prolog (Montréal, Marseille, 1973)

application areas

- semantic web & web services
- deductive databases, active databases
- agent programming languages, e.g. Golog, JACK
- workflow systems, modeling business rules, e.g. RuleML

a bit of background on Lisp

- created by the AI pioneer John McCarthy


major AI programming tool

- 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
- Guile/Scheme: dialect of Lisp, extension language for FSF products, e. g. Gimp

Lisp as an *extensional* language

- embedding Lisp makes it easy to extend an application
- creation of new languages built "on" Lisp

many versions

- available on all major and many minor platforms
  - all 1st year MIT computer students as well as students from about 95 other universities get their intro to CS through MIT Scheme.
- industrial-strength versions are usually standardized on Common Lisp

course info

- Website:

  http://www.cse.yorku.ca/course/3401
- Texts:
  - *Common LISPcraft*, R. Wilensky, 1986
  - *Programming in Prolog*, W. F. Closksin & C. S. Melish (5th ed.), 2004
- Evaluation:
  - 3 reports (6% + 7% + 7%) = 20%
  - 2 class tests (2 x 20%) = 40%
  - Final exam 40%

Basic Lisp

Shakil M. Khan
adapted from G. Gotshalks & Y. Lespérance

Lisp good points

- consistent structure for data and programs
  - **both are lists**
- clean design for 'pure' Lisp
  - Common Lisp not so clean – lots of operational features
- promotes modular programming through lots of small functions

Lisp bad points

- excessive use of parenthesis can make it difficult to understand
  - **L**ots of **I**nsignificant **S**illy **P**arenthesis
- prefix for all operators makes arithmetic clumsy
  - but for everything else matches procedure calls
- Lambda calculus underpinning can be difficult for beginners to understand

a general form for data

- Lisp uses different basis than conventional data structures
  - compound structures (arrays, strings) usually built from numbers, chars
- a more general approach
  - base all compound data structures on the **list**.

Lisp data structures

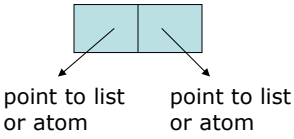
- atoms – essentially simple (number or word), but can have a complex internal structure
  - see notes on symbols
- lists – sequence of any kinds of objects (including lists)
  - recursive data structure (what's another example?)
  - E.g.: (a), (a b c)
    (a (b c) d), (a (((b))) c))

expressions made of symbols

- lists are not typed: not restricted to just numbers or just chars etc.
  - lists are not arrays -- why?
    list elements not referenced by an index
- lists are the main kind of symbolic expressions *(s-expressions)*
  - constructed from symbols and numbers.

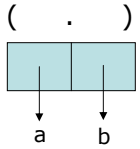
dotted notation

- most general notation
- underlying structure of lists:
  - actually binary trees
  - built using a constructor: . = 'dot'



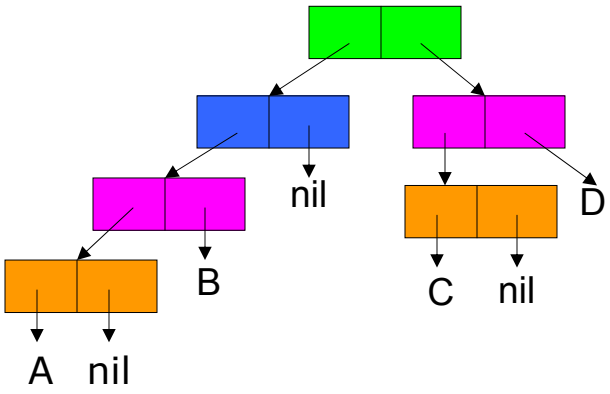
dotted notation (cont'd)

- parenthesis pair denote a cell with a dot separating the two parts of the cell
- an s-expression is an atom or dotted pair of s-expressions
  - note the recursive definition



(a . b)

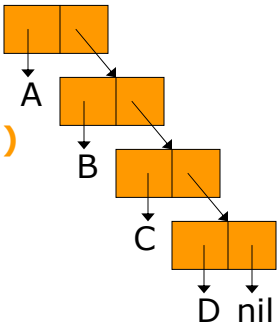
dotted notation example



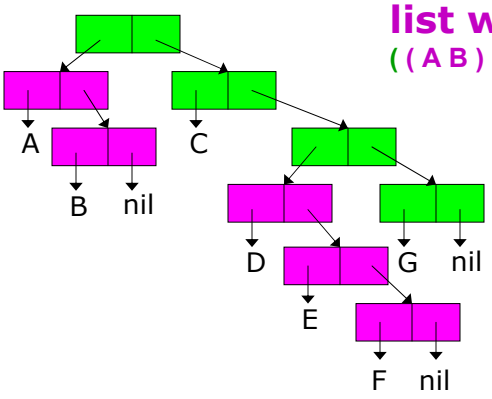
((((A.nil).B).nil).((C.nil).D))

s-expressions

- most common structure is a list
- simplify by removing the 'redundant' dots and parenthesis
  (A B C D)
- instead of
  (A . (B . (C . (D . nil))))



s-expressions – example 1

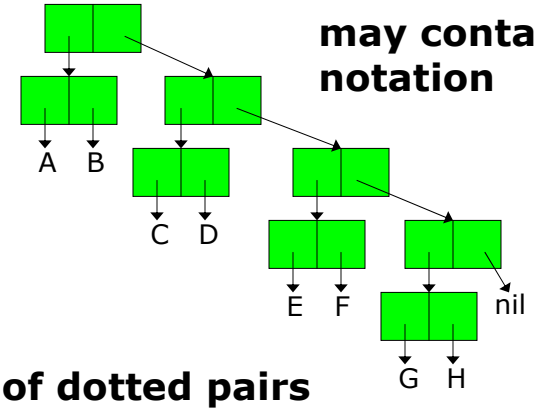


list with sublists

((A B) C (D E F) G)

((A . (B . nil)) . (C . ((D . (E . (F . nil))) . (G . nil))))

s-expressions – example 2



may contain dotted notation

list of dotted pairs

((A . B) (C . D) (E . F) (G . H))

s-expressions (cont'd)

- the empty list is
  nil = **( )**
- it is both an **atom** and a **list** !

dotted => s-expression

- apply the following rules from **right to left**
  1 replace . nil) with)
  - end of a list
  - (A . nil) ----> (A)

  2 replace . (...) with 'space' ...
  - end of a list
  - (A . (B . C)) ---> (A B . C)
  - what is the length of the above list?

example

Can apply rule 1 in three places

((A.(B.nil)).(C.((D.(E.(F.nil))).(G.nil))))

((A.(B)).(C.((D.(E.(F))).(G))))

Can apply rule 2 in three places

((A.(B)).(C.((D.(E.(F))).(G))))

((A B).(C.((D.(E F))G)))

Successive applications of rule 2

((A B).(C.((D.(E F))G)))

((A B).(C.((D E F)G)))

((A B).(C(D E F)G))

((A B)C(D E F)G)

Basic Lisp Operations

Shakil M. Khan
adapted from G. Gotshalks & Y. Lespérance

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

function invocation

- it is an S-expression – just another list!

  (function arg1 arg2 ... argN)
  - first list item is the function – **prefix notation**
  - the other list items are the arguments to the function.
  - arguments can themselves be lists
  - arguments are evaluated before the function is applied

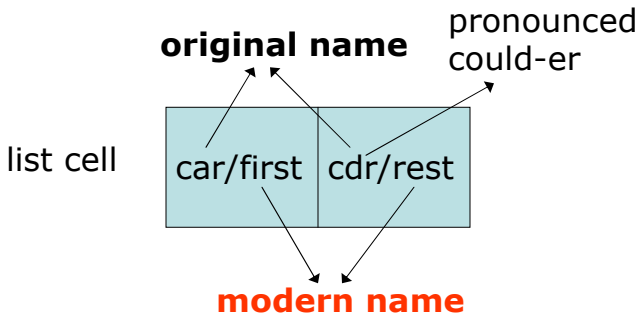
example

- (+ 1 2 3 (+ 4 5 6) 7 8 9) ==> 45
  - outer + has 7 arguments, inner + has 3 arguments
  - eval(1) = 1
  - …
  - eval(+ 4 5 6) = (+ eval(4) eval(5) eval(6))
    = (+ 4 5 6) = 15
  - eval(+ 1 2 3 15 7 8 9) = 45

basic functions

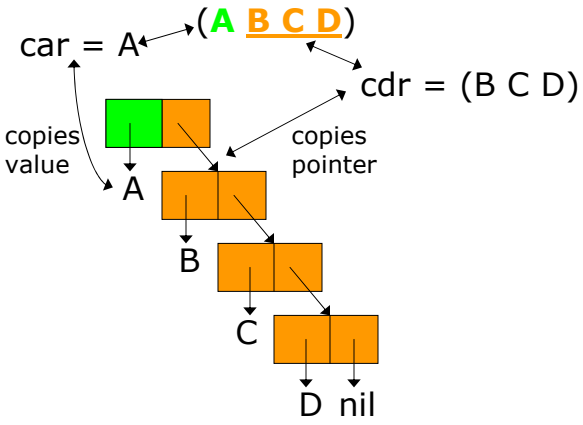
- list access & creation
  - car or first – access first in list
  - cdr or rest – access all but first
  - cons – construct a list cell
- other
  - quote or ' – take literally, do not interpret
  - atom – true if argument is an atom
  - eq – true if arguments are same object

list access functions

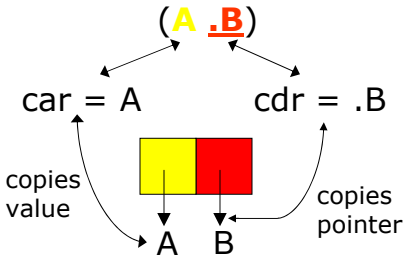


(car '(a b c)) = (first '(a b c)) = a
(cdr '(a b c)) = (rest '(a b c)) = (b c)

car/cdr structural view 1



car/cdr structural view 2

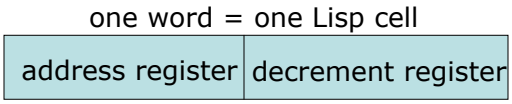


(car '(a b c)) – why the quote?

- recall that arguments are evaluated before the function
- if we wrote – **(car (a b c))**
  - argument (a b c) would be evaluated before the car is applied
  - a would be a function call
  - but we literally want the list (a b c) not the result of evaluating a on the arguments b and c.
- **'(…)** is syntactic sugar for **(quote …),** a special function whose arguments are not evaluated
  - (car '(a b c)) = (car (quote (a b c)))

why the names car/cdr?

- original Lisp developed for an IBM 704/709 computer which had 18 bit registers
- pairs of registers could be handled as a single 36 bit 'word'



CAR = **C**ontents **A**ddress **R**egister
CDR = **C**ontents **D**ecrement **R**egister

short hand for nested car's and cdr's

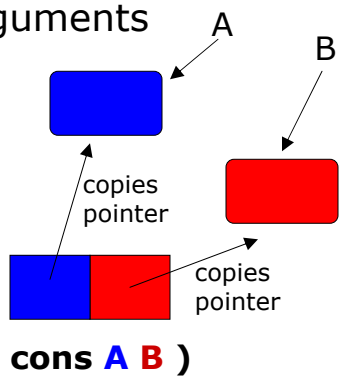
- accessing deeper into Lisp structures occurs so frequently that additional functions are introduced into Lisp.
- for example

  (c<u>ddda</u>r ...) = (cdr (cdr (cdr (car ...))))

  - interpret from right to left
  - length depends upon the implementation

creating a new Lisp cell

- only one constructor function : **cons**
- copies pointers to the arguments
- laws
  - **(car (cons A B)) = A**
  - **(cdr (cons A B)) = B**



(cons A B)

destructive List Construction

- cons is expensive as it creates a new cell
  - memory allocation is invoked
- but it is non destructive – no side effects
- following is dangerous – do **NOT** use!
  for efficiency Common Lisp provides a set of destructive operations – they change lists
  - (replca cell newValue) & (replcd cell newValue)
    - replace the car and cdr fields of cell with pointers to newValue
  - (nconc x y)
    - replace the cdr field of the last component of x with a pointer to y

setq – define a symbol value

- (setq x value)
  - if the symbol x does not exist it is created
  - symbol x is given the value value
- in this course USE ONLY AT THE GLOBAL LEVEL to create symbols required to test your programs
- eg.: (setq x '(1 2 4)) sets the value of x to the list (1 2 4)
  - note the x is not quoted but the second argument is if you do not want to evaluate it.

compare set and setq

- (setq x 'y)
  - x has the value y
- (set x 'z)
  - x still has the value y
  - but a new symbol y is created with the value z
  - why?
- See the notes on symbols

defun – define a function

- (defun functionName (argumentList)
  list of s-expressions to evaluate when the function is invoked – usually only one s-expression
)
  - eg. (defun add (a b) (+ a b))
  - value of the function is the value of the last s-expression that is executed
- functions in Lisp are typically small
  - rarely more than 1/2 a page in length

Basics of Using Lisp

Shakil M. Khan
adapted from Gunner Gotshalks

getting into and out of Clisp

- entering **% clisp**
- do Lisp work
- exiting **(bye)**
  - a function with no arguments
  - CTRL–d can also be used

do Lisp work

- edit in files with extension ".lsp"
- load the files
  (load "filename.lsp")
  - > Loading executes the S-expressions in the file
  - > Loading defines symbols
  - > No other computational effects are seen
  - > Comments begin with " ;"
- interactively execute S-expressions
  - define and invoke functions
  - use setq to define symbol values

the Lisp interpreter

- …is a loop over the following functions
  - **read**
    - an s-expression
  - **eval**
    - the s-expression
  - **print**
    - the result
- you can redefine these functions !!!
  - but it is dangerous if you do not know what you are doing