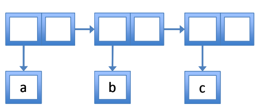
Programming Languages and Paradigms
COMP 3109

Lecture 3: Functional Programming II

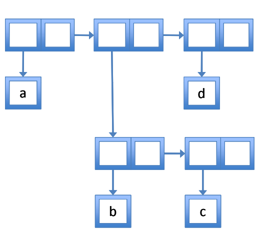
Internal Representation of Lists

- Binary-tree like structure for lists
  - An inner node is a ***cell***
  - A cell has a left and right reference denoted as ***car*** and ***cdr***
  - Left and right reference might point to ***nil***
  - Leafs are atoms (numerals, strings, symbols, etc.)
- Example: ‘(a b c)



Internal Representation (cont’d)

- Example: ‘(a (b c) d)



Construction of Cells

- Function (***cons*** <car> <cdr>) returns a new cell
- Construction of lists
  - List '(a b c d ...) is constructed as (***cons*** 'a (***cons*** 'b (***cons*** 'c (***cons*** 'd (...)))))
- Short form of cell construction
  - Example: '(a . b) is equal to (***cons*** 'a 'b)
  - Note above example is not a list because ***cdr*** is pointing to an atom!
- Short form of list construction: (***list***<elem$_1$> ... <elem$_n$>)
- Example
  - ➢ (***list*** 'a 'b 'c)

    (a b c)
- Merge lists: (***append***<list$_1$> ... <list$_n$>)
- Example
  - ➢ (***append*** '(a b) '(cd) '(ef))

    (a b c d e f)

Programming Languages and Paradigms

Group Exercise

- Visualize the binary trees of following terms

  '(a b (c d (e f) g) (h i))

  '(a . b)

  '(a . (c . (d . nil)))

  '((a . c) (d . e))
- Find simpler notations for the terms above

Programming Languages and Paradigms

List Access

- Access first element: (***first***<list>) or (***car***<list>)
- Example
  - ➢ (***first*** '(a b c))

    a
- Access remaining list without first element (***rest***<list>) or (***cdr***<list>)
- Example
  - ➢ (***rest*** '(a b c))

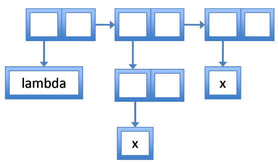
    (bc)
- Accessing nth element (using zero-based counting)
  - ➢ (***nth*** 2 '(a b c))

    c
- Accessing the last element of a list
  - ➢ (***last*** '(a b c))

    c

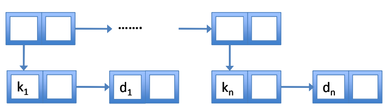
Remark

- LISP is homo-iconographic
  - Functions are represented as lists as well
- Example: ‘(lambda (x) x)



Association Lists

- Association lists store set of keys with data
  ‘((k_1 d_1) … (k_n d_n))



- Example
  '((hat object) (monkey mammal) (parrot bird)))

Access Function

- Retrieve association: (***assoc*** <key> <alist>)
- Examples
  - (***assoc*** ‘cars ‘((cars fast)(horses slow)))
    (cars fast)
  - (***assoc*** ‘trains ‘((cars fast)(horses slow)))
    nil
  - (cdr (***assoc*** ‘cars ‘((cars fast)(horses slow))))
    fast

Group Exercise

- Write your own ***assoc***

Copy / Add New Entries

- Copy association list: (***copy-alist*** <alist>)
  - Deep copy of <alist>
- Add a new pair:
  - (append alist '((key data)))
- Example
  - (***setq*** vehicles '((cars.fast)(horses slow)))

  ((cars fast) (horses slow))
  - (***setq*** vehicles2 (append (copy-alist vehicles) '((train fast))))

  ((cars fast)(horses slow)(train fast))

Equalities

- Identity
  - two terms reference the same object
  - Example: (***eq*** 'x 'x), (***eq*** 412 412)
- Atom Equality
  - two strings or numbers atoms are equal
  - (**eql** 10 10), (**eql** x x)
- Numerical Equality
  - two numbers (might have different type) are equal
  - (= 100 100.0)
- Structural Equality
  - two lists have the same structure and equal atoms
  - (equal (list 1 2) (list 1 2)) is true whereas
  - (eq (list 1 2) (list 1 2)) is false!

Group Exercise

- Write your own definition of equal
  - Assume there are no cyclic list structures

Eval/Quote

- Delayed evaluation with quote and eval
- Quote: no rewrite rules are applied inside the argument of quote
- Eval: forces the evaluation of a symbol
- Example
  - ➢(***setq*** a (quote (+ 1 2)))
  - (+ 1 2)
  - ➢(***eval*** a)
  - 3

Function Application

- Format of apply: (***apply*** <fn> (<args>))
- Example:
  - ➢(***apply*** ‘(lambda (x y) (+ x y) ‘(3 4))
  - 7
- Format of funcall: (***funcall*** <fn> <a_1>…<a_n>)
- Example:
  - ➢(***funcall*** ‘(lambda (x y) (+ x y)) 3 4)
  - 7

Evaluation order of arguments

- Applicative order
  - Evaluate arguments first, then apply function to evaluated arguments
  - what you're used to in imperative languages
  - usually faster
- Normal order (aka. Lazy evaluation)
  - Expand out function definition first
  - like call-by-name: don't evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser property)

Example / Lazy Evaluation

- Example

```
(defun endless-loop ()
   (endless-loop))
(defun oops (x y)
   x)
(oops 0 (endless-loop))
```

- Applicative-order
  - Args 0 and (endless-loop) evaluated before test call
  - Causes endless loop
- Delayed evaluation
  - Program terminates because endless-loop is not evaluated

Lazy Evaluation in LISP

- Arguments are encapsulated in quote
- Use of parameters in function body need an ***eval*** to force evaluation.
- Terminating example:

```
(defun endless-loop ()
   (endless-loop))
(defun happy (x y)
   (eval x))
(happy (quote 0) (quote (endless-loop)))
```

Programming Languages and Paradigms

Lazy Evaluation in Scheme

- Scheme provides delay/force functions
- Example

```
(define naturals
  (letrec ((next (lambda (n)
       (cons n (delay (next (+ n 1)))))))
(define head car)
(define tail (lambda (stream)
    (force (cdr stream))))
(head naturals) => 1
(head (tail naturals)) => 2
```

Programming Languages and Paradigms

Higher Order Functions

- Higher-order functions
  - Take a function as argument, or return a function as a result
  - You are able to write highly compressed code
- Example
  - Apply a function to elements of a list and the result is the result of each function application
  - (***mapcar*** ‘(lambda (x) (+ x 1) ‘(1 2 3 4))

(2 3 4 5)

Programming Languages and Paradigms

Definition of ***mapcar***

- Recursive Definition

```
(defun mapcar (f l)
   (cond ((null l) nil)
         (T (cons (funcall f (car l))
                  (mapcar f (cdr l))))))
```

Binary to Unary

- Convert a binary function to a nested unary
```
(defun bu(f x)
   (function(lambda(y)(f x y))))
```
- Constructs an unnamed function with a single argument
- Computes the result of applying f to x and y
- Example
  - (***bu*** #’ sum 1) returns a function that adds one to its argument if (sum xy) adds two numbers.

Continuations in Functional Languages

- Additional control flow technique for functional languages
  - Implements a goto in functional languages
  - It is a dynamic goto (not static!)
  - Similar to C's setjmp/ longjmp
  - Somehow related to exceptions (throw/catch)
- Scheme implements call/cc
  - Captures current closure
  - Passes it own to its argument
  - Closure can be invoked to return to call context

Scheme Example

- Continuation Example
```
(define (find p l)
 (call/cc
   (lambda (return)
     (for-each (lambda (e)
        (if (p e)
            (return e)))
        l)
     #f)))
```
- Searches a list and terminates if element is found
- Termination is triggered by inner function
- Value #f is returned if for-each cannot find element
- Allows to escape from a deeply nested function call
  - i.e. exit continuation
  - Continues computation at call/cc call

Continuations (cont'd)

- Full continuation
  - Resume computations though function is already exited
- Example:

```
➢(define ret #f)
➢(+ 2 (call/cc
        (lambda (c)
          (set! ret c)
            1)))
3
➢(ret 23)
25
```


Functional Outlook

- LISP is the first functional PL
- Other dialects
  - Pure (original) Lisp
  - Interlisp, MacLisp, Emacs Lisp
  - Common Lisp
  - Scheme
- What is there else?

Typed Lambda Calculus

- Types are introduced
  - i.e. functions have signatures
- Modern functional PL are typed
- Examples
  - Haskell
  - ML
  - OCAML

Haskell

- First version of Haskell introduced in 1990
- Emerged from language called Miranda
- Two wide-spread implementations
  - Hugs
  - GHC
- Language has
  - types and function signatures
  - pattern matching
  - guards
  - currying
  - algebraic data types
  - lazy evaluation
  - monads
  - type classes

Programming Languages and Paradigms

Function Definition in Haskell

- Square a number
```
square :: Int ->Int
square = n * n
```
- Function **square** needs a signature
  - i.e., specification of the domain and range of the function
- Addition of two numbers
```
add :: Int ->Int ->Int
add n m = n + m
```
- Expression types of add, add 2, add 2 3:
```
add :: Int ->Int ->Int
add 2 :: Int ->Int
add 2 3 :: Int
```

Programming Languages and Paradigms

More Examples in Haskell

- Two arrays a and b: assign b all elements whose values are less than 101
```
a :: [Int]
b :: [Int]
b = [n | n<-a, n<= 100]
```
- Equal to set notation in maths
  - $b = \{n \mid n \in a \wedge n \leq 100\}$
- Quicksort in Haskell
```
qs:: [Int] -> [Int]
qs[] = []
qs(x:xs) = qs [y | y<- xs, y<=x] ++ [x] ++ qs[y | y<- xs, y>x]
```
- qs(x:xs) splits argument array into single element x and remaining array xs
- [y | y<- xs, y<=x] describes an array whose elements are smaller than or eual to x
- [y | y<- xs, y>x] describes an array whose elements are greater than x
- a ++ b ++ c concatenates arrays a, b and c
- Three lines of code vs. 2 pages of code in an imperative language!

More Examples in Haskell

- Pattern matching capabilities
  ```
  fac :: Integer -> Integer
  fac 0 = 1
  fac n | n> 0 = n * fac (n-1)
  ```
- Base and recursive case is spelled out
- Recursive case has a guard (| n> 0) to allow definition only for positive numbers