Computing Science (CMPUT) 325

Nonprocedural Programming

Department of Computing Science
University of Alberta

Higher-order Functions

- Definition of higher order function:
  - Takes other function(s) as input, and/or
  - Produces function(s) as output
- Often used to separate a computation pattern from the specific action
- Example: iterate over a list, and
  "do something" with each list element
- Example: reduce a list to a single result
  by repeatedly applying the same computation
  (e.g. add number to a sum)

Higher-order Functions - Lecture Plan

- Look at some typical higher-order functions
- Start defining them in Fun
- Then look at Lisp language support,
  and Lisp implementations

Some Typical Higher Order Functions

- Map
- Reduce
- Filter
- Vector

Map

- Apply a given function to each element in a list
- Collect all the results in a list
- Example: salary raise
- Inputs: payroll, and a function implementing the raise
- Output: the new payroll with increased salaries

Salary Raise Example (continued)

- Payroll representation:
- List of dotted pairs (name . salary)
- Example: ((John . 23000) (Mary . 50000) ....)
- Raise salary: increase every salary by $100
- In Fun: `inc(E) = cons(car(E), cdr(E) + 100)`
- Example: `inc((Mary . 50000))`
  `= (Mary . 50100)`

Salary Raise Example (continued)

- Given list L, call inc for each element and collect the results
- In Fun:

```
raise(L) = if null(L) then nil
           else cons(inc(car(L)),
                     raise(cdr(L)))
```

- We have seen many similar examples, it gets tedious
- Let's separate:
  - The iteration over the list
  - What is done to each element

Map Function

- Idea: write a function `map` that:
- Can take any function `f` as an argument
- Applies `f` to every element in a given list
- Returns the list of results

Map in Fun

```
map(f,L) = if null(L) then nil
           else cons(f(car(L)),
                     map(f, cdr(L)))
```

- Example: `map` for the salary raise problem

```
map(inc, ((John . 23000) (Mary . 54560)))
```

Map Example

- Main computation steps:
  `map(inc, ((John . 23000) (Mary . 54560)))`
- $\longrightarrow$ `cons((John . 23100),`
  `map(inc, ((Mary . 54560))))`
- $\longrightarrow$ `cons((John . 23100),`
  `cons((Mary . 54660), map(inc, NIL)))`
- $\longrightarrow$ `cons((John . 23100),`
  `cons((Mary . 54660), NIL))`
- $\longrightarrow$ `((John . 23100) (Mary . 54660))`

Mapcar in Lisp

- In Common Lisp, the built-in map function is named `mapcar`
- Example:

```
* (defun plus1 (x) (+ x 1))
PLUS1
* (mapcar 'plus1 '(1 2 3 4 5))
(2 3 4 5 6)
* (defun inc (N)
        (cons (car N) (+ 100 (cdr N))))
INC
* (mapcar 'inc '((John . 23000) (Mary . 54560)))
((JOHN . 23100) (MARY . 54660))
```

Reduce

- Idea: in a list, repeatedly apply the same function to two arguments, which produces a single argument
- Given a function `g`, its identity `id`,
  and a list `L = (A1 A2 ... An)`:
- Compute `(g A1 (g A2 ... (g An id) ... ))`
- Example: sum of a list of numbers, using function `+` and identity `0`
- Example: product of a list of numbers, with function `*` and identity `1`

Reduce in Fun

```
reduce(f, id, L) =
    if null(L) then id
    else f(car(L),
           reduce(f, id, cdr(L)))
```

```
Example:
reduce(*, 1, (2 6 4))
-> (* 2 (* 6 (* 4 1)))
-> (* 2 (* 6 4))
-> (* 2 24)
-> 48
```

Payroll Example

- Goal: get the total payroll after the raise
- How about: `reduce(+, 0, map(inc, L))`
- with `L` bound to the pay roll list
- This is incorrect. Why?

Fixing the Example

- `reduce +` needs a list of **numbers**
- Payroll is a list of dotted pairs
- Idea: use `map` to get the list of numbers
- `reduce(+, map(cdr, map(inc, L)))`
- In Lisp, this would be written
  `(reduce '+ (mapcar 'cdr (mapcar 'inc L)))`

Reduce Without Identity

- The identity is not needed if we define reduce as
- `(g A1 (g A2 ...  (g An-1 An) ...  ))`
- The Lisp built-in `reduce` works this way by default
- Exercise: Define your own version `myreduce` that takes two arguments.

```
* (reduce '* '( 2 6 4))
48
* (reduce 'append '((a b) (c) (d e)))
(A B C D E)
```

MapReduce for Big Data

- Imagine huge amounts of data
  spread over many disks (or distributed memory)
  - Examples: transaction records, web pages, image databases
- Imagine a statistical query on all that data
  - Examples: find all overdrawn accounts,
    find all webpages containing the word “Lisp”,
    find all images of cats
- Map: apply the same operation to each data item
- Reduce: compute summary statistics, or sort and present the top 10 pages, ...

MapReduce in the Real World

- Several large scale implementations exist, e.g. in MPI, or Hadoop
- Provides an easy API for using large clusters
- The aim is to hide the complexity of the distributed computing support
- Performance can be worse than specialized database technology
- Good for quick, “one shot” tasks

Filter

- Goal: Select all elements from a list which satisfy a given test predicate
- In Fun:

```
filter(Pre, L)
= if null(L) then nil
  else if Pre(car(L)) then
     cons(car(L),
          filter(Pre, cdr(L)))
  else
     filter(Pre, cdr(L))
```

- Example: think of internet search as applying a filter to a list of all web pages

Vector

- Apply a list F of functions to an object x and get the list of all results of the applications.

```
vector(F, x) =
    if null(F) then nil
    else cons(car(F)(x),
              vector(cdr(F), x))
```

Defining new Higher Order Functions in Lisp

- Why define your own higher order functions?
- Use case: a common computation pattern, where the details (e.g. function to apply to each element in a list) can vary
- In terms of software engineering, this means removing code duplication
- **Remember: All code duplication is evil!**
- How to define new higher order functions in Lisp?

- Consider language implementation
- In most languages, it is clear what is a function call, and what is not
- In Lisp, no such “syntax barrier” between code and data
- Not easy for Lisp to figure out which s-expr is really a function call within a higher order function
- Lisp builtin functions `apply` and `funcall` tell Lisp that a function application is meant.

`apply` and `funcall`

- Using `apply` and `funcall` tells Lisp that there is a function to be called
- These two built-in functions have the same functionality but differ in syntax
- In practice, choose whichever one is more convenient

```
(apply function_name (arg1 ... argn))

(funcall function_name arg1 ... argn)
```

Side Note - Creating a List of Arguments

- `apply` expects a list with all arguments
- The simplest way to create it is using the built-in function `list`
- `(list arg1 ... argn)`
  $\Rightarrow$ `(arg1 ... argn)`
- Next slides: examples, re-implement some of the higher order functions above

Re-implement `mapcar` with `funcall` and `apply`

```
(defun xmapcar1 (f L)
      (if (null L)
           nil
           (cons (funcall f (car L))
                  (xmapcar1 f (cdr L)))
      )
)

(defun xmapcar2 (f L)
      (if (null L)
           nil
           (cons (apply f (list (car L)))
                  (xmapcar2 f (cdr L)))
      )
)
```

Re-implement `reduce` and `filter`

```
(defun xreduce (f L Id)
     (if (null L)
          Id
          (funcall f (car L)
                         (xreduce f (cdr L) Id))
)    )

(defun filter (Pre L)
   (if (null L)
     nil
     (if (not (eq (funcall Pre (car L)) nil))
       (cons (car L)
             (filter Pre (cdr L)))
       (filter Pre (cdr L))
) ) )
```

Built-in Lisp Functions

- Lisp built-in `remove-if-not` is same as our `filter`
- Yes, there is also a built-in `remove-if`

```
* (remove-if  'atom '(a b (c)))
((C))
* (remove-if-not 'atom '(a b (c)))
(A B)
```

Summary and Outlook

- Looked at higher-order functions in Lisp
- So far: functions that take other functions as input
- Standard examples map, reduce, filter
- Lisp support for writing new higher-order functions: `apply` and `funcall`
- What about producing functions as output? (see next slide)

Outlook

- In principle, we can now write a higher-order function that produces a function definition as its output
- In practice, it is simpler to work with so-called **lambda functions**
- Lambda functions are “functions without a name”