Functional Programming
Scheme Basics
Week 5

Dr. John Moore
moorejo@uwl.ac.uk

22nd February 2015

Working with data

- We have seen how to process lists with recursion
- We have seen how we can also use higher order functions to work with lists
- Scheme is based on Lisp which is all about list processing
- We will look at some other useful list creation and manipulation techniques
- We will introduce hash tables to give you another useful data structure to use
- We will combine hash tables with lists to make applications

List creating functions

```
(define (make-train name
                    departs
                    carriages
                    carriage_capacity)
  `((name ,name)
    (departs ,departs)
    (carriages ,carriages)
    (carriage_capacity ,carriage_capacity)))
```

We have defined a function which takes 4 arguments. We then use values to construct a list. You can think of this function as a template for creating lists containing train information.

Working with data

- We can keep using this function with new data to create more entries
- You should see I have introduced some new syntax to create the lists
- The first thing to note is the list does not begin with the usual quote
- If you look closely you will see we use ‘ instead of ’
- This symbol is known as quasiquote and allows us to pass the values of variables into the list
- We use , to indicate where the variables are
- We call this comma unquote

List creating functions

```
(define gordon
  (make-train "Gordon" "10:00" 5 20))
(define thomas
  (make-train "Thomas" "09:00" 10 40))
(define edward
  (make-train "Edward" "11:32" 7 25))
(define henry
  (make-train "Henry" "10:07" 2 50))

(define trains (list gordon thomas edward henry))
```

We can now supply data to the function several times and collect the result in a new list.

List creating functions

```
(((name "Gordon")
  (departs "10:00")
  (carriages 5)
  (carriage_capacity 20))
 ((name "Thomas")
  (departs "09:00")
  (carriages 10)
  (carriage_capacity 40))
 ((name "Edward")
  (departs "11:32")
  (carriages 7)
  (carriage_capacity 25))
 ((name "Henry")
  (departs "10:07")
  (carriages 2)
  (carriage_capacity 50)))
```

- We have pretty-printed the result
- the pretty-print function is very useful when working with nested list data
- We can observe a number of properties about the list we just created
- We have a list of length 4
- Each list item is another list containing more lists
- At this stage you should be able to extract data from these lists

Retrieving data from lists

- Having created a nested list structure it would be useful to be able to easily search and access parts of it
- We can of course use the techniques we have learnt already such as traversing the list with recursion
- However, it would be nice to be able to easily retrieve individual entries in the list based on some search data
- We can achieve this using associaton lists

Retrieving data from lists

```
(assoc '(name "Edward") trains)

;; -> ((name "Edward") (departs "11:32")
      (carriages 7) (carriage_capacity 25))
```

The list is searched sequentially to see if we can match against the first item in that list. If we get a match that list is returned.

Retrieving data from lists

```
(assq 'departs (assoc '(name "Gordon") trains))

;; -> (departs "10:00")
```

We can also apply the same technique to the previous list returned to obtain a more refined answer. In this case we use assq instead of assoc because we are matching against symbols. Look up the differences between eq, eqv, equal to learn about these subtle differences.

Retrieving data from lists

- Association lists are convenient and easy to use
- Because they work sequentially they operate in $O(n)$ time
- Another very convenient data structure to use is a hash table
- Hash tables are conceptually very simple
- We have a key which retrieves some data which can be anything from a symbol to a complex nested list
- Hash tables will be more scalable as they are designed to operate in $O(1)$ time

Working with hash tables

```
(define db (make-hash))
(hash-set! db 'name "John")
(hash-set! db 'name "John Doe")
(hash-ref db 'name)
(hash-remove! db 'name)
(hash-ref db 'name)
```

First we create a hash table. We then store the name “John” using the key 'name. We however, overwrite this entry with “John Doe”. We retrieve the entry based on the key. We remove the entry so we can no longer retrieve it based on the key.

Let's work through an application

Some useful steps to consider

- Get your code working
- Improve your code by...
- Improving structure
- Improving readability
- Improving performance

Getting started

```
(define db (make-hash))

(define (add-new-student id student)
  (hash-set! db id student))

(define s1
  '((fname "john") (sname "smith") (age 21)))
(define s2
  '((fname "joe") (sname "blogs") (age 22)))

(add-new-student 100 s1)
(add-new-student 101 s2)
```

Get it working

```scheme
(define (get-student-v1 id field)
  (cond
    ((eq? field 'age)
     (cadr (assq 'age (hash-ref db id))))
    ((eq? field 'fname)
     (cadr (assq 'fname (hash-ref db id))))
    ((eq? field 'sname)
     (cadr (assq 'sname (hash-ref db id))))))

(get-student-v1 100 'sname)
;; -> "smith"
```

- A first attempt
- Some repetition
- No error handling

First attempt to improve

```
(define (get-student-v2 id field)
  (let ((record (hash-ref db id)))
    (case field
      ((age) (cadr (assq field record)))
      ((fname) (cadr (assq field record)))
      ((sname) (cadr (assq field record)))
      (else "wrong field type"))))

(get-student-v2 100 'sname)
;; -> "smith"
```

- A second attempt
- Removed some repetition
- Some error handling

Second attempt to improve

```
(define (get-student-v3 db id field)
  (if (hash? db)
    (let ((record (hash-ref db id)))
      (if (memq field '(age fname sname))
        (cadr (assq field record))
        "wrong field type"))
    "not valid hash table"))

(get-student-v3 db 100 'sname)
;; -> "smith"
```

- Third attempt
- Removed more repetition
- Better error handling
- Not perfect though!

Things to try

- Look over the solutions from last week
- Add missing error handling to last example
- Based on the student record example, create a video library. Try to have a good selection of fields. See if you can create a function for loading the list data into the hash table.
- Based on the train data given earlier, create a list of train capacities and obtain the total capacity of all trains