Elements of Functional Programming

Abelson & Sussman & Sussman chapter 1.1

APP - Part A - A deeper lo ing

- In this part of APP we use the language Scheme to explore:
  - Functional programming
  - Programming in general
- We can only scratch the surface but we will look at:
  - Functional programming in Scheme.
  - Building data abstractions.
  - Modularity, Objects and state.
- These topics correspond to chapters 1, 2 and 3 of the textbook (over 350 pages!!).
- We will skip some sections (phew!).
  - But the skipped parts still make interesting reading.

Getting started with Scheme

- Scheme is an interpreted language.
- To start the interpreter type:

```
scheme
```

- The following prompt should appear

```
1 ]=>
```

- The scheme interpreter is now waiting for us to type in some expressions.
- *NOTE:* this shows use of MIT Scheme (used in SICP), as installed on CATS computers – we will also show you DrRacket, which you can install on your own machine

Expressions in

- Scheme is a (mostly) functional languages.
- As with all functional languages computation consists of the evaluation of expressions.
  - e.g. (+ (* 4 3) (- 6 4)) => (+ 12 2) => 14
- Note that all Scheme functions are prefix.
  - keeps things simple but you have to draw lots of brackets.
- To get Scheme to evaluate an expression you just type that expression in:

```
1 ]=> (+ (* 4 3) (- 6 4))
```

- Gives:

```
;Value: 14
```

If you get into trouble

- as with all interpreters it is possible to get into trouble:

```
(+ 3 true)
error in eval loop
2 error>
```

- This is Scheme asking for debug commands.
- To get back to the normal prompt type Control-C (twice)
- To get out of the system entirely type control-D at the normal prompt

```
End of Input Stream reached
  Happy Happy Joy Joy
```

- Scheme likes that.

Definitions in Sche

- Scheme has a rich set of primitive operations.
  - If all we could do is evaluate expressions containing these operations then scheme would be a sophisticated calculator.
  - Fortunately, as with most languages, we can extend this set of operations.
- Scheme also lets programmers add their own operations
- in a file test.scm:

```
(define size 2)
```

- at our scheme prompt

```
(load "test.scm")
...
size
2
```

More definitions in scheme

- Some more definitions:

```
(define pi 3.14159)
(define radius 10)
```

- We can write expressions in terms of our definitions:

```
(* pi (* radius radius))
```

- We can write other definitions in terms of our definitions:

```
(define circumference (* 2 (* pi radius)))
```

- Upon evaluation

```
circumference
62.8318
```


Defining fun

- Define can also be used to define functions:

```
(define (square x) (* x x))
(define (sum-of-squares x y)(+ (square x)(square y))
```

- at the command prompt

```
(sum-of-squares 3 4)
25
```

- More definitions

```
(define (first x y) x) ;fn returning its first arg
(define (const0) 0)    ;a constant function
```

- At the prompt

```
(first (const0) 3)
0
```

More language elements

- For a language to be complete, a programmer must be allowed to specify:
  1. a sequence of computations.
  2. conditional computation.
  3. repetitive computation.
- We have already seen how Scheme supports the first of these, indirectly, through evaluation of simple expressions.
- Scheme supports the second using primitives `if` and `cond`
  - explained next slide.
- Scheme supports the third using recursive definitions
  - explained after that.

Conditional Exp

- Scheme provides an if expression:

```
(define (abs x)
   (if (< x 0)
       (- x)
       x))
```

- And a cond expression:

```
(define (abs x)
   (cond ((< x 0) (- x))
         ((= x 0)   x)
         ((> x 0)   x)))
```

- Alternatively:

```
(define (abs x)
   (cond ((< x 0) (- x))
         (else     x )))
```

Conditional expressions

- Conditional expressions are supported by the usual boolean operators:
  - and, or and not
- Examples:

```
(and (< 3 4) (= 2 3))
(or (and (< 3 4) (= 2 3)) (= 5 5))
(define (>= x y) (or (> x y) (= x y)))
(define (>= x y) (not (< x y))
```


Repetiti

- In Scheme, repetition is expressed using recursive definitions.
- Examples:

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1)))))
(define (log2 x)
  (if (< x 2)
      0
      (+ 1 (log2 (/ x 2)))))
```

Substitution model of evaluation(A&S 1.1.5)

- Functions are applied to their arguments.
- To do this:
  1. Arguments are evaluated
  2. Arguments are substituted into the function body
  3. The function body is evaluated.
- Example:

(sum-of-squares 4 3) =>

(+ (square 4) (square 3)) =>

(+ (* 4 4) (square 3)) =>

(+ 16 (square 3)) =>

(+ 16 (* 3 3)) =>

(+ 16 9) => 25

Applicative vs Normal

- Scheme expressions are (by default) evaluated in applicative order.
- That is: leftmost-innermost
  - Most programmers are familiar with this order.
- There are other evaluation orders.
- Normal order is an important one: leftmost-outermost
  - Example of this on next slide.
- Is order important?
  - In functional programs the result is not affected by evaluation order. However....
- Order will sometimes determine whether we get a result at all! - see next.

Applicative vs. Normal Order evaluation

- Example - we have the definitions:

```
(define (forever a) (forever a))
(define (first x y) x)
```

- Now, evaluate (first 3 (forever 2)) in applicative order:

  (first 3 (forever 2)) => {definition of forever}

  (first 3 (forever 2)) => {definition of forever}

  (first 3 (forever 2)) => .... computation never stops

- Now evaluate same expression in normal order:

  (first 3 (forever 2)) => {definition of first}

  3

- Normal order terminated and Applicative order didn't
  - normal order evaluated *first first* rather than *forever first*

Church-Rosser The

- Two important theorems about evaluation orders.
- Church-Rosser theorem 1 - you can evaluate a functional expression in any order and, if you get a result using both orders, both results will be the same.
  - This is a nice theorem - order doesn't affect our result - one less thing to worry about.
- Church-Rosser theorem 2 - if evaluation of an expression can possibly terminate then normal order evaluation of that expression will terminate.
  - If termination is a concern then using normal order gives the best chance of completion.
  - However, applicative order is easy to understand and, often, more efficient so many languages use applicative order anyway.
- These theorems work only if there are no side-effects.

Example: Square Roots (A&S 1.1.7)

- Newtons method for finding square-roots
  - guess, divide into original number, average with last guess, continue...
  - until the square of the guess is close to the original number.

Guess	Quotient	Average
1	(2/1) = 2	((2 + 1)/2) = 1.5
1.5	(2/1.5) = 1.3333	((1.3333 + 1.5)/2) = 1.4167
1.4167	(2/1.4167) = 1.4118	((1.4167 + 1.4118)/2) = 1.4142
1.4142		

Square ro

- The iteration in the previous slide can be implemented as:

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
       guess
       (sqrt-iter (improve guess x) x)))
```

- Define improve:

```
(define (improve guess x)
  (average guess (/ x guess)))
```

- Define average:

```
(define (average x y) (/ (+ x y) 2))
```

- Define good-enough?:

```
(define (good-enough? guess x)
   (< (abs (- (square guess) x)) 0.001))
```

Square Roots

- Finally, define the sqrt function to call sqrt-iter with an initial guess of 1.0.

```
(define (sqrt x) (sqrt-iter 1.0 x))
```

- Note, as with other programming languages there is more than one way of writing a given function in Scheme.

Alternative definitio

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- Note the use of local definitions.
- Note the removal of x, the local parameter, from the local definitions.
- x is now free in the context of the local definitions
  - Though it is bound in the context of the outer definition.

Exercises:

- First, satisfy yourself that sqrt works. Then try:
- A&S (SICP) 1.4, 1.5, 1.7