Scheme Exercises

Alexandre Bergel
Software Composition Group, Institutfür Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland
{bergel, ducasse}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg

April 13, 2006

Exercise 1

Introduction and Basic

1.1 Objectives of this Chapter

Even if many Universities were touched by the ”Java syndrome”, Scheme is still widely spread over academic institution for language programming lectures.

The goal for you is assimilate some fundamental points such as:

- Understand the execution principle. Scheme uses a Read-Eval-Print loop (REP in short) which is quite far from the Editing-Compilation-Running process offered by the C/C++/Java world.
- Play with some simple data manipulation

1.2 Installation of Dr. Scheme

Dr. Scheme (`http://www.drscheme.org/`) is one of the richest Scheme interpreter. Of course it runs on all the platforms, but only distribution for windows and MacOSX are provided from the Scheme lecture webpage.

A solaris release has been preinstalled, to run it from the Sun-Rays, just type the following:

```
bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/local/teTeX/2.0.2/lib
~scg/Software/plt/bin/drscheme
```

After choosing your favorite (spoken) language, you have to choose the language you want to use with Dr Scheme (Dr. Scheme is not a simple R5RS Scheme[1] interpreter, but a platform using many dialects). Advanced Student should be enough, even if it provides much more things than you need.

1.3 Using Dr. Scheme

The frame is cut down in two panes:

- the upper one contains all your code (intended to be evaluated many times or saved into a file), whereas
- the lower one is called *transcript* and is useful for interactively evaluate expression or to display messages.

In the first lecture the expressions presented are small and simple. Just type them in the transcript one by one separated by pressing the return key.

[1] R5RS stands for Revised^5 Report on the Algorithmic Language Scheme which is the specification of Scheme

Your job: Evaluate the following: `(* 5 6)`, `(+ 2 4 6 8)`, `(/ 1 3)`, `(* 2 (/ 1 2))`

1.4 Manipulating Lists

List construction and manipulation are fully integrated into the language itself. Only three functions are needed:

- `(cons v1 v2)` to build a pair formed by two values `v1` and `v2`
- `(car p)` to get the first element of a pair,
- `(cdr p)` to get the second element of a pair.

When pairs are nested, the formed structure is called a *list*. List can be build in many different ways: by calling `cons` or by calling `list`.

Your job: Try in the transcript the following: `(cons 'a 'b)`, `(cons 'a (cons 'b (cons 'c '())))`, `(list 'a 'b 'c)`

Even the function `list` can be expressed in term of these three operations! Scheme provides useful functions or build-in value such as:

- `()` is the empty list, called also nil element,
- `(null? el)` returns true if `el` is the nil element,
- `(length L)` to compute the length of a list,
- `(map f L)` return a new list containing the result of applying `f` to all the elements of `L`,
- `apply` ...

Your job: The meaning of `length` can be illustrated by evaluating: `(length '(a b (c d) e f))`

1.4.1 First Simple Functions

In this subsection most function you will have to write has to be in a non-terminal recursive style. Just write answers in their simplest form. The terminal one is tackled with local function and continuation.

Let's start to write some more interesting things related to list manipulations by defining your own `length` function named `mylength`.

Your job: Write `mylength` using a non-terminal recursion.

Your job: Write the `iota` function taking two integer as parameter and returning a list containing the set of integers contained between the two provided (e.g., `(iota 5 12) => (5 6 7 8 9 10 11 12)`

Your job: Write a function `(member el L)` returning `#t` if `el` is contained into `L`. E.g., `(member 'hello '(hi john! hello Bob)) => #t` and `(member 'hello '((hi john!) (hello Bob))) => #f`

Your job: Write `(member* el L)` for which `(member 'hello '((hi john!) (hello Bob))) => #t`

1.4.2 Reverting a List

Dr. Scheme provides a primitive `reverse`.

Your job: Try `(reverse (iota -10 10))`

Your job: Try `(reverse '(1 2 (3 4))`. What can you conclude about `reverse`?

Your job: Write your own reverse function named `myreverse`. You need to use the primitive `append` and `list`.

Optional job: Write `myreverse*` useful to reverse elements (which could be lists) contained into a list. E.g., `(myreverse* '((1 2) (3 4)))` => `((4 3) (2 1))`

1.4.3 Comparing List

The goal of this exercise is to write a function `same-fringe?`:

- `(same-fringe? '(1 (2 3)) '((1 2) 3))` => `#t`
- `(same-fringe? '(1 2 3) '(1 (3 2))` => `#f`

Your job: First write a function `flatten` such as `(flatten '(1 (2 (3 4) (5 6 7))))` => `(1 2 3 4 5 6 7)`

Your job: Then write the function `same-fringe?`

1.4.4 Puzzle

Your job: Given that `+` and `*` can be applied to lists of arbitrary length, what is the result of `(+)` and `(*)`?

Your job: However, `-` also applies to lists, but `(-)` is an error. Explain.

1.5 High Order Function

Within Scheme, functions are first class object, this mean they can be manipulated like other value such as numbers or strings.

1.5.1 Manipulating Functions

Your job: *Function as Arguments*: Write a function `filter` taking a function f and a list L as argument and returning elements of L for which $f(el) = \#t$.

Your job: *Function as Result*: Write a function `mult-by` describing the behavior `((mult-by 2) ((mult-by 3) 4))` => `24`

Your job: *Composed Functions*: Write a function `odd-list` according to: `(odd-list '((1 2 3 4) (5 6 7 8) (10 11 12 13)))` => `((1 3) (5 7) (11 13))`

1.5.2 Counter

Here is the definition of a simple counter incremented by one each time it is evaluated:

```
(define (create-counter)
  (let ((index -1))
      (lambda ()
          (set! index (+ 1 index))
          (index))))
```

And it can be used by evaluating the following within the Transcript:

```
> (define mycounter (create-counter))
> (mycounter)
0
>  (mycounter)
1
>  (mycounter)
2
```

Your job: In the previous definition, there is a bug, identify it and explain why.

Your job: Modify `create-counter` for allowing to reset a counter like `(mycounter 'reset)`

Your job: A counter increments by one, modify it for passing as argument the value used for incrementing. E.g.,

- `(let ((c (create-counter))) (+ (c) (c) (c) (c)))` => 6
- `(let ((c (create-counter 2))) (+ (c) (c) (c) (c)))` => 12

Exercise 2

Macros, Lazy Evaluation and Streams

2.1 Experiments with Macroes: The Loop `do`

You have to create the macro `do` useful for performing loop. It has the following pattern:

```
(do ((var1 init1 incr1)
        ...
     (varN initN incrN))
       (test-end [resultat])
     body)
```

The effect of evaluating this special form is to initialise local variables `var1, ..., varN` with expressions `init1, ..., initN`. Then, if the expression `test-end` has `#f` for value, `body` is executed. The loop restarts after having incremented the variable `varj` with `incrj`.

Example

```
(define (table x)
  (do ((i 1 (+ i 1)))
      (display-alln x " x " i " = " ( * i x))))
 (table 7)

 7 x 1 = 7
 7 x 2 = 14
 7 x 3 = 21
 ...
```

Your job: Express a `do` construct as a `letrec`

Your job: Propose an implementation of `do`

2.2 `fluid-let`

The special form `fluid-let` is a kind of `let` without any local variable creation but with a temporary modification of visible variables. The syntax of `fluid-let` is the same than the one of `let`:

```
(fluid-let ((x1 e1)
             ...
            (xN eN))
     s1 s2 ...)
```

The semantic consists of saving the value of variables `xi`, to affect them value of `ei`, then to compute the body within this environment, and then finally to restore the value of `xi`. The result is the value of the body.

Example 1

```
(let* ((a 10)
       (f (lambda (u) (+ u a))))
     (fluid-let ((a 0))
        (list a (f 0))))

(0 0)
```

Example 2

```
(define counter 0)
(define (bump-counter) (set! counter (+ 1 counter)) counter)
counter =>0

(fluid-let ((counter 99  ))
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline))
100
101
counter => 0

(let ((counter 99))
  (display (bump-counter )) (newline)
  (display (bump-counter )) (newline))
1
2
counter => 2
```

Your job: Express a `fluid-let` construct as a `letrec`

Your job: Propose an implementation of `do`

2.3 Playing with Streams

A classical application of streams is the Erastosthene Sieve aimed to enumerate the prime numbers. As 2 is the smallest prime number, we start with the following list (2 3 4 5 6 7 8 9 10 11 12 13 ...). The principle is to remove all the multiple of 2, the remaining is (3 5 7 9 11 13 ...). So 3 is a prime number because it is not divisible by a number lower than itself. Then multiple of 3 are removed, the remaining is (5 7 11 13 ...), so 5 is a prime number. Same thing with multiple of 5, the resulting list is (7 11 13 ...) so 7 is a prime number... At each step the smallest integer is a prime number because there is no smallest number than it.

Your job: Write the list of prime numbers.

Example

```
> (stream-ref prime-numbers 1)
3
> (stream-ref prime-numbers 0)
2
> (stream-ref prime-numbers 1)
3
```

```
> (stream-ref prime-numbers 2)
5
> (stream-ref prime-numbers 3)
7
> (stream-ref prime-numbers 200)
1229
```