Chapter 11

Scheme – Imperative Features

Imperative Features of Scheme

- Sequence of statements
- Assignment (side effects)
- Iteration

Sequences

Indicate a sequence with a begin block

Example:

```
(begin
        (display "Welcome to my program")
        (newline)
        (display "Enter a number: " )
        (newline)
        (define x (read))
)
```

Sequences

Scheme allows multiple statements without a begin block, but other functional languages don't.

Use (begin .....)

Side Effects

! convention – procedures having side effects have names ending with !

set! – takes an element and an expressions and makes the element evaluate to the result of the expression

set-car! – takes a list and an expression and sets the car of the list to the result of the expression

set-cdr! – takes a list and an expression and sets the cdr of the list to the result of the expression

Side Effects

For **set!** the element needs to already have been defined as a constant

Example:

```
(define x 10)
(write x)
(newline)
(set! x (+ x 1))
(write x)  => 10     11
```

Better to surround all with (begin)

Side Effects – instead use 'let'

For **set!** the element needs to already have been defined as a constant

Example:

```
(begin
        (define x 10)
        (write x)
        (newline)
        (set! x (+ x 1))
        (write x)
)  => 10     11
```

Built-In Procedure - let

let - takes two arguments, a list of pairs and an expression. Each pair in the first list consists of an element and an expression. The 2nd argument of let is evaluated using the constant assignments given in the first argument. In other words, the variables are bound by the let for the duration of the expression.

Example:

```
(let (                  ; first argument
        (a 10)
        (b 15)
    )
    (+ a b)             ; second argument
)
```

Built-In Procedure - let

let - the initial values are computed and results are bound to variables

let* - from left to right, compute initial value and bind to variable, so earlier bindings can be used in later bindings

Example:

```
(let* ((a 2) (b 3 ) (c (+ a b)))
        (begin
                (display "c is ")
                (display c)
                (newline)
        )
)
```

I/O Scheme Functions

I/O Scheme functions:

read – read one Scheme object from the standard input and return it (thus, this is a function with a side effect , a new constant exists in the environment)

(define x (read))

write – print a representation of its argument to standard output so data could be read back in

(write (+ 3 5)) => 8

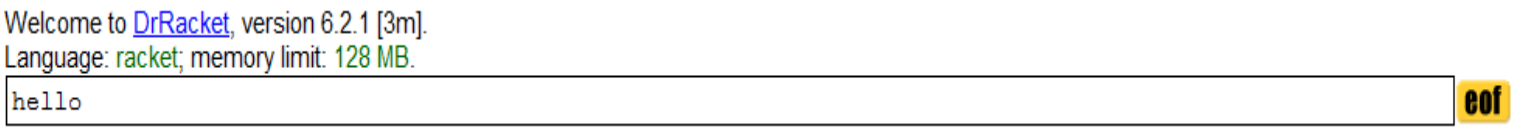
display – similar to write but datatypes are written as raw bytes or characters (not meant to be written back in)

Scheme Function read

read – read one Scheme object

Example:

```
(define x (read))
x
```



Returns ‘hello

Scheme Function read

read – read one Scheme object

Example:

```
(define x (read))
x
```

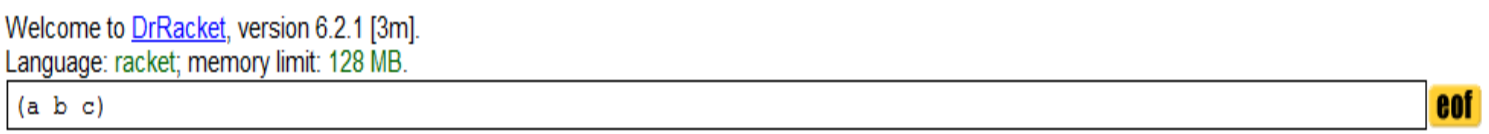
Returns ‘(a b c)

Scheme Function read-char

read-char – read a single character

Example:

```
(define x (read-char))
x
```



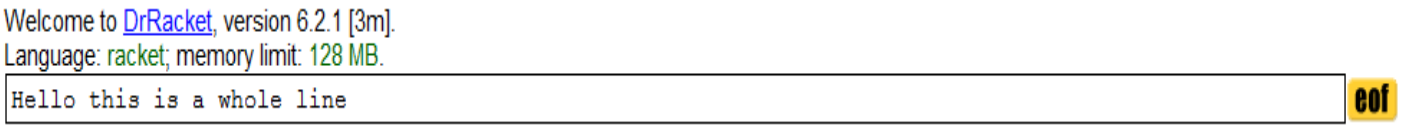
Returns #\(

Scheme Function read-line

read-line – read up to end of file

Example:

```
(define x (read-line))
x
```



Returns "Hello this is a whole line"

I/O Scheme File Functions

I/O Scheme functions:

open-input-file – as expected

Example: (open-input-file “table.csv”)

Example:

(read-line (open-input-file “table.csv”))

eof-object? – as expected

Example:

```
(if (eof-object? (read-line
                  (open-input-file “table.csv”)))
```

Ports

Have the usual ports – input, output and error

open-input-file – takes a string pathName, opens the file for input and returns an input-port, or triggers *exn:fail:filesystem* exception

open-output-file – similarly, but will create a file, so it can't already exist

close-input-port
close-output-port
custodian-shutdown-all

CSV Readers

Comma-Separated Value (CSV) Parsing

Useful package: csv-reading by Neil Van Dyke

Once installed add:
(require csv-reading)

https://docs.racket-lang.org/csv-reading/index.html

CSV-Reading documentation

1. Introduction
2. Inputs – we can use the defaults
3. Making reader makers – works well
4. Making readers
5. High level conveniences
6. Converting csv to sxml
7. History

https://docs.racket-lang.org/csv-reading/index.html

Example

```
#lang racket

(require csv-reading)   ; Package that contains utilities for reading csv files.

; Build a reader that can read a csv file containing the scan table.
; Use the default parameters.
(define scanTable-csv-reader
  (make-csv-reader-maker '())
)

; Function that uses the csv reader to open a csv file once, and allow
; repeatedly readding the next row, returning the row contents as a list.
(define next-row
    (scanTable-csv-reader (open-input-file "lexicalTable.csv"))
)


; Function that reads and displays each row of the csv file as a list.
(define display-contents
  (lambda ()
    (let
        ((line (next-row)))
        (unless (null? line)
          (begin
            (display "Next line: ")
            (display line)
            (newline)
            (display-contents)
            )))))



(display-contents)
```

Scheme Reference Sheet

http://www.nada.kth.se/kurser/su/DA2001/sudata14/examination/schemeCheatsheet.pdf