

SCHEME

after signing on unix, type the command "scheme"

Scheme saved ...

Release 7.3.1

Microcode 11.146

Runtime 14.166

1]=> 7

;Value: 7

1]=> (+ 3 4)

;Value: 7

1]=> (* 2 2 (+ 1 2))

;Value: 12

1]=> (+ A B)

;Unbound variable: b

;To continue, call RESTART with an option number:

; (RESTART 3) => Specify a value to use instead of b.

; (RESTART 2) => Define b to a given value.

; (RESTART 1) => Return to read-eval-print level 1.

2 error> (RESTART 1)

;Abort!

1]=> (+ b a)

;Unbound variable: a

;To continue, call RESTART with an option number:

; (RESTART 3) => Specify a value to use instead of a.

; (RESTART 2) => Define a to a given value.

; (RESTART 1) => Return to read-eval-print level 1.

2 error> (RESTART 1)

;Abort!

1]=> (+)

;Value: 0

1]=> (*)

;Value: 1

1]=> (A)

;Unbound variable: a

;To continue, call RESTART with an option number:

; (RESTART 3) => Specify a value to use instead of a.

; (RESTART 2) => Define a to a given value.

; (RESTART 1) => Return to read-eval-print level 1.

2 error> (RESTART 1)

;Abort!

1]=>

The symbol A can be bound to a **value**.

Values are the usual things: numbers, strings, etc.,
but other objects can be values including *functions*

(define a 5)

;Value: a

1]=> (+ a 3)

;Value: 8

1]=> (define A (lambda (X) (+ X 1)))

;Value: a

1]=> (a (+ 3 2))

;Value: 6

1]=> (+ a 3)

;The object #[compound-procedure 1 a], passed as the first argument

;To continue, call RESTART with an option number:

;(RESTART 2) => Specify an argument to use in its place.

;(RESTART 1) => Return to read-eval-print level 1.

2 error> (RESTART 1)

;Abort!

1]=>

unix2% which scheme

/usr/local/depts/cs/scheme/7.3/usr/bin/scheme

unix2% scheme

Scheme Microcode Version 11.146

MIT Scheme running under SunOS

Type '^C' (control-C) followed by 'H' to obtain information about

Scheme saved on Sunday November 21, 1993 at 9:15:23 PM

Release 7.3.1

Microcode 11.146

Runtime 14.166

1]=> (define x 1)

;Value: x

1]=> (+ x 1)

;Value: 2

1]=> (define id (lambda (x) x))

;Value: id

1]=> (id (+ x x))

;Value: 2

1]=> ((lambda (x) (+ x 10)) 100)

;Value: 110

1]=> (1 2 3)

;The object 1 is not applicable.

**;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.**

2 error> (RESTART 1)

;Abort!

1]=> (define xls '(1 2 3))

;Value: xls

1]=> (car xls)

;Value: 1

1]=> (cdr xls)

;Value 3: (2 3)

1]=> (cons 0 xls)

;Value 4: (0 1 2 3)

1]=> (eq? xls xls)

;Value: #t

1]=> (eq? xls '(1 2 3))

;Value: ()

1]=> (equal? xls '(1 2 3))

;Value: #t

1]=> (eq? xls (cons (car xls) (cdr xls)))

;Value: ()

(now back to unix)

acunix2% cat fns_1

(define (cube x) (* x x x))

(define (lastelem ls)

(if (null? ls)

()

(if (null? (cdr ls))

(car ls)

(lastelem (cdr ls)))))

acunix2% scheme

Scheme Microcode Version 11.146

MIT Scheme running under SunOS

Type '^C' (control-C) followed by 'H' to obtain information about

^L

Scheme saved on Sunday November 21, 1993 at 9:15:23 PM

Release 7.3.1

Microcode 11.146

Runtime 14.166

1]=> (/ 355.0 113.0)

;Value: 3.1415929203539825

1]=> (define x 100)

;Value: x

1]=> (define y (* x x))

;Value: y

1]=> (load "fns_1")

;Loading "fns_1" -- done

;Value: lastelem

1]=> (cube x)

;Value: 1000000

1]=> (lastelem '(1 2 3 4 10))

;Value 1: 10

1]=> (lastelem (1 2 3 4 10))

;The object 1 is not applicable.

;To continue, call RESTART with an option number:

; (RESTART 2) => Specify a procedure to use in its place.

; (RESTART 1) => Return to read-eval-print level 1.

2 error> ^G

;Quit!

1]=>

Pseudo Random Numbers

SCHEME has a pseudo-random number generator function called "random" that is not covered in Dybvig's book.

If n is bound to a positive integer, then (random n) returns an integer between 0 and $n-1$.

```
1 ]=> (random 12)
```

```
;Value: 3
```

```
1 ]=> (random 12)
```

```
;Value: 1
```

```
1 ]=> (random 12)
```

```
;Value: 6
```

```
1 ]=> (random 12)
```

```
;Value: 9
```

```
1 ]=> (random 12)
```

```
;Value: 4
```

```
1 ]=>
```

Minor problem: Every time scheme is invoked, the first five calls to (random 12) produce exactly 3, 1, 6, 9, and 4.

This is good for debugging, but not for the final program.

The function "random" is controlled by a variable called `*random-state*`. We can cause `*random-state*` to be given a somewhat arbitrary initial value using the "make-random-state" function:

```
(set! *random-state* (make-random-state #t))
```

By evaluating this expression initially in our split function, we get different ways of splitting the rock pile.

```
(define split (lambda (n)
  (cond ((< n 2) 0)
        (#t (set! *random-state* (make-random-state #t))
              (split2 n) ) ) ))
```

```
(define split2 (lambda (n)
  (cond ((< n 2) 0)
        (#t (let ((k (+ 1 (random (- n 1)))))
              (write-string "Splitting ") (write n)
              (write-string " rocks into ") (write k)
              (write-string " and ") (write (- n k))
              (newline)
              (+ (* k (- n k)) (split2 (- n k))
                 (split2 k) ) ) ) ) ))
```