KRR – RCS

Introduction to Scheme (1)

Sl. dr. ing. Andrei Horia Mogos

Contents

- Data types
- Conditionals
- Functions

Contents

- **Data types**
- Conditionals
- Functions in Scheme

Data types

- Simple data types:
  - Booleans
  - Numbers
  - Characters
  - Symbols
- Compound data types
  - Strings
  - Vectors
  - Dotted pairs
  - Lists

Data types

- **Simple data types:**
  - Booleans
  - Numbers
  - Characters
  - Symbols
- Compound data types
  - Strings
  - Vectors
  - Dotted pairs
  - Lists

Data types

- **Booleans:**

```
(boolean? #t)        =>  #t
(boolean? "Hello, World!") =>  #f

(not #f)        =>  #t
(not #t)        =>  #f
(not "Hello, World!") =>  #f
```

Data types

- **Numbers:**

```
(number? 42)       =>  #t
(number? #t)       =>  #f
(complex? 2+3i)    =>  #t
(real? 2+3i)       =>  #f
(real? 3.1416)     =>  #t
(real? 22/7)       =>  #t
(real? 42)         =>  #t
```

```
(rational? 2+3i)   =>  #f
(rational? 3.1416) =>  #t
(rational? 22/7)   =>  #t
(integer? 22/7)    =>  #f
(integer? 42)      =>  #t
```

Data types

- **Numbers (cont.):**

```
(eqv? 42 42)   =>  #t
(eqv? 42 #f)   =>  #f
(eqv? 42 42.0) =>  #f

(= 42 42)   =>  #t
(= 42 #f)   -->ERROR!!!
(= 42 42.0) =>  #t

(< 3 2)    =>  #f        ; other relational operators: >, <=
(>= 4.5 3) =>  #t
```

Data types

- **Numbers (cont.):**

```
(+ 1 2 3)    =>  6
(- 5.3 2)    =>  3.3
(- 5 2 1)    =>  2
(* 1 2 3)    =>  6
(/ 6 3)      =>  2
(/ 22 7)     =>  22/7
(expt 2 3)   =>  8
(expt 4 1/2) =>  2.0
```

```
(- 4) =>  -4
(/ 4) =>  1/4
(max 1 3 4 2 3) =>  4
(min 1 3 4 2 3) =>  1
(abs  3) =>  3
(abs -4) =>  4
; other functions: floor,
; exp, sqrt
```

Data types

- **Characters:**

```
(char? #\c) =>  #t
(char? 1)   =>  #f
(char? #\;) =>  #t

(char=? #\a #\a)  =>  #t
(char<? #\a #\b)  =>  #t
(char>=? #\a #\b) =>  #f
```

```
(char-ci=? #\a #\A) =>  #t
(char-ci<? #\a #\B) =>  #t

(char-downcase #\A) =>  #\a
(char-upcase #\a)   =>  #\A
```

Data types

- **Symbols:**

```
#t  =>  #t
42  =>  42      ; self-evaluating
#\c =>  #\c
```

Symbols = identifiers for variables

```
(define xyz 9)

(quote xyz) =>  xyz
'xyz => xyz
xyz => 9
```

```
(set! xyz #\c)
xyz =>  #\c

(symbol? 'xyz) =>  #t
(symbol? 42)   =>  #f
```

Data types

- Simple data types:
  - Booleans
  - Numbers
  - Characters
  - Symbols
- **Compound data types**
  - Strings
  - Vectors
  - Dotted pairs
  - Lists

Data types

- **Strings:**

```
"Hello, World!" =>  "Hello, World!"
```

```
(define hello (string #\H #\e #\l #\l #\o))
hello =>  "Hello"
```

```
(define greeting "Hello; Hello!")
(string-ref greeting 0) =>  #\H
```

```
(define a-3-char-long-string (make-string 3))
```

string? -> predicate

Data types

- **Strings: (cont.)**

```
(string-append "E " "Pluribus " "Unum")
=> "E Pluribus Unum"


(define hello (string #\H #\e #\l #\l #\o))
(string-set! hello 1 #\a)
hello => "Hallo"
```

string-set! -> only for strings created using string, make-string, and string-append

Data types

- **Vectors:**

```
(vector 0 1 2 3 4) =>  #(0 1 2 3 4)
```

```
(define v (make-vector 5))
```

vector-ref -> the same as **string-ref** for strings
vector-set! -> the same as **string-set!** for strings
vector? -> predicate

Data types

- **Dotted pairs:**

```
(cons 1 #t) =>  (1 . #t)
'(1 . #t) =>  (1 . #t)
(1 . #t)  -->ERROR!!!

(define x (cons 1 #t))
(car x) =>  1
(cdr x) =>  #t

(set-car! x 2)
(set-cdr! x #f)
x => (2 . #f)
```

```
(define y (cons (cons 1 2) 3))
y =>  ((1 . 2) . 3)

(car (car y)) => 1
(cdr (car y)) => 2

(caar y) => 1
(cdar y) => 2

c…r -> maximum 4 ‘a’ or ‘d’
```

Data types

- **Lists:**

- special types of dotted pairs

(cons 1 (cons 2 (cons 3 (cons 4 5)))) => (1 2 3 4 . 5)
(1 2 3 4 . 5) -> abbreviation for (1 . (2 . (3 . (4 . 5))))

'() => () ; **empty list**
(cons 1 (cons 2 (cons 3 (cons 4 **'()**)))) => (1 2 3 4) -> **list**
(1 2 3 4) -> abbreviation for (1 . (2 . (3 . (4 . ()))))

Data types

- **Lists (cont.):**

```
(list 1 2 3 4) => (1 2 3 4)
'(1 2 3 4) => (1 2 3 4)

(define y (list 1 2 3 4))

(list-ref y 0) =>  1
(list-ref y 3) =>  4
(list-tail y 1) =>  (2 3 4)
(list-tail y 3) =>  (4)
```

```
(pair? '(1 . 2)) =>  #t
(pair? '(1 2))   =>  #t
(pair? '())      =>  #f
(list? '())      =>  #t
(null? '())      =>  #t
(list? '(1 2))   =>  #t
(list? '(1 . 2)) =>  #f
(null? '(1 2))   =>  #f
(null? '(1 . 2)) =>  #f
```

Data types

- **Conversions between data types:**

(char->integer #\d) => 100
- other: integer->char

(string->list "hello") => (#\h #\e #\l #\l #\o)
- other: list->string, vector->list, and list->vector

(number->string 16) => "16"
- other: string->number

(symbol->string 'symbol) => "symbol"
- other: string->symbol

Contents

- Data types
- **Conditionals**
- Functions

Conditionals

- **if:**

```
(if test-expression
   then-branch
   else-branch)

(define p 80)
(if (> p 70)
   'safe
   'unsafe)
=> safe
```

Conditionals

- **cond:**

```
(if (char<? c #\c)
    -1
    (if (char=? c #\c)
            0
            1))

(cond
    ((char<? c #\c) -1)
     ((char=? c #\c) 0)
     (else 1))
```

Conditionals

- **and, or:**

```
(and 1 2)  =>  2
(and #f 1) =>  #f
```

```
(or 1 2)  =>  1
(or #f 1) =>  1
```

- Any value that is not #f is considered to be **true**

Contents

- Data types
- Conditionals
- **Functions**

Functions

- **Lambda expressions:**

```
(lambda (x) (+ x 2))
((lambda (x) (+ x 2)) 5) =>  7

(define add2
  (lambda (x) (+ x 2)))

(add2 4) =>  6
(add2 9) =>  11
```

Functions

- **Recursive functions using the stack:**

```
(define factS
 (lambda (n)
   (if (= n 0)
       1
       (* n (factS (- n 1))))))

(factS 5) => 120
```

Functions

- **Recursive functions using the stack (cont.):**

```
(define lenS
 (lambda (l)
  (if (null? l)
    0
    (+ 1 (lenS (cdr l))))))

(lenS '(1 2 3 4 5)) => 5
```

Functions

- **Recursive functions using tail recursion:**

```
(define factT0
  (lambda (n acc)
    (if (= n 0)
        acc
        (factT0 (- n 1) (* n acc)))))

(define factT (lambda (n) (factT0 n 1)))

(factT 5) => 120
```

Functions

- **Recursive functions using tail recursion (cont.):**

```
(define lenT0
  (lambda (l acc)
    (if (null? l)
        acc
        (lenT0 (cdr l) (+ acc 1)))))

(define lenT (lambda (l) (lenT0 l 0)))


(lenT '(1 2 3 4 5)) => 5
```

Credit

- The examples from slides 6 – 25 are taken from:

D. Sitaram, “Teach Yourself Scheme in Fixnum Days”, 1998 - 2004