1 Racket

1.1 Comments

```
; single line comment
#|
  multi-line-comment
        can span
        multiple lines
  end of comment
|#
```

1.2 Datum evaluation

- (quote <datum>) or '<datum> leaved the datum as-is
- (unquote <datum>) or ,<datum> is the opposite of quote
- (quasiquote <datum>) or ,@<datum> allows to apply the unquote where needed

```
'(1 2 3); => (1 2 3)
(1 ,(+ 1 1) 3) ; => '(1 2 3)
```

1.3 Predicates

- all predicates end with ?
- checks if a number is even: even?
- checks if a number is odd: odd?
- check if a datum is true: true?

```
(even? 2) ; => #t
(odd? 2) ; => #f
(true? #t) ; => #t
```

1.3.1 Equivalence

- check if two numbers are equal: =
- checks if two objects or numbers are the same: eq?
- checks if two objects are the same: eqv?
- checks if two objects are the same: equal?

```
(= 1 1) ; => #t
(eq? 1 1) ; => #t
(eqv? 1 1) ; => #t
(equal? 1 1) ; => #t
```

1.4 Data types

- integer: 9125
- binary: #b10001110100101
- octal: #o21645
- hexadecimal: #x23a5
- real: 91.25
- rational: 91/25
- complex: 91+25i
- boolean: #t, #f
- character: #A, #λ, #\\u30BB
- string: "Hello, world!"
- null element: '(), null
- lists: '(1 2 3)

```
(define x 5) ; => x = 5
(define y "Hello, world!") ; => y = "Hello, world!"
(define z #t) ; => z = #t
null ; => '()
```

1.4.1 Basic operations

All the operators are in the form (<operator> <operand> ...) *(prefix notation)*.

Operations on numbers

- arithmetic operations: +, -, *, /
- exponentiation: exp
- quotient: quotient
- remainder: remainder
- add 1: add1
- subtract 1: sub1

```
(+ 1 2 3) ; => 6
(- 1 2 3) ; => -4
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(add1 5) ; => 6
(sub1 5) ; => 4
```

Operations on strings

- string length: string-length
- string append: string-append
- string to list: string->list
- list to string: list->string
- get n-th character: string-ref

```
(string-length "Hello, world!") ; => 13
(string-append "Hello, " "world!") ; => "Hello, world!"
(string->list "Hello") ; => '(#\H #\e #\l #\l #\o)
(list->string '(#\H #\e #\l #\l #\o)) ; => "Hello"
(string-ref "Hello" 0) ; => #\H
```

Operations on bools

- logic operations: and, or, not, xor, nor, nand
- implication: implies

```
(and #t #f) ; => #f
(or #t #f) ; => #t
(not #t) ; => #f
(xor #t #f) ; => #t
(nor #t #f) ; => #f
(nand #t #f) ; => #t
(implies #t #f) ; => #f
```

1.5 Functions

- anonymous functions: (lambda (<arg1> <arg2> ...) <body>)
- named functions: (define (<name> <arg1> <arg2> ...) <body>)
- old way: (define <name> (lambda (<arg1> <arg2> ...) <body>))

```
(lambda (x) (+ x 1)) ;
(define (add1 x) (+ x 1)) ;
```

1.5.1 Higher order functions

- apply a function to each element of a list: map
- apply a filter: filter
- apply a function to each element of a list and flatten the result: apply
- fold a list: foldl, foldr

```
(map add1 '(1 2 3)) ; => '(2 3 4)
(filter even? '(1 2 3 4)) ; => '(2 4)
(apply append '((1 2) (3 4))) ; => '(1 2 3 4)
(foldl + 0 '(1 2 3)) ; => 6
(foldr + 0 '(1 2 3)) ; => 6
```

1.6 Variables

- parallel binding: let
- serial binding: let*
- recursive binding: letrec

```
(let ((x 5) (y 2)) (list x y)) ; => '(5 2)
(let* ((x 1) (y (add1 x))) (list x y)) ; // '(1 2)
```

1.7 Collections

1.7.1 Structs

- definition: (struct <struct-name> (<field> ...))
- constructor: (define <name> <struct-name> <field-value> ...)
- getter: <struct-name>-<field-name>
- setter: set-<struct-name>-<field-name>!
- predicate: <struct-name>?
- structs and fields are immutable by default
- use #:mutable keyword on struct or field to make it mutable

```
(struct point (x y)) ; => point
(define p (point 1 2)) ; => p = (point 1 2)
(point-x p) ; => 1
(point? p) ; => #t

(struct mut-point (x y #:mutable)) ; => point
(define mp (mut-point 1 2)) ; => mp = (mut-point 1 2)
(set-mut-point-x! mp 5) ; => mp = (mut-point 5 2)
```

1.7.2 Pairs

- definition: (cons <first> <second>)
- getter of first element: car
- getter of second element: cdr
- car and cdr can be composed *(cdadadr, caaar)*
- pairs are immutable

```
(cons 1 2) ; => '(1 . 2)
(car '(1 . 2)) ; => 1
(cdr '(1 . 2)) ; => 2
(caar '((1 . 2) . 3)) ; => 1
(cadr '((1 . 2) . 3)) ; => 2
(cdar '((1 . 2) . 3)) ; => 2
(cddr '((1 . 2) . 3)) ; => 3
```

1.7.3 Lists

- lists are composed of pairs
- manually defined via quote: ’(1 2 3)
- empty list: ’()
- lists are made by pairs
    - the car contains the first value
    - the cdr contains the the rest of the list
    - the last pair has cdr equal to ’()

```
'(1 2 3) ; => '(1 2 3)
'(1 . (2 . (3 . ()))) ; => '(1 2 3)
```

Operations on lists

- list length: length
- add an element at the beginning: cons
- add an element at the end: append
- take the first element: first
- take the last element: last
- take the n-th element: list-ref <list> <n>

- take the n-th element after pos: `list-tail <list> <pos>`
- count the occurrences of an element: `count <predicate> <list>`
- apply a filter: `filter <predicate>`
- apply a function to each element: `map <function>`
- get the reverse of a list: `reverse`
- get the first element: `first`
- get the elements after the first: `rest`

```
(length '(1 2 3)) ; => 3
(cons 1 '(2 3)) ; => '(1 2 3)
(append '(1 2) '(3 4)) ; => '(1 2 3 4)
(first '(1 2 3)) ; => 1
(last '(1 2 3)) ; => 3
(list-ref '(1 2 3) 1) ; => 2
(list-tail '(1 2 3) 1) ; => '(2 3)
(count even? '(1 2 3 4)) ; => 2
(filter even? '(1 2 3 4)) ; => '(2 4)
(map add1 '(1 2 3)) ; => '(2 3 4)
(reverse '(1 2 3)) ; => '(3 2 1)
(first '(1 2 3)) ; => 1
(rest '(1 2 3)) ; => '(2 3)
```

Lists folding

- lists can be folded from the left with `foldl`
- lists can be folded from the right with `foldr`
- the accumulator is the first argument of the function
- the list is the second argument of the function
- the function is applied to the accumulator and the first element of the list

```
(foldl + 0 '(1 2 3 4)) ; => 10
(foldr * 1 '(1 2 3 4)) ; => 24
```

1.7.4 Vectors

- definition: `#(<element> ...)`
- getter: `vector-ref`
- vector are immutable, fixed size and zero-indexed

```
#(1 2 3) ; => '#(1 2 3)
(vector-ref '#(1 2 3) 0) ; => 1
```

1.7.5 Sets

...

1.7.6 Hash

...

1.8 Control flow

1.8.1 Conditionals

if ...

cond ...

pattern matching ...

2 Erlang

3 Haskell