Data Types

Boolean: #t #f

> : (boolean? 1) ⟶ #f
> (boolean? #f) ⟶ #t
>
> Predicate statement is a statement that returns either true or false.
>
> (and #f #f) ⟶ #f
>
> (not #t)⟶ #f
>
> (eq? #t #t) ⟶ #t
>
> (eq? #f #t) ⟶ #f

Number

> 1, -2 1.5, -173 are the literals for the numer data type.
>
> (number? -2.5) ⟶ #t
>
> (number? #t) ⟶ #f
>
> +, *, - and / are predefined proceduric values.
>
> (eq? 2.5 -2.3) ⟶ #f
>
> (= 1 1.0) ⟶ #t
>
> > <, >, >=, <=

Integer

> Any integer is a number but not all numbers are integers.
>
> (integer? #t)
>
> (integer? 1) ⟶ #t
>
> (integer? 2.3) ⟶ #f
>
> We also have the equality predicate

Characters

> #\A ⟶ Capital A, characters have a strange syntax.

> #\space ⟶ Special characters are given by this notation
>
> (char ?) ⟶ The type predicate for characters
>
> eq?
>
> char=?

Strings

> string?
>
> "hello world" ⟶ This is a string literal
>
> string-length
>
> (string-ref "hello" 1) ⟶You get #\e, the character in the specified index (1).
>
> (string #\h #\e #\l #\l #\o) ⟶ You can construct a sequence of values by using the constructor of the string.

Symbol

> Used very frequently in functional programming languages.
>
> What is a Symbol?
>
> - Say we have this reference:
>   - (+ x + 1), where x is a variable.
>   - (not y), complement the value of y, and here y is a variable reference
>   - (quote y), this is a special, expectional syntax. When we have quote here, the value we get here will be y^{symbOl}
>
> (defere x 1) ⟶
>
> (symbol ? 1) ⟶ #f
>
> (symbol ? x) ⟶ #f
>
> (symbol ? (quote x)) ⟶ #t, here quote x is symbol x
>
> (symbol ? 'x) ⟶#t
>
> (symbol 'y) ⟶ #t even if the y is not defined, we just check if y is a symbol.
>
> (symbol? '5) ⟶ f, Same thing as asking if 5 is a symbol

> We talked about type predicate and eq? predicate.
>
> - (eq? 'x 'y) ⟶ #f, these are two symbol values one is x xymbol and other s y symbol,

Lists

Very important for functional programming languages. First programming language we have is called LISP because of the name `LISt Processing`. The elements might have different data types, and we can have lists where each element has a different data type.

- Confusingly, lists use the smooth paranthesis notation.
    - (1 #t hello) is a list with 3 elements.

An element of a list can also be a list, we can have a list like: (1 (#t #f) ()). First element is number 1, second element is a list consisting of two boolean values and third element is an empty list.

We have the type predicate for the list data type as well:

- (list ? #t) ⟶ #f

How do we generate list values? There are several ways:

- Using the `quote` operator:
    - '(1 2 3), this is a generated list value with 3 elements
    - (list? '(1 2 3)) ⟶ #t
    - (list? '(a b (cd))) ⟶ #t, The list is consisting of 2 symbol values and a list with 2 symbol values with 3 elements in total.

(quote (a b 1 2)) will generate this list: (a b 1 2).

- Another way of constructing a list is by using the constructor:
    - (list ex1 ex2 ... exn). Say ex1 is evaluated to v1, ex2 to v2 and exn to vn. The list expression evaluation will evaluate to (v1 v2 ... vn).
    - (list 1 (+2>) 'x) ⟶ (1 5 x), List is another procedure that we can use to construct the list data type.
    - (list) ⟶ Having 0 elements is also possible
- Anohter way of generating lists is by using `cons` procedure. It takes two parameters only. So when we evaluate cons:
    - (cons exp1 exp2), second expression should evaluate to a list value such as (v1 v2 ... vn), and exp1 should evaluate to a value. The evaluation of the cons will evaluate to ⟶ (v0 v1 v2 ... vn). There's an exception for this.
    - Say we have (cons 'a '(bc)) ⟶ A list with 3 elements (a b c)
    - (list 'a '(bc)) ⟶ We have two operands, so we have two elements to construct. ⟶ (a (bc)).
    - (cons '(ab) '(cd)) ⟶ Value of the first expresson will be a list, and the rest of the list will come from the second operand ((a b) c d)
    - (cons '() '(c d)) ⟶ (() (c d))
    - (cons '(cd) '()) ⟶ ((cd))

What if exp2 is not a list for cons?

- (cons 'a 'b), when we evaluate this we will see something like a list but with a . ⟶ (a . b), and the evaluated value is a pair. Pair is related to a list, and list is implemented by making use of list.