# L3. Racket Basics 2

| | |
|---|---|
| 유형 | 강의 |
| 복습 여부 | ✓ |
| Status | Done |

## What kinds of PL elements exist for Computers?

- Numbers and Arithmetic
- Variables and Functions
- Conditional Expressions
- Conditional Functions
- Symbols
- Type Definitions
- Type Deconstruction
- Lists

## Type Definitions

### What is Type?

- Abstract definition of data (c.f. class in Java)
- Everything can be a type

  Ex. we can categorize human into quiet or talkative types (define-type *type-id*

[*variant_id*$_1$ (*field_id*$_{11}$ *contract_expr*$_{11}$)

...

(*field_id*$_{1n}$ *contract_expr*$_{1n}$)]

...

[*variant_id*$_m$ (*field_id*$_{m1}$ *contract_expr*$_{m1}$)

(*field_id*$_{ml}$ *contract_expr*$_{ml}$)])

E.g.,

```
(define-type human
      [mother (name string?)
              (age number?)
               (job string?)]
      [father (name string?)
                (age number?)
                (hobby string?)
                ...
```

- A constructor *variant_id*$_1$ is defined for each variant.
- Each constructor takes an argument for each field of its variant.
- The value of each field is checked by its associated contract_Expr$_{ij}$.
- Defines predicates *type_id?* and *variant_id*$_i$*?*, and accessors *variant_id*$_i$*-field_id*$_{jk}$.

Ex1. model GUI as a type

```
(define-type GUI
  [label  (text string?)]
  [button (text string?)
          (enabled? boolean?)]
  [choice (items (listof string?))
          (selected integer?)])
```

```
; create instances based on the definitions
(label "Pick a fruit")
(button "Ok" false)
(choice '("Apple" "Strawberry" "Banana") 0)
```

```
(label "Pick a fruit")
(button "Ok" #f)
(choice '("Apple" "Strawberry" "Banana") 0)
```

```
; assign this entire instance into the identifier named 'ch'
(define ch (choice '("Apple" "Strawberry" "Banana") 0))
; checks whether 'ch' is instance of choice or not
(choice? ch)
; [variant_id]-[field_id]
(choice-selected ch)
; checks whether 'ch' is instance of GUI or not
(GUI? ch)
```

```
#t
0
#t
```

## Ex2. Animal

```
(define-type Animal
  [bear (num_child number?)
        (color string?)]
  [giraffe (height number?)
           (name string?)
           (len_neck number?)]
  [snake (length number?)
         (poison boolean?)])
```

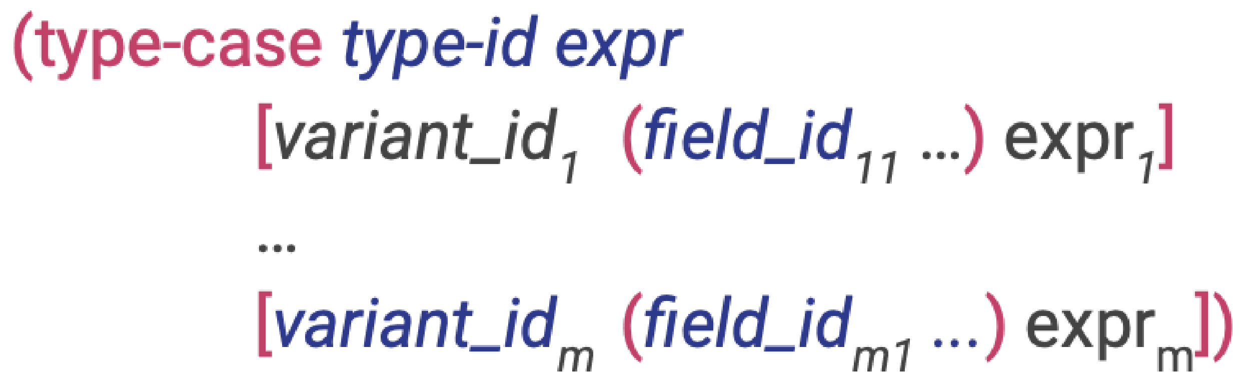
```
(define myBear(bear 2 "white"))
myBear
; prints the instance

(Animal? myBear)
; returns true because myBear is an instance of Animal
(bear? myBear)
; returns true because myBear is an instance of bear
(snake? myBear)
; returns false because myBear is not an instance of snake

; [variant_id]-[field_id]
(bear-num_child myBear)
```

```
(bear 2 "white")
#t
#t
#f
2
```

## Type Deconstruction

Deconstruct instance created by the type definition.

From a given instance of a specific type, get required values or do a specific task for the instance.



```
(type-case type-id expr
               [variant_id_1  (field_id_11 ...) expr_1]
               ...
               [variant_id_m  (field_id_m1 ...) expr_m])
```

Ex1. GUI

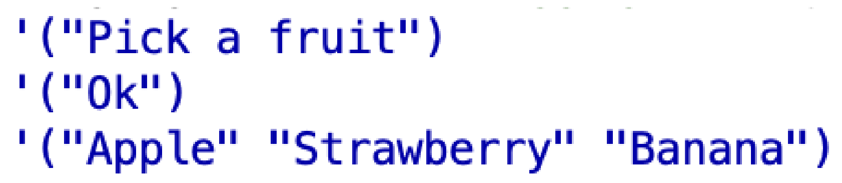
In the case of GUI, all the variants have at least one string type value.

We want to create a function that extracts string type values from any instance of the GUI type.

```
; type definitions
(define-type GUI
  [label  (text string?)]
  [button (text string?)
          (enabled? boolean?)]
  [choice (items (listof string?))
          (selected integer?)])

; create instances based on the definitions
(define myLabel(label "Pick a fruit"))
```

```
(define myButton(button "Ok" false))
(define myChoice(choice '("Apple" "Strawberry" "Banana") 0))
```

```
; we're only accepting instances of GUI type
(define (read-screen g)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]))

(read-screen myLabel)
(read-screen myButton)
(read-screen myChoice)
```



```
'("Pick a fruit")
'("Ok")
'("Apple" "Strawberry" "Banana")
```

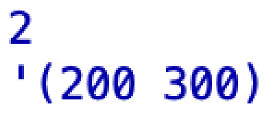
Ex2. Animal

In the case of Animal, all the variants have at least one number type value.

We want to create a function that extracts number type values from any instance of the Animal type.

```
; type definitions
(define-type Animal
  (bear (num_child number?)
        (color string?))
  (giraffe (height number?)
           (name string?)
           (len_neck number?))
  (snake (length number?)
         (poison boolean?)))

(define myBear(bear 2 "white"))
(define myGiraffe(giraffe 200 "JC" 300))
```

```
; getnumber: Animal -> list of numbers
(define (getnumber a)
  (type-case Animal a
    [bear (n c) n]
    [giraffe (h n l_n) (list h l_n)]
    [snake (l p?) l]))

(getnumber myBear)
(getnumber myGiraffe)
```



```
2
'(200 300)
```

## Pairs and Lists

list는 pair인데, pair라고 모두 lists는 아님.

### Pairs

A pair combines exactly two values.

The first value is accessed with the car procedure, and the second value is accessed with the cdr procedure.

Pairs are not mutable.

### List

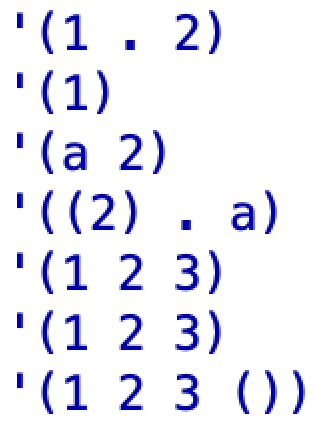
A list is either the constant *null*, or a pair whose second value is a list.

(list 1 2 3) or '(1 2 3)로 나타낼 수 있음

A list can be used as a single-valued sequence. The elements of the list serve as elements of the sequence.

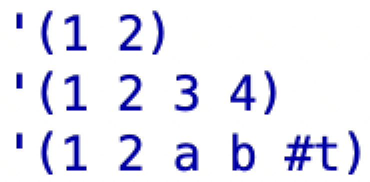
```
(cons 1 2) ; non-list pair
(cons 1 empty)
(cons 'a (cons 2 empty))
(cons (cons 2 empty) 'a) ; non-list pair
(list 1 2 3)
```

```
(cons 1 (cons 2 (cons 3 '())))
(list 1 2 3 empty)
```



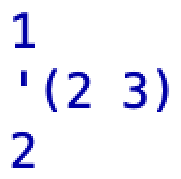
```
'(1 . 2)
'(1)
'(a 2)
'((2) . a)
'(1 2 3)
'(1 2 3)
'(1 2 3 ())
```

```
(append (list 1 2) empty)
(append (list 1 2) (list 3 4))
(append (list 1 2) (list 'a 'b) (list true))
```



```
'(1 2)
'(1 2 3 4)
'(1 2 a b #t)
```

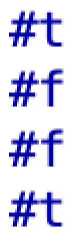
```
(first (list 1 2 3))
(rest  (list 1 2 3))
(first (rest (list 1 2)))
```



```
1
'(2 3)
2
```

```
; distinguish empty / non-empty lists
(empty? empty)
(empty? (cons "head" empty))
```

```
(cons? empty)
(cons? (cons "head" empty))
```

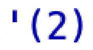


```
#t
#f
#f
#t
```

Racket documentation 4.10 Pairs and Lists

잘 이해했는지 확인!

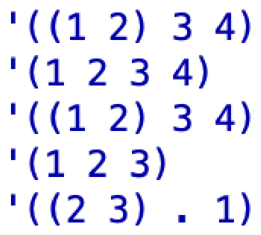
```
(cons (list-ref (rest (list 1 2 3)) 0) empty)
```



```
'(2)
```

**cons vs append**

```
(cons (list 1 2) (list 3 4))
(append (list 1 2) (list 3 4))
(list (list 1 2) 3 4)
(cons 1 (list 2 3))
(cons (list 2 3) 1)
```



```
'((1 2) 3 4)
'(1 2 3 4)
'((1 2) 3 4)
'(1 2 3)
'((2 3) . 1)
```

`cons` takes a single element and a list. It then prepends the element onto the list, making a new list.

It takes in any two types and pairs them up.

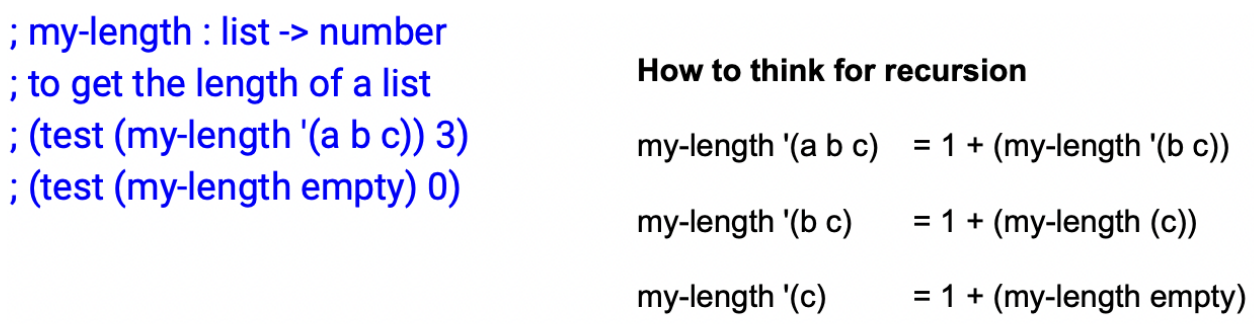
If the type of the second element is a list of the type of the first element, the whole thing is treated like a list.

If the two types don't have this match up, the whole thing is treated as a pair.

`append` combines two lists into a flat list.

## Coding Tip

### Recursion in Racket



```
; my-length : list -> number
; to get the length of a list
; (test (my-length '(a b c)) 3)
; (test (my-length empty) 0)
```

**How to think for recursion**

| | |
|---|---|
| my-length '(a b c) | = 1 + (my-length '(b c)) |
| my-length '(b c) | = 1 + (my-length (c)) |
| my-length '(c) | = 1 + (my-length empty) |

```
; list -> number
; to get the length of a list
; (test (my-length '(a b c)) 3)
; (test (my-length empty) 0)

(define (my-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (my-length (rest lst)))] ))

(test (my-length '(a b c)) 3)
(test (my-length empty) 0)
```

```
good (my-length '(a b c)) at line 12
  expected: 3
  given: 3

good (my-length empty) at line 13
  expected: 0
  given: 0
```