ITP30011
Racket Basics (2)

Lecture03
JC

Q&A

- remainder, modulo (% in other languages)
  - (test (modulo 3 2) 1)
  - (test (remainder 4 2) 0)
- A symbol is just an identifier
  - but a number preceded by ' will just be a number
    - '3 is same as just 3 (number).

```
(define (show-symbol s) s)
(show-symbol '3)
(show-symbol 'A1B2)

3
'A1B2
```

Q&A

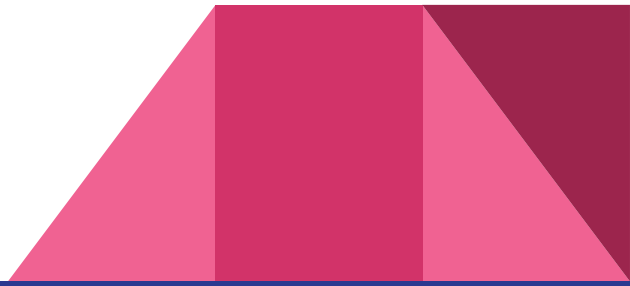
- Racket function produces output. (i.e. it returns something)
  - But no need to put 'return' keyword like c or Java.

Cautions for HW1

- Do problem analysis (contract, purpose, tests) first!!!
  - Then, start implementation
- At least two ***test cases*** for each function (not just function call). e.g., ~~(add 1 2)~~ (test (add 1 2) 3)
- All problems can be solved by using L2 and L3 slides.
  - You may use 'if'.
    - But for this HW, avoid to use 'if' to be familiar with ***conditional functions***.
  - We use 'if' later but rarely.
- Use function names as described in HW description
  - e.g., dollar->won
  - If you use not exactly same function names, our test cases will not run on your code and you will lose points.
  - Case-sensitive: dollar->won is not same as Dollar->Won

Language elements for this PL class

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



What is Type?

- Abstract definition of data (c.f. class in Java)
- Everything can be a type.
  - Window
  - Door
  - Human
  - Creature
  - Number
  - String
  - Screen
  - Projector
  - ...

Language elements for this PL class

- Type Definitions

  (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*$_{ml}$)])

  - 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*$_{jk}$.

https://docs.racket-lang.org/plai/plai-scheme.html#%28form._%28%28lib._plai%2Fmain..rkt%29._define-type%29%29

Language elements for this PL class

- Type Definitions

(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*$_{jk}$.

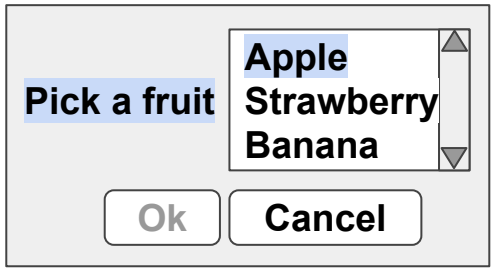
https://docs.racket-lang.org/plai/plai-scheme.html#%28form._%28%28lib._plai%2Fmain..rkt%29._define-type%29%29

Language elements for this PL class

- Example Type Definition: GUI

Language elements for this PL class

- Example Type Definition: GUI



Do not confuse!
We are not drawing this GUI by Racket
but just ***modeling*** GUI as a type

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

Language elements for this PL class

- Example Type Definition: GUI

(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*$_{ml}$)])

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

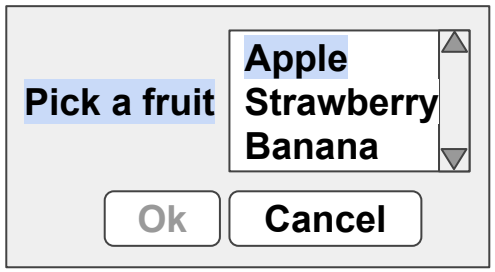
Language elements for this PL class

- Example Type Definition: GUI
  - 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*$_{jk}$.

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

Language elements for this PL class

- Example Type Definition: GUI




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

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

Language elements for this PL class

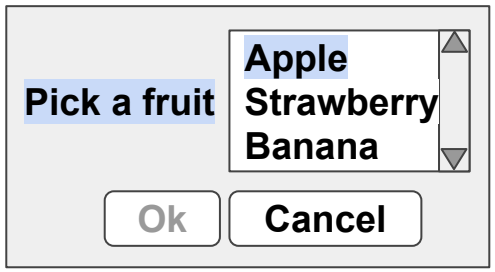
- Example Type Definition: GUI
  - **Defines predicates *type_id?* and *variant_id$_i$?*, and accessors *variant_id$_i$-field_id$_{jk}$*.**

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

(define ch (choice '("Apple" "Strawberry" "Banana") 0))
(choice? ch)
(choice-selected ch) ; [variant_id]-[field_id]
(GUI? ch)
```

Language elements for this PL class

- Example Type Definition: GUI

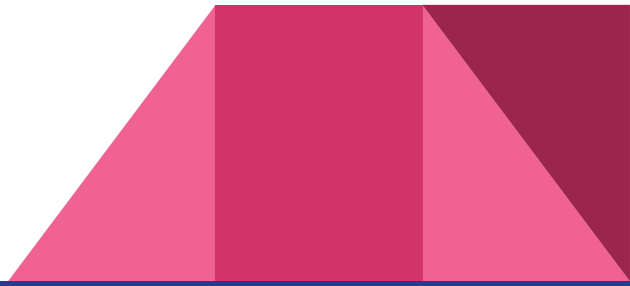



```
(define-type GUI
      [label          (text string?)]
      [button  (text string?)
                      (enabled? boolean?)]
      [choice  (items (listof string?))
                      (selected integer?)])
```

```
(define ch (choice '("Apple" "Strawberry" "Banana") 0))
(choice? ch)
(choice-selected ch) ; [variant_id]-[field_id]
(GUI? ch)
```

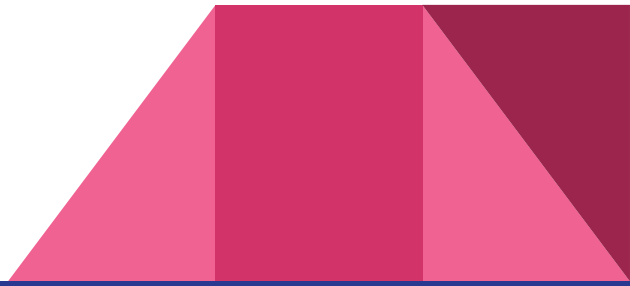
Language elements for this PL class

- Practice together: Define Circle (CRA, HAC, MIC,...) at HGU



Language elements for this PL class

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



Language elements for this PL class

- Type Deconstruction
  - From a given instance of a specific type, get required values or do a specific task for the instance.
  - Why do we need this?
    - We can access a value by using field accessors. Then, why do we need this?

Language elements for this PL class

- Type Deconstruction

  (type-case *type-id expr*

  [*variant_id*$_1$ (*field_id*$_{11}$ …) expr$_1$]

  …

  [*variant_id*$_m$ (*field_id*$_{m1}$ …) expr$_m$])

Language elements for this PL class

- Type Deconstruction

(type-case *type-id expr*
[*variant_id*$_1$ (*field_id*$_{11}$ …) expr$_1$]
…
[*variant_id*$_m$ (*field_id*$_{m1}$ …) expr$_m$])

```
; read-screen : GUI -> list-of-string
(define (read-screen g)
    (type-case GUI g
        [label (t)          (list t)]
        [button (t e?)      (list t)]
        [choice  (i s) i]))
```

Language elements for this PL class

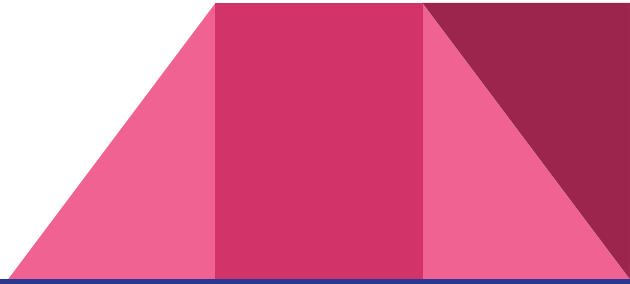
- Type Deconstruction

(type-case *type-id expr*
[*variant_id*$_1$ (*field_id*$_{11}$ …) expr$_1$]
…
[*variant_id*$_m$ (*field_id*$_{m1}$ …) expr$_m$])

```
; read-screen : GUI -> list-of-string
(define (read-screen g)
      (type-case GUI g
            [label (t)          (list t)]
            [button (t e?)      (list t)]
            [choice  (i s) i]))
(define ch (choice '("Apple" "Strawberry" "Banana") 0))
(choice? ch)
(choice-selected ch) ; [variant_id]-[field_id]
(read-screen ch)
```

Language elements for this PL class

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



Language elements for this PL class

- Lists (like arrays)
  - (list 1 2 3) or '(1 2 3)
  - A list is either the constant *null*, or it is a pair whose second value is a list.
  - null, empty
  - Use full operators
    - cons
    - list
    - append
    - first
    - rest
    - map, foldl, foldr, filter,...
      http://docs.racket-lang.org/reference/pairs.html
      https://docs.racket-lang.org/guide/Pairs__Lists__and_Racket_Syntax.html

Language elements for this PL class

- Lists

```
(cons 1 empty)                      ; => (list 1)
(cons 'a (cons 2 empty))            ; => (list 'a 2)

(list 1 2 3)                        ; => (list 1 2 3)
(list 1 2 3 empty)                  ; => (list 1 2 3 empty)

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

Language elements for this PL class

- Lists

```
(first (list 1 2 3))                ; => 1
(rest (list 1 2 3))                 ; => (list 2 3)
(first (rest (list 1 2)))           ; => 2

; '(...) creates a list. it distributes over elements:
'(1 2 3)              ; => (list 1 2 3)
'(a b)                ; => (list 'a 'b)
'((1 2) (3 4))        ; => (list (list 1 2) (list 3 4))
'10                   ; => 10 (just number)

(cons 1 2)            ; => '(1 . 2)
                      ;      which is a non-list pair
```

Language elements for this PL class

- Lists (to distinguish empty/non-empty lists)

```
(empty? empty)                    ; => #t
(empty? (cons "head" empty))      ; => #f
(cons? empty)                     ; => #f
(cons? (cons "head" empty))       ; => #t
```

Coding tip and example in Racket

- Recursion in Racket (simple example)

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

https://docs.racket-lang.org/guide/Lists__Iteration__and_Recursion.html

Coding tip and example in Racket

- Recursion in Racket (simple example)

```
; 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)

https://docs.racket-lang.org/guide/Lists__Iteration__and_Recursion.html

Coding tip and example in Racket

- Recursion in Racket (simple example)

```
; 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)))] ))
```

https://docs.racket-lang.org/guide/Lists__Iteration__and_Recursion.html

Are we learning Raket or PLT???

Why do we need Type Deconstruction?

JC

jcnam@handong.edu

https://lifove.github.io

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or created by JC based on the main text book.