Programming Fundamentals 1: Midterm Exam

Prof. Carlo A. Furia, USI · 4 November 2022

Exercise 1: Types and values (points: 37)

For each of the following *expressions*, determine if the expression is **valid** (that is, it can be evaluated without errors); if it is valid, determine its **type** and its **value**.

EXPRESSION	VALID?	TYPE	VALUE
`(3 + 7)`	NO		
`(/ 4 1)`	YES	`Number`	`4`
`(cons '() '())`	YES	`List`	`(list '())`
`(cons '())`	NO		
`(cons? 3 '())`	NO		
`(cons? '())`	YES	`Boolean`	`#false`
`(posn? (list "x" "y"))`	YES	`Boolean`	`#false`
`(posn 3 4)`	NO		
`(make-posn '() '())`	YES	`Posn`	`(make-posn '() '())`
`(list (cons 3 '()))`	YES	`List`	`(list (list 3))`
`(cond [#true #false])`	YES	`Boolean`	`#false`
`(cond [#false 3])`	NO		
`(posn-x (make-posn "y" 3))`	YES	`String`	`"y"`
`(string-append "3" "+" "4")`	YES	`String`	`"3+4"`

E-1

Exercise 2: Data types (points: 24)

Consider all possible lists of numbers that are **not empty** and have an **even** number of elements; that is, all lists of numbers consisting of 2, or 4, or 6, or . . . elements. (Note that the constraint is only on the **length** of the list; the content of the list can be any numbers.)

- Define a suitable **data type** `EvenList<Number>` that consists of all such lists.
- Define three **constants** `EL1`, `EL2`, and `EL3` bound to three different **instances** of the data type `EvenList<Number>`.

E-2

Solution 2:

```
; an EvenList<Number> is one of the following:
;  - (cons Number (cons Number '()))  ; base case: 2-element list
                                      ; recursive case: list with 4, 6, 8, ... elements
;  - (cons Number (cons Number EvenList<Number>))

(define EL1 (cons 1 (cons 2 '())))
(define EL2 (cons 1 (cons 2 (cons 3 (cons 4 '())))))
(define EL3 (list 1 2 3 4 5 6 7 8))
```

Common mistakes:

- Defining data type `EvenList` simply by means of a verbal description (basically repeating the same description given in the exercise description).
- Defining lists of even numbers instead of lists of even length: the difference was explicitly pointed out in the exercise description.

S-2

Exercise 3: Templates (points: 30)

Consider the following definition of a data type `Mixture`:

```
; a Mixture is one of:
;   - a non-empty String
;   - a negative Number between -10 and -3 (both included)
;   - (make-posn Boolean Boolean)
```

Write a **header** and a **template** for a function `fmix` with signature
`fmix : Mixture -> Number`.

E-3

Solution 3: The following template is sufficient for a *non-checked* function: if we assume `mix` matches the input type `Mixture`, then `mix` is a `String` if and only if it is a non-empty `String`, and so on for the other cases of the conditional.

Adding more specific checks (for example, for the first conditional case, using the question: `(and (string? mix) (> (string-length mix) 0))` is also correct.

```
; header
(define (fmix mix) 0)

; template
(define (fmix mix)
  (cond
    [(string? mix)
     ... mix ...]
    [(number? mix)
     ... mix ...]
    [else
     ... (posn-x mix) ... (posn-y mix) ...]))
```

Common mistakes:

- In the template, checking the second case of type `Mixture` by means of something like `(<= -10 mix -2)`: since `<=` fails if its arguments are not numbers, the third case of the template becomes effectively unreachable.
- Omitting the selector functions `(posn-x mix)` and `(posn-y mix)` in the "answer" part of the last case.

S-3

Exercise 4: Tests and implementations (points: 27)

Consider the following function `pl`, which inputs a list of numbers `lon`:

```
(define (pl lon)
  (cond
    [(not (empty? (rest lon)))
     (cond
       [(< (first lon) 0)
        (+ (first lon) (pl (rest lon)))])]
    [else
     (first lon)]))
```

- Using `check-error`, write **3 different examples** of input (that is, three different instances of lists of numbers) on which `pl` **fails** (that is, it raises an error if called on that input).
- Using `check-expect`, write **3 different examples** of input (that is, three different instances of lists of numbers) on which `pl` **evaluates without errors**; for each of them, also specify the **returned value**.

E-4

Solution 4: Function `pl` evaluates without errors on all non-empty lists that consist of only negative numbers—except for the last element that can be any number. It returns the sum of all elements in the list. Note that the exercise explicitly asks to give examples of inputs that are list of numbers, so this was required to get full marks.

```
(check-error (pl '()))
(check-error (pl (list 1 -2 -3 -4 5)))
(check-error (pl (list -1 2 -3 -4 5)))
(check-error (pl (list -1 -2 0 -4 5)))

(check-expect (pl (list 3))
              3)
(check-expect (pl (list -1 -2 -3 -4 5))
              (+ -1 -2 -3 -4 5))
(check-expect (pl (list -1 -2 -3 -4 -5))
              (+ -1 -2 -3 -4 -5))
```

S-4

Exercise 5: Structures (points: 21)

Consider the following definitions of constants and tests (`check-expect`).

```
(define S
  (make-str
   (make-x 0 1)
   (x? 0)
   (x? (make-x 1 1))))

(define T
  (make-x
   (make-str (str? "a") (str? "b") 10)
   8))

(check-expect
  (+ (x-a (str-y S)) (x-b T))
  8)

(check-expect
 (str-a S)
 #false)
```

```
(check-expect
 (str-b S)
 #true)
```

Using `define-struct`, define two **structures** `str` and `x` such that all constants and tests above can be evaluated *without errors*.

You are not required to provide a full type declaration; it's sufficient to write down two correct usages of `define-struct`.

E-5

Solution 5:

- By inspecting how constructors `make-str` and `make-x` are used, we see that `str` has 3 fields, whereas `x` has two fields.
- By looking at the selector functions, we see that `str`'s fields are called `y`, `a`, and `b`; and that `x`'s fields are called `a` and `b`.
- Since `(str-a S)` and `(str-b S)` are booleans, field `str`'s field `y` must be the first one. Since `(x? 0)` in `S` is `#false`, field `a` must be the second field, and thus `b` is the third field.
- Finally, the first test implies that `x`'s field `b` must be a number, and hence it must be the second one (in `T`'s definition).

In all, we have the following `define-struct`s:

```
(define-struct str [y a b])
(define-struct x [a b])
```

S-5