Questions 1-2

Bookmark this page

Consider the following definitions for `Student` and `ListOfStudent`:

```
(define-struct student (name id))
;; Student is (make-student String Natural)
;; interp. a student with name and student id
(define S1 (make-student "Eva" 3124))
(define S2 (make-student "John" 7893))

#;
(define (fn-for-student s)
  (... (student-name s)
       (student-id s)))
```

```
;; ListOfStudent is one of:
;; - empty
;; - (cons Student ListOfStudent)
;; interp. a list of students
(define LOS1 empty)
(define LOS2 (cons S1 empty))
(define LOS3 (cons S1 (cons S2 empty)))

#;
(define (fn-for-los los)
  (cond [(empty? los)(...)]
        [else
         (...(fn-for-student (first los))
             (fn-for-los (rest los)))]))
```

Suppose we would like to design a function that consumes a list of students and produces a list of student cards, where each student card contains the name and ID of the student. Assume a student card is simply a string of the form:

`"<student name> <student id>"`

For example:

```
(student-cards (cons (make-student "John" 7893 (cons (make-student "Eva" 3124) empty)))
```

should produce

```
(cons "John 7893" (cons "Eva 3124" empty))
```

Question 1

1/1 point (graded)

Here is a partial design of the function `student-cards`:

```
;; ListOfStudent -> ListOfString
;;produces a list of student cards, each of the form "<name><id>"

(check-expect (student-cards empty) empty)
(check-expect (student-cards LOS2) (cons "Eva 3124" empty))
```

Do we need more `check-expects`?

- (•) yes
- ( ) no

✔

Explanation
We need a `check-expect` that is at least two elements long, because it will show us any errors in the natural recursion.

Submit Show Answer

Answers are displayed within the problem

Question 2

1/1 point (graded)

What is wrong, or could be improved about the following function definition for `student-cards`?

```
(define (student-cards los)
  (cond [(empty? los) empty]
        [else
         (cons (string-append (student-name (first los))
                              " "
                              (number->string (student-id (first los))))
               (student-cards (rest los)))]))
```

- ( ) Nothing since all tests pass
- ( ) The base case is wrong

- (•) A helper function that operates on `Student` is missing
- ( ) A helper function that operates on `ListOfStudent` is missing

✔

Explanation
Since the type of `(first los)` is `Student` which is not primitive, we need to follow the reference rule and design a function that consumes a student and produces the corresponding student card. The resulting `student-cards` function will then be:

```
(define (student-cards s)
  (cond [(empty? s) empty]
        [else
         (cons (student-card (first los))
               (student-cards (rest los)))]))
```

Go ahead and complete the design of `student-card`

Submit

Show Answer

Answers are displayed within the problem

Previous Next