Quiz

Bookmark this page

For this multiple choice design quiz, download the starter and complete the problems.

Once you have finished, answer the multiple choice questions about your design.

Unlike the lecture questions, you will only have one attempt to answer each question in the quiz, so make sure to read each answer carefully before selecting one and pressing submit.

Question 1

1/1 point (graded)

Questions 1-3 deal with PROBLEM 1 in the starter file.

How many accumulators are necessary for the design of `longest-length`?

- 0
- [✓ 1]
- 2
- 3

✓

Explanation
We need an accumulator to keep track of the longest string so far.

Submit You have used 1 of 1 attempt Show Answer

Answers are displayed within the problem

Question 2

0/1 point (graded)

The following are pairs of Type Comments and Invariants for accumulators.

Select all that are valid accumulators for `longest-length`

- [✓ `;; acc is Natural; the length of the longest string so far in los`] ✓
- `;; acc is Boolean; true if the current string is the longest so far in los`
- `;; acc is String; the first letter of the longest string so far in los`
- `;; acc is String; the longest string so far in los` ✓

✗

Explanation
We need to know the length of the longest string in the list.
If the accumulator is Natural, and keeps track of the length of the longest string so far, we have exactly what we need.
If the acculutor is String, and keeps track of the longest string so far, we can take the length at the end.
The other two options will not keep track of the necessary information.

Submit You have used 1 of 1 attempt Show Answer

Answers are displayed within the problem

Question 3

1/1 point (graded)

If your accumulator is a Natural, representing the length of the longest string in the list, what should its initial value be?

- empty

○ ""

[✓ 0]

○ 1

✓

Submit You have used 1 of 1 attempt

Show Answer

Question 4

1/1 point (graded)

Question 4 deals with PROBLEM 2 in the starter file.

Which of the following is the correct way to update the accumulator(s) in the natural recursion? Assume the function consumes the list first, then any accumulators.

○ `(fibonacci? (rest lon) (first lon))`

○ `(fibonacci? (rest lon) n-1)`, where `n-1` was the element before `(first lon)`

○ `(fibonacci? (rest lon) n-2 n-1)`, where `n-1` was the element before `(first lon)`, and `n-2` was the element before that.

[✓ `(fibonacci? (rest lon) n-1 (first lon))`, where `n-1` was the element before `(first lon)`]

✓

Explanation
`fibonacci?` needs to know about the two elements before the list. If the list is `(rest lon)` the two previous elements are (first lon), and `n-1` which was the element before `(first lon)`.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 5

1/1 point (graded)

Questions 5-7 deal with PROBLEM 3 in the starter file.

What type of accumulator is necessary to refator `fact` to make it tail recursive?

○ a context preserving accumulator, to keep track of the original value of `n`

○ a result so far accumulator because the call to `*` is not in tail position

[✓ a result so far accumulator because the recursive call to `fact` is not in tail position]

○ a worklist accumulator

✓

Explanation
A function is tail recursive if all recursive calls are in tail position, so we need a rsf accumulator so that `fact` can be in tail position.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 6

1/1 point (graded)

Which of the following is true about your refactored design of `fact` compared to the original?

○ the results of both the `zero?` case and the `else` case were unchanged

○ the result of the the `zero?` case changed, but the result of the `else` case was unchanged

○ the resultt of the `zero?` case was unchanged, but the result of the `else` case changed

[✓ the result of both the `zero?` case and the `else` case changed]

✓

Explanation
The resuts of both cond cases change in the refactoring of `fact`.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 7

1/1 point (graded)

Drag the pieces of code from the scroll bar to complete the refactored design of `fact`. You do not need to use every piece of code, and you can use each piece more than once.

```
(define (fact n)
  (local [(define (fact n  rsf  )
            (cond [(zero? n)  rsf   ]
                  [else
                    (fact (sub1 n)(* n rsf))]))]
    (fact n    1   )))
```

◀	rsf	1	0	(* n rsf)	n		▶

✓

Answer:

Submit You have used 1 of 1 attempt

✓ Correct (1/1 point)

Question 8

1/1 point (graded)

Questions 8-11 deal with PROBLEM 4 in the starter file.

In the encapsulated template for fn-for-region, which functions have elements that are not tail recursive, and will need to be refactored?

- [✓] `fn-for-region`
- [ ] `fn-for-type`
- [✓] `fn-for-lor`

✓

Explanation
`fn-for-type` has no recursive calls, so it has no elements that are not tail recursive.
Both `fn-for-region` and `fn-for-lor` have recursive calls that are not in tail position.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 9

1/1 point (graded)

Select the accumulators that are necessary in the tail-recursive design of `count-regions`?

- [✓] ;; todo is (listof Region); a worklist accumulator
- [ ] ;; todo is Region; a worklist accumulator
- [ ] ;; rsf is Region; the last region visited
- [✓] ;; rsf is Natural; the number of regions seen so far
- [ ] ;; visited is (listof Region); all regions seen so far

✓

Explanation
To make this function tail recursive, we need a worklist accumulator that is a list fo the Regions we need to visit, and a result so far accumulator that is the number of regions seen so far.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 10

1/1 point (graded)

Where do we update `todo`, the worklist accumulator? Select all places where a function is passed a new value of `todo`.

- [ ] there is no worklist accumulator
- [x] `fn-for-region`
- [x] `fn-for-lor`
- [ ] `fn-for-type`

✓

Explanation
In `fn-for-region` we append `region-subregions` and `todo`.
In `fn-for-lor` we take `(rest todo)`.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Question 11

1/1 point (graded)

Where do we update `count`, the result so far accumulator? Select all places where a function is passed a new value of `count`.

- [ ] there is no result so far accumulator
- [x] `fn-for-region`
- [ ] `fn-for-lor`
- [ ] `fn-for-type`

✓

Explanation
In `fn-for-region` we append we add one to `count`.

Submit You have used 1 of 1 attempt

Show Answer

Answers are displayed within the problem

Previous Next