dan-then-dog in 4 lines?

You are subscribed. Unsubscribe

No tags yet. + Add Tag

Sort replies by: Oldest first Newest first Most popular

KL Tah · 3 days ago

I have a rather inefficient implementation of dan-then-dog in 6 lines that involves a circular looking if then else. I can see that the arguments could be anything. it's whats consed on that's important. . if the argument is dog then cons dan on to the pair with the function calling dan as argument. if dan is the argument then cons dog on to the pair with dog as argument. i can sense that this is rather wasteful but i can't quite put finger on what exactly...

0 · flag

Pierpaolo Bernardi · 3 days ago

You can also (cons dan (cons dog dan-then-dog)), with no if required (the grader liked this solution, but I don't remember how was the sample solution)

0 · flag

KL Tah · 3 days ago

hmm.. i'm not sure what you mean. since the deadline has already passed, he's my really inefficient version. i hope to improve it to 4 lines. any tips?

```
(define dan-then-dog
  (letrec ([f (lambda (d)
              [EDIT 2014-11-30T22:20Z, PE: Code removed]
    (lambda () (f "dog"))))
```

0 · flag

Mikhail Maltsev · 3 days ago

Tip: use two mutually recursive lambdas, use one of them (i.e. as a variable binding) as the result.

The sample solutions are shown in peer evaluation. Or is it unavailable for those who missed the deadline (I'm not aware, what the policy is in this case)?

1 · flag

Pierpaolo Bernardi · 3 days ago

It's not particularly inefficient. This solution is OK. You can use symbols instead of strings. 'dan instead of "dan".

0 · flag

A post was deleted

+ Comment

Chad Miller · 3 days ago

APPROVED

I defined a `dog-then-dan` and made them mutually recursive. :)

1 · flag

+ Comment

KL Tah · 2 days ago

aha! that was what was on the tip of my tongue. state machine. only thing is we haven't learned how to do mutual recursion in racket. or do we also have an "and" keyword analogous to ML?

0 · flag

Chad Miller · 2 days ago

It's nothing that complicated. I actually defined it as a nested `define`, though simply having two separate `define` statements would also work; in Racket what matters is that the name is defined at the time the code is run, so definitions can reference each other without issue.

1 · flag

Charilaos Skiadas COMMUNITY TA · 2 days ago

There's also letrec.

1 · flag

Peter Eriksen COMMUNITY TA · a day ago

> *do we also have an "and" keyword analogous to ML?*

The closest analog in this context is letrec.

Simultaneous variable declarations in ML

```
val id1 = e1
and ...
and idn = en
```

evaluates `e1` to `en`, and then binds `id1` to `idn` to the values. This is like Racket's let.

In simultaneous function declarations in ML

```
fun id1 arg1 = e1
and ...
and idn argn = en
```

each function may call any of the other `id1` to `idn` functions. Racket has a more general variant of this construct, letrec.

The normal `let` in ML evaluates each binding in turn, and each expression `ei` may reference an earlier binding, but not a later one:

```
let val v1 = e1
    ...
    val vn = en
in e0 end
```

This is like let* in Racket.

1 · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B I Link <code> Pic Math Edit: Rich Preview

☐ Make this post anonymous to other students

☑ Subscribe to this thread at the same time

Add post