BYU CS classes

Objective:

Learn more about higher-order functions, especially using them as decorators for other functions.

Preparation:

You will again use DrRacket, which you should have downloaded and installed for the previous labs.

Deliverables:

For this lab, you will need to implement the following decorator functions in Racket:

- `default-parms` - decorates a function to add default parameter values
- `type-parms` - decorates a function to enforce parameter type checking

You will also chain these together to add both parameter type checking and default parameters to a function.

default-parms

```
(define (default-parms f values) ...)
```

where `f` is a function of arbitrary arity, `values` is a list of default parameter values of the same length as the arity of `f`, and the result is a function that behaves **exactly** like `f` but fills in corresponding default parameter values from the `values` list if given fewer parameters than expected.

For example, if `f` takes two parameters,

```
(define g (default-parms f (list 42 99))
```

would result in a function `g` such that `(g)` behaves the same as `(f 42 99)`, `(g 8)` behaves the same as `(f 8 99)`, and `(g 8 9)` behaves the same as `(f 8 9)`.

Hint: You might find it useful to use the Racket function `list-tail`, which returns the rest of a list after some number of elements. For example, `(list-tail (list 1 2 3 4 5) 2)` returns the list `(3 4 5)`.

type-parms

```
(define (type-parms f types) ...)
```

where `f` is a function of arbitrary arity, `types` is a list of type predicates (e.g., `number?`, `string?`, etc.) of the same length as the arity of `f`, and the result is a function that behaves **exactly** like `f` but first checks each parameter passed in to make sure it is the correct type.

For example, if `f` takes two parameters,

```
(define g (type-parms f (list number? string?))
```

would result in a function `g` such that `(g 0 “hello”)` behaves just like `(f 0 “hello”)`, but passing anything but a number for the first parameter or anything other than a string for the second parameter results in an error message.

Hint: You may want to write a small helper function that takes a list of values and a list of type predicates, then makes sure each of the values satisfies the respective type predicate. This is straightforward to write recursively. Can you write it in shorter form by mapping a two-parameter function pairwise on the lists?

Chaining Decorators

If `default-parms` and `type-parms` are working correctly, you should be able to chain them together like this:

```
(define g (default-parms
            (type-parms
             f
             (list number? string?))
            (list 0 "")))
```

would result in a function `g` that fills in any missing parameters with `0` and "" respectively, and then verifies that the passed parameters are a number and a string.

Use your `new-sin` function from your first assignment, `type-parms`, and `default-parms` to create a function called `new-sin2` that behaves just like `new-sin` but

- uses `0` and `'radians` as defaults for the parameters, and
- verifies that the first parameter is a number and that the second parameter is a symbol.

Notes:

You again **do not** need to bulletproof the code to enforce proper inputs. Your code only needs to return correct values given correct inputs.

Hints:

Remember from class that our decorators have the same basic form:

```
(define (<name of the decorator> f ...)
  (lambda args
    ...
    (apply f args)))
```

---

cs330_f2016/rackethof2.txt · Last modified: 2016/09/13 08:50 by morse