1 Challenge 1: `optimize-self-reference`

It's common to see this pattern

```
(letrec ([f.1 (lambda (cp.2 x.3)
                (bind-free (cp.2 f.1)
                  ...))])
  ...)
```

if `f.1` calls itself. But `f.1` in the closure is redundant: under any circumstances, it's the same thing as `cp.2`. So you can safely remove it from the closure and replace every `f.1` in the body with `cp.2`.

To make sure your optimization is working, we suggest you writing a analyzing pass `analyze-closure-size` to count the number of closure created and free variables. It should run just before `introduce-procedure-primitives`, after all the optimizations you've done. You can also count lines of code before the final pass as an *approximate* measure of code quality.

2 Challenge 2: `optimize-free`

In Scheme, procedures are first-class citizens. So we need a uniform representation, namely closure, to pass procedures around. But sometimes this is not necessary if we only use a procedure locally. Say we have

```
(letrec ([f.1 (lambda () (g.2))]
         [g.2 (lambda () (f.1))])
  (f.1))
```

after `convert-closure` and `optimize-known-call` it becomes

```
(letrec ([f$1 (lambda (cp.3)
                (bind-free (cp.3 g.2)
                  (g$2 g.2)))]
         [g$2 (lambda (cp.4)
                (bind-free (cp.4 f.1)
                  (f$1 f.1)))])
  (f$1 f.1))
```

This is nonsense, but you get the idea. We can well write this in the following equivalent form

```
(letrec ([f$1 (lambda ()
                (bind-free (dummy) ; 'dummy' just makes the wrapper happy
                  (g$2)))]
         [g$2 (lambda ()
                (bind-free (dummy)
                  (f$1)))])
  (f$1))
```

because the only thing we do to `f.1` or `g.1` is to apply it. If a procedure *is* used otherwise, we say it *escapes*. We can avoid allocating closures for non-escape procedures.

Firstly we need to recognize non-escape procedures. The following simple criteria should suffice: `f.1` is non-escape if it's only used in the form `(f$1 f.1 ...)`. We add a pass `uncover-well-known` to record them in the `(closures (...) (well-known (...) ...))` form. The name "well-known" echoes `optimize-known-call`.

Then we remove their corresponding closures. Only those without free variables can be removed.

Or is it? the above example does not satisfies this, but we can still remove `f.1` and `g.2`. So if all the free variables are well-known, we can still remove it.

Or is it? consider the following example:

```
(let ([x.5 10])
  (letrec ([f$1 (lambda (cp.3)
                  (bind-free (cp.3 g.2)
                    (g$2 g.2)))]
           [g$2 (lambda (cp.4)
                  (bind-free (cp.4 x.5)
                    x.5))])
    (f$1 f.1)))
```

`g.2` is well-known, but we cannot remove `g.2` because of `x.5`, and it prevent us from removing `f.1`. So you need to do some simple analysis on the graph to identify truly well-known procedures.

You can also do something so called "lambda lifting", but with caveats. The idea is to pass free variables as additional parameters. For example, the above code becomes

```
(let ([x.5 10])
  (letrec ([f$1 (lambda (x.6)
                  (bind-free (dummy)
                    (g$2 x.6)))]
           [g$2 (lambda (x.7)
                  (bind-free (dummy)
                    x.7))])
    (f$1 x.5)))
```

This is not always an optimization because if you call a procedure with a lot of free variables many times, they are repeatedly moved to stack as parameters and fetched by the procedure. But a closure avoids this. This approach also requires "propagating" additional parameters across procedures.