CS 5510 Homework 7

Due: Wednesday, October 21st, 2015 11:59pm

Start with `letcc2.rkt`, which is like `letcc.rkt`, but it changes the representation of function expressions, application expressions, and closure values to support a list of arguments—which is the boring work behind part 2 below. The `parse` function in `letrec2.rkt` still matches only single-argument functions and applications, so you'll have to change that when you're ready to work on part 2.

Part 1 — More Arithmetic Operators

Add two new arithmetic operators: `neg` and `avg`. The `neg` form takes a single number and returns its negation; the `avg` form (which evaluates its subexpressions left-to-right) takes three numbers and returns the average of the numbers. Also, add `if0` as usual.

```
<Expr> = <Num>
       | <Sym>
       | {+ <Expr> <Expr>}
       | {* <Expr> <Expr>}
       | {lambda {<Sym>} <Expr>}
       | {<Expr> <Expr>}
       | {let/cc <Sym> <Expr>}
       | {neg <Expr>}
       | {avg <Expr> <Expr> <Expr>}
       | {if0 <Expr> <Expr> <Expr>}
```

Implement `neg` and `avg` as core forms, instead of sugar. More generally, implement them without relying on the interpreter's existing implementation of addition and multiplication (e.g., don't generate an `doAddK` continuation in the process of interpreting negation or averaging).

As in recent previous homeworks, provide `interp-expr`, which takes an expression, interprets it with an empty substitution, and produces either a number S-expression or `` `function``; return `` `function`` when the result is a closure or continuation.

```
(test (interp-expr (parse '{neg 2}))
      '-2)
(test (interp-expr (parse '{avg 0 6 6}))
      '4)
(test (interp-expr (parse '{let/cc k {neg {k 3}}}))
      '3)
(test (interp-expr (parse '{let/cc k {avg 0 {k 3} 0}}))
      '3)
(test (interp-expr (parse '{let/cc k {avg {k 2} {k 3} 0}}))
      '2)
(test (interp-expr (parse '{if0 1 2 3}))
      '3)
(test (interp-expr (parse '{if0 0 2 3}))
      '2)
(test (interp-expr (parse '{let/cc k {if0 {k 9} 2 3}}))
      '9)
```

Part 2 — Functions that Accept Multiple Arguments, Again

Change your interpreter to support multiple or zero arguments to a function, and multiple or zero arguments in a function call:

```
<Expr> = <Num>
       | <Sym>
       | {+ <Expr> <Expr>}
       | {* <Expr> <Expr>}
       | {lambda {<Sym>*} <Expr>}
       | {<Expr> <Expr>*}
       | {let/cc <Sym> <Expr>}
       | {neg <Expr>}
       | {avg <Expr> <Expr> <Expr>}
       | {if0 <Expr> <Expr> <Expr>}
```

Assume that each argument `<Sym>` is distinct for a `lambda` expression. All continuations still accept a single argument.

```
(test (interp-expr (parse '{{lambda {x y} {+ y {neg x}}} 10 12}))
      '2)
(test (interp-expr (parse '{lambda {} 12}))
      `function)
(test (interp-expr (parse '{lambda {x} {lambda {} x}}))
      `function)
(test (interp-expr (parse '{{{lambda {x} {lambda {} x}} 13}}))
      '13)

(test (interp-expr (parse '{let/cc esc {{lambda {x y} x} 1 {esc 3}}}))
      '3)
(test (interp-expr (parse '{{let/cc esc {{lambda {x y} {lambda {z} {+ z y}}}
                                          1
                                          {let/cc k {esc k}}}}
                            10}))
      '20)
```

Part 3 — Extra Credit: Faster Exceptions

This exercise is optional, for extra credit.

Extend your interpreter to bring back `try`:

```
<Expr> = ...
       | {try <Expr> {lambda {} <Expr>}}
```

When raising an exception for an erroneous expression, instead of searching the continuation for a `tryK` continuation, arrange for the interpreter to keep track of the current handler so that an exception is raised in constant time (no matter how long the continuation between the error and the enclosing `try`).

Last update: Tuesday, October 6th, 2015
mflatt@cs.utah.edu