CS 5510 Homework 6

Due: Wednesday, October 7th, 2015 11:59pm

Implement an interpreter with lazy evaluation and the following grammar:

```
<Expr> = <Num>
       | <Sym>
       | {+ <Expr> <Expr>}
       | {* <Expr> <Expr>}
       | {lambda {<id>} <Expr>}
       | {<Expr> <Expr>}
       | {let {[<id> <Expr>]} <Expr>}
       | {if0 <Expr> <Expr> <Expr>}
       | {cons <Expr> <Expr>}
       | {first <Expr>}
       | {rest <Expr>}
```

That is, a language with single-argument functions and application, an if-zero conditional, and `cons`, `first`, and `rest` operations. (The language does not include recursive bindings or records.) The `cons` operation does not require its second argument to be a list, so `rest` can also return a non-list.

Implement your interpreter with the `plai-typed` language, not a lazy language.

Evaluation of the interpreted langauge must be lazy, however. In particular, if a function never uses the value of an argument, then the argument expression should not be evaluated. Similarly, if the first or rest of a cons cell is never needed, then the first or rest expression should not be evaluated.

Start with `more-lazy.rkt`. Expand the `parse` function to support the new forms: `if0`, `cons`, `first`, and `rest`. Also, as in HW 5, provide an `interp-expr` function; the `interp-expr` wrapper for `interp` should take an expression and return either a number S-expression, `` `function `` for a function result, or `` `cons `` for a cons result. (Meanwhile, the `interp` function should never return the symbol `` `cons ``, just like the starting `interp` function never returns the symbol `` `function ``.)

```
(test (interp-expr (parse '10))
      '10)
(test (interp-expr (parse '{+ 10 17}))
      '27)
(test (interp-expr (parse '{* 10 7}))
      '70)
(test (interp-expr (parse '{{lambda {x} {+ x 12}}
                             {+ 1 17}}))
      '30)

(test (interp-expr (parse '{let {[x 0]}
                              {let {[f {lambda {y} {+ x y}}]}
                                {+ {f 1}
                                   {let {[x 3]}
                                     {f 2}}}}}))
      '3)

(test (interp-expr (parse '{if0 0 1 2}))
      '1)
(test (interp-expr (parse '{if0 1 1 2}))
      '2)

(test (interp-expr (parse '{cons 1 2}))
      `cons)
(test (interp-expr (parse '{first {cons 1 2}}))
      '1)
(test (interp-expr (parse '{rest {cons 1 2}}))
      '2)

;; Lazy evaluation:
(test (interp-expr (parse '{{lambda {x} 0}
                            {+ 1 {lambda {y} y}}}))
      '0)
(test (interp-expr (parse '{let {[x {+ 1 {lambda {y} y}}]}
                             0}))
      '0)
(test (interp-expr (parse '{first {cons 3
                                        {+ 1 {lambda {y} y}}}}))
      '3)
(test (interp-expr (parse '{rest {cons {+ 1 {lambda {y} y}}
                                       4}}))
      '4)
(test (interp-expr (parse '{first {cons 5
                                        ;; Infinite loop:
                                        {{lambda {x} {x x}}
                                         {lambda {x} {x x}}}}}))
      '5)

(test (interp-expr
       (parse
        '{let {[mkrec
                ;; This is call-by-name mkrec
                ;;  (simpler than call-by-value):
                {lambda {body-proc}
                  {let {[fX {lambda {fX}
                              {body-proc {fX fX}}}]}
                    {fX fX}}}]}
            {let {[fib
                   {mkrec
                    {lambda {fib}
                      ;; Fib:
                      {lambda {n}
                        {if0 n
                             1
                             {if0 {+ n -1}
                                  1
                                  {+ {fib {+ n -1}}
                                     {fib {+ n -2}}}}}}}}]}
              ;; Call fib on 4:
              {fib 4}}}))
      '5)

(test (interp-expr
       (parse
        '{let {[mkrec
                ;; This is call-by-name mkrec
                ;;  (simpler than call-by-value):
                {lambda {body-proc}
                  {let {[fX {lambda {fX}
                              {body-proc {fX fX}}}]}
                    {fX fX}}}]}
           {let {[nats-from
                  {mkrec
                   {lambda {nats-from}
                     ;; nats-from:
                     {lambda {n}
                       {cons n {nats-from {+ n 1}}}}}}]}
             {let {[list-ref
                    {mkrec
                     {lambda {list-ref}
                       ;; list-ref:
                       {lambda {n}
```

```
                           {lambda {l}
                             {if0 n
                                  {first l}
                                  {{list-ref {+ n -1}} {rest l}}}}}}}]}
                 ;; Call list-ref on infinite list:
                 {{list-ref 4} {nats-from 2}}}}}))
        '6)
```

Last update: Tuesday, September 29th, 2015
mflatt@cs.utah.edu