Homework #5: Interpreter for The Functional Register Language in the Environment Model 2019

Out: Thursday, June 06, 2019, Due: Friday, June 14, 2019, 15:59

This assignment is non-mandatory, and the grading for it will be counted as BONUS points. However, you need to get a grade of at least a 60 for these bonus points to count.

This homework has three parts:

1. A completion of the implementation of the simple ROL language that we have started to implement in the previous assignment – In the environment model.
2. Implementing with expressions as a syntactic sugar (through call and fun).

An interpreter for FLANG that supports `the environment model` appears at the end of this file.

Administrative

The language for this homework is:

```
#lang pl
```

Reminders (this is more or less the same as the administrative instructions for the previous assignment):

Important: the grading process requires certain bound names to be present. These names need to be global definitions.

This homework is for individual work and submission.

Integrity: Please do not cheat. You may consult your friends regarding the solution for the assignment. However, you must do the actual programming and commenting on your own!! This includes roommates, marital couples, best friends, etc… I will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Comments: Submitted code for each question should include at least two lines of comments with your personal description of the solution, the function and its type. In addition, you should comment on the process of solving this question – what were the main difficulties, how you solved them, how much time did you invest in solving it, did you need to consult others. **A solution without proper and elaborate PERSONAL comments describing your work process may be graded 0.**

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests.

Note: Your tests should cover your whole code; otherwise the server will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the server will run its own tests over your code, which means that you will not be able to submit code that does not work. Reminder: this means that most of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests.

General note: Code quality will be graded. Write clean and tidy code. Consult the Style Guide, and if something is unclear, ask questions on the course forum.

The code for all the following questions should appear in a single .rkt file named <your ID>_3 (e.g., 333333333_3 for a student whose ID number is 333333333).

1. Re-Implementing the ROL language in the Environment Model

In the previous assignment, you have implemented an interpreter for the language "ROL" in the **substitution** model. Here, you need to re-implement it in the environment model.

Clearly, there should be no change in the interface of a user (programmer) with your ROL interpreter. Thus, the following tests should still go through (all tests given in the previous to last assignment should also work).

```
(test (run "{ reg-len = 3
              {with {identity {fun {x} x}}
                {with {foo {fun {x} {or x {1 1 0}}}}
                  {call {call identity foo} {0 1 0}}}}}")
      => '(1 1 0))
 (test (run "{ reg-len = 3
               {with {x {0 0 1}}
```

```
           {with {f {fun {y} {and x y}}}
             {with {x {0 0 0}}
               {call f {1 1 1}}}}}}")
     => '(0 0 1))

(test (run "{ reg-len = 4
           {with {foo {fun {z} {if {maj? z} z {shl z}}}}
             {call foo {if {maj? {0 0 1 1}} {shl {1 0 1 1}} {1 1 0 1}}}}}")
     => '(0 1 1 1))
```

2. "with" as a Syntactic Sugar

Here, you are asked to slightly change the implementation of the eval function, and specifically, the way it handles the With variant. Rather than directly eval such an AST, you are to translate it into a Call tree (also using Fun) and evaluating it as that. Clearly, again, there should be no change in the interface of a user (programmer) with your ROL interpreter. Hence, all the above tests should still work.

Note: eval should recursively call itself only once in the line dealing with the With variant (rather than twice, as was the case so far).

```
---<<<FLANG-ENV>>>-----------------------------------------------------

  ;; The Flang interpreter, using environments

  #lang pl

  #|
  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }

  Evaluation rules:
    eval(N,env)                = N
    eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
    eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
    eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
    eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
    eval(x,env)                = lookup(x,env)
```

```
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                              if eval(E1,env1) = <{fun {x} Ef}, env2>
           = error!           otherwise
|#

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Types for environments, values, and a lookup function

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])
```

```
(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (NumV (op (NumV->number val1) (NumV->number val2))))

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV bound-id bound-body f-env)
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
      [(NumV n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))
```