Homework #5: Environment based Interpreter – Implementing a Global Environment (Non Mandatory)

Out: Sunday, June 14, 2020, Due: Thursday, June 25, 2020, 23:55

Administrative

The language for this homework is:

```
#lang pl
```

The homework is basically about implementing a global enviroenment and adding it to the implementation of the FLANG interpreter– In the environment model (use the interpreter we have seen in class as a basis for your code – see here).

Important: the grading process requires certain bound names to be present. These names need to be global definitions. The grading process *cannot* see names of locally defined functions.
This homework is non-mandatory and only individual submissions are allowed.

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 (as pairs, if you so choose)!! 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 comments may be graded 0.** In general, comments should appear above the definition of each procedure (to keep the code readable).

If you choose to consult any other person or source of information, this **must be** clearly declared in your comments.

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. ***Important:*** 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 much of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests. Note that your tests should not only cover the code, but also all end- cases and possible pitfalls.
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>_5 (e.g., 333333333_5 for a student whose ID number is 333333333).

1. Introduction

In class we saw the **FLANG** language and **ENV** based interpreter. In this work, we will change the language in three aspects:

1. We will allow functions to have either a single argument or two arguments.
2. We will have the basic arithmetic operations changed to functions (rather than the special forms they currently are). Specifically, expressions like **{+ 4 {* 3 2} }** will no longer be valid in our language. Instead, we will now be allowing only expressions such as **{call + 4 {call * 3 2}}**.
3. We will make the global environment – i.e., the one that **run** applies **eval** with – non-empty. Indeed, the primitive arithmetic (binary) functions will be contained in the global environment.

Use the environment based interpreter we have seen in class as the basis for your work (see here).

Here are some tests that should work after you are done:

```
;; tests
  (test (run "{call + 4 5}") => 9)
  (test (run "{with {add3 {fun {x} {call + x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {x 3}
              {with {f {fun {y} {call + x y}}}
                {with {x 5}
                  {call f 4}}}}")
      => 7)
  (test (run "{call {fun {x y} {call + x { call - y 1}}} 4 2}")   => 5)
   (test (run "{with {first {fun {x y} x}}
              {with {second {fun {x y} y}}
                {call first {call second 2 123} 124}}}")
      => 123)
  (test (run "{+ 4 5}") =error> "parse-sexpr: bad syntax in")
  (test (run "{* 4 5}") =error> "parse-sexpr: bad syntax in")
  (test (run "{with {add3 {fun {x} {call + x 3}}}
              {call add3 1 2}}")
      =error>  "expected a single argument, got two in: ")
  (test (run "{with {add3 {fun {x stam} {call + x 3}}}
              {call add3 1}}")
      =error>  "expected two arguments, got one in: ")
```

2. The new FLANG BNF language

The new BNF and Parser should support the above mentioned syntax. The BNF is given to you. Use it to understand the requirements.

```
#|  The grammar:
```

```
<FLANG> ::= <num>
      | { with { <id> <FLANG> } <FLANG> }
      | <id>
      | { fun { <id> } <FLANG> }    ;;a function may have a
                                      single formal parameter
      | { fun { <id> <id> } <FLANG> } ;; or two formal parameters
      | { call <FLANG> <FLANG> }      ;;a function has either a
                                          single actual parameter
      | { call <FLANG> <FLANG> <FLANG> } ;; or two actual parameters
|#
```

3. Extending the Parser

Use the above test examples and the original interpreter code to accommodate the new syntactic requirements. Fill in the missing parts…

```
(define-type FLANG
   [Num  Number]
   [Id   Symbol]
   [Add FLANG FLANG] ; Never created by user
   [Sub FLANG FLANG] ; Never created by user
   [Mul FLANG FLANG] ; Never created by user
   [Div FLANG FLANG] ; Never created by user
   [With Symbol FLANG FLANG]
   […
    ... <-- fill-in -->
    …] ;; There should be several variants here (hint - you could use
two new Fun-like and two new Call-like constructors)
```

```
(: parse-sexpr : Sexpr -> FLANG)
 ;; to convert s-expressions into FLANGs
 (define (parse-sexpr sexpr)
   (match sexpr
 …
     [(cons 'fun more)
      (match sexpr
        <-- fill-in -->
        <-- fill-in -->
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
     [<-- fill-in -->]
     [<-- fill-in -->]
     [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

4. Extending eval

In order to support your code, you will need to slightly change the `VAL` structure. The definitions of `ENV`, `lookup`, and `arith-op` should remain unchanged.

```
(define-type VAL
        […
     ... <-- fill-in -->
     …]) ;; There should be several variants here (hint - you could use
two new FunV-like constructors)
```

Use the above defined procedures and the formal rules below to complete the code below for the `eval` procedure. Consult the provided tests at the introduction part of the assignment.

```
   eval:  Evaluation rules:
    eval(N,env)                 = N
    eval(x,env)                 = lookup(x,env)
    eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
    eval({fun {x1} E},env)      = <{fun {x1} E}, env>
    eval({fun {x1 x2} E},env)   = <{fun {x1 x2} E}, env>
    eval({call E-op E1},env1)
           = eval(Ef,extend(x1,eval(E1,env),envf))
                        if eval(E-op,env) = <{fun {x} Ef}, envf>
             = error!   otherwise
    eval({call E-op E1 E2},env1)
         = eval(Ef,extend(x2,eval(E2,env),extend(x1,eval(E1,env),envf))
                       if eval(E-op,env) = <{fun {x1 x2} Ef}, envf>
         = error!       Otherwise

  (: eval : FLANG ENV -> VAL)
  ;; evaluates FLANG expressions by reducing them to values
  (define (eval expr env)
    (cases expr
      [(Num n)  (NumV n)]
      [(With bound-id named-expr bound-body)
       (eval bound-body
             (Extend bound-id (eval named-expr env) env))]
      [(Id name) (lookup name env)]
      [(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))]
     <-- fill-in --> ;; There should be several cases here
      [  <-- fill-in -->
            (error 'eval "expected two arguments, got one in: ~s"
                                 fval)]
           [else (error 'eval "`call' expects a function, got: ~s"
                                 fval)]))]
      [  <-- fill-in -->
             (error 'eval "expected a single argument, got two in: ~s"
                                 fval)]
           [else (error 'eval "`call' expects a function, got: ~s"
                                 fval)]))]))
```

5. Constructing the Global-Environment

To allow all programs to know and use basic arithmetic operations, we will incorporate them (as two-input functions) into the environment that is provided to every run.

```
(: createGlobalEnv : -> ENV)
  (define (createGlobalEnv)
    (Extend '+   <-- fill-in -->   ))
```

Hint: Consider using the known Add, Sub, Mul, and Div variants that are not yet ever really used (being created) in your code.

6. Extending the run procedure

Finally, we will allow the interface procedure to use the global environment. Complete the code for the `run` procedure.

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