Homework #4: Interpreter for The Functional Register Language and More 2019

Out: Friday, May 24, 2019, Due: Thursday, June 6, 2019, 23:59

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 substitution model.
2. Counting free instances in the substitution model FLANG interpreter.
3. A touch on the difference between the substitution model and the substitution-cache model (static vs. dynamic scoping).

An interpreter that supports **`the substitution model`** as well as **`the substitution-cache model (eval and run are renamed evalSC and runSC, respectively)`** 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. Adding function expressions to the ROL language

In the previous assignment, you have implemented the language "ROL", analogously to the WAE language we have seen in class. In this part, you need to extend your language to also support **fun** and **call** similarly to the FLANG language we have seen in class. Make sure to implement your interpreter in the **substitution** model (use the interpreter we have seen in class as a basis for your code – see at the end of this file).

- Extend your BNF and Parser to Support this syntax.
- Extend the eval procedure to support first class functions.

Remark: Since you are required to submit a single .rkt file for all solutions, you will need to change the interface here. Please add an **FROL** suffix to all appropriate functions. Specifically, **run** should be renamed **runFROL**.

Following are some tests that should go through (all tests given in the previous assignment should also work).

```
(test (runFROL "{ 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 (runFROL "{ 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 (runFROL "{ 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. Counting Free Instances of a Given Symbol

Here, you are to use the FLANG interpreter in the substitution model. Write a function **countFreeSingle** that takes an abstract syntax tree for the FLANG language and a symbol, and returns the number of free instances of the given symbol appear in the expression (tree).

To consider some tests that should go through, define the following wrapper function:

```
(: CFSingle : String Symbol -> Natural)
(define (CFSingle expr name)
  (countFreeSingle (parse expr) name))
```

Then, the following should go through,

```
(test (CFSingle "{+ r r}" 'r) => 2)
(test (CFSingle "{fun {r} {+ r e}}" 'e) => 1)
(test (CFSingle "{fun {r} {+ r e}}" 'r) => 0)
(test (CFSingle "{call {fun {r} {+ r e}}
                       {with {e {+ e r}}
                         {fun {x} {+ e r}}}}"
                'r) => 2)

(test (CFSingle "{call {fun {r} {+ r e}}
                       {with {e {+ e r}}
                         {fun {x} {+ e r}}}}"
```

```
'e) => 2)
```

a. An interesting test case

Consider the following FLANG code:

```
"{with {foo {fun {y} {+ x y}}}
       {with {x 4}
         {call foo 3}}}"
```

What happens when you run the following code?

```
(CFSingle "{with {foo {fun {y} {+ x y}}}
                 {with {x 4}
                   {call foo 3}}}" 'x)
```

What happens when you run the FLANG interpreter (substitution model) on this code?

```
(run "{with {foo {fun {y} {+ x y}}}
            {with {x 4}
              {call foo 3}}}")
```

Explain what goes on and why it happens.

3. Static versus Dynamic Scoping (recursion)

Here, you are to use the FLANG interpreter presented below (copy it into your solution file). This interpreter works for either the substitution model (using **run**) or the substitution-cache model (using **runSC**). You need to write a FLANG language code – your code should be a string, and should be named **loop**. Fill in the missing parts.

```
(define loop "{with {<your name> <<- fill in ->>}}")
```

The code **loop** should be such that in the substitution model a "no binding" error message is given, and in the substitution-cache model, eval should go into an infinite loop.

Specifically, for the extended interpreter below, the following tests should go through:

```
(test (runSC loop) =error> "exceeded 500 times") ;; subst-cache model
(test (run loop) =error> "free identifier: f") ;; substitution model
```

Note that the 500 barrier was added to evalSC to prevent real infinite loops.

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


  ;; The Flang interpreter - supporting both the substitution model and
the substitution-cache model

  #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> }
|#

  (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)))
```

;;;;;; the evaluation part for the substitution model

```
(: subst : FLANG Symbol FLANG -> FLANG)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
```

```
      [(Id name) (error 'eval "free identifier: ~s" name)]
      [(Fun bound-id bound-body) expr]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr)])
         (cases fval
           [(Fun bound-id bound-body)
            (eval (subst bound-body
                         bound-id
                         (eval arg-expr)))]
           [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))])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

;;;;;; The evaluation part for the substitution cache model

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
```

```
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

 (: counterx : Natural)
 (define counterx 0)
 ;;;above eval

(: evalSC : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (evalSC expr sc)
  (set! counterx (add1 counterx))
  (if (> counterx 500)
      (error 'eval "exceeded 500 times")
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (evalSC l sc) (evalSC r sc))]
    [(Sub l r) (arith-op - (evalSC l sc) (evalSC r sc))]
    [(Mul l r) (arith-op * (evalSC l sc) (evalSC r sc))]
    [(Div l r) (arith-op / (evalSC l sc) (evalSC r sc))]
    [(With bound-id named-expr bound-body)
     (evalSC bound-body
          (extend bound-id (evalSC named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (evalSC fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (evalSC bound-body
               (extend bound-id (evalSC arg-expr sc) sc))]
         [else (error 'evalSC "`call' expects a function, got: ~s"
                      fval)]))])))

(: runSC : String -> Number)
;; evaluate a FLANG program contained in a string
(define (runSC str)
  (let ([result (evalSC (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'runSC
                   "evaluation returned a non-number: ~s" result)])))
```