CS 496: Homework Assignment 4
Due: 24 March, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually (unless indicated otherwise by the instructor): each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

The aim of this assignment is to extend the interpreter for EXPLICIT-REFS in two different ways.

1. The first extension consists in allowing for procedures to declare multiple parameters. Consequently, the language should also allow a call to a procedure to receive multiple arguments.
2. The second extension consists in adding a `for` construct.

The resulting language will be called EXPLICIT-REFS-PLUS. In order to implement it, we must modify the following files from EXPLICIT-REFS:

- `lang.scm`: the language grammar (Extension 1 and 2)
- `data-structures.scm`: Closures and the definition of environments (Extension 1)
- `environments.scm`: the initial environment and the `apply-env` operation (Extension 1)
- `interp.scm`: the interpreter (Extension 1 and 2)

Your starting point should be the code for EXPLICIT-REFS which is available here `https://github.com/mwand/eopl3/tree/master/chapter4/explicit-refs`

3 Extension 1: Functions with Multiple Parameters

Detailed indications on each of the modifications will be provided below. Before doing so however, we give an example of a program written in EXPLICIT-REFS-PLUS:

```
let a = newref(3)
in let f = proc(x,y) setref(x,-(deref(x),y))
   in begin
        (f a 2);
        deref(a)
      end
```

Here `f` is defined to be a function that takes two parameters `x` and `y` which are separated by commas.

This extension to EXPLICIT-REFS is not strictly necessary since we can already write the following code which although equivalent in its behavior, is less clear.

```
let a = 3
in let f = proc(x) proc(y) setref(x,-(deref(x),y))
   in begin
        ((f a) 2);
        deref(a)
      end
```

3.1 Modifying the Language Grammar

Start by implementing the following modification. In the file `lang.scm` of EXPLICIT-REFS, replace the specification for "proc"

```
(expression
      ("proc" "(" identifier ")" expression)
      proc-exp)
```

with the following code:

```
(expression
      ("proc" "(" (separated-list identifier ",") ")" expression)
      proc-exp)
```

Likewise for procedure calls. In the file `lang.scm` of EXPLICIT-REFS, replace the specification of procedure calls:

```
(expression
    ("(" expression expression ")")
    call-exp)
```

with the following code:

```
(expression
    ("(" expression (arbno expression) ")")
    call-exp)
```

You can try out the resulting parser by running *just* the file `lang.scm`. Here are some examples of the abstract syntax produced by the parser:

```
> (scan&parse "proc (x,y,z) x")
(a-program (proc-exp '(x y z) (var-exp 'x)))
> (scan&parse "proc () x")
(a-program (proc-exp '() (var-exp 'x)))
> (scan&parse "(x 1 2 3 4)")
(a-program (call-exp (var-exp 'x) (list (const-exp 1) (const-exp 2)
    (const-exp 3) (const-exp 4))))
> (scan&parse "(x)")
(a-program (call-exp (var-exp 'x) '()))
```

The definition of the abstract syntax for EXPLICIT-REFS-PLUS], which is generated automatically for you, is now:

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
  (const-exp (num number?))
  (var-exp (var identifier?))
  ...
  (proc-exp
        (params (listof symbol?))
        (body expression?))
  (call-exp
        (rator expression?)
        (rands (listof expression?)))
)
```

3.2 Modifying the Closures and Environments

You need to modify the definitions of `proc` and `environment` in the file `data-structures.scm` so that environments are now a sequence of associations between *lists* of symbols and *lists* of expressed values. Recall that you can use `listof <type>` to indicate that an argument have type list of type `<type>`. For example an argument of type `listof symbol?` is a list of symbols.

```
  (define-datatype proc proc?
    (procedure
      (bvar symbol?)    ;; change here
      (body expression?)
      (env environment?)))

  (define-datatype environment environment?
    (empty-env)
```

```
    (extend-env
      (bvar symbol?)    ;; change here
      (bval expval?)    ;; change here
      (saved-env environment?))
    (extend-env-rec*
      (proc-names (list-of symbol?))
      (b-vars (list-of symbol?))
      (proc-bodies (list-of expression?))
      (saved-env environment?)))
```

Since you changed the structure of environments you must update also the `init-env` and `apply-env` (just the case for `extend-env`, the others remain as they are) in the file `environments.scm`. For the `apply-env` you might need to use the `location` function defined at the end of this file. These changes are left to you.

3.3 Modifying the Interpreter

You may now address the final step, namely updating the interpreter itself. Update the following two cases of the `value-of` function in `interp.scm` so that it is capable of handling the language extension:

```
(define value-of
  (lambda (exp env)
    (cases expression exp
    ...
    (let-exp (var body)
      (write "update me!"))

    (proc-exp (var body)
      (write "update me!"))

    (call-exp (rator rand)
      (write "update me!")))))
```

The changes to the original implementation of these cases in EXPLICIT-REFS is minimal. So I suggest starting off from them.

Note that since the `call-exp` case uses the `apply-procedure` function, you shall also have to update that one (it is located in the same `interp.scm` file).

4 Extension 2: For Loops

Here is an example that uses the `for` loop extension.

```
let x = newref(0)
in begin
     for i=1 to 10 (
      setref(x,-(deref(x),-1))
     );
     deref(x)
```

```
   end
```

The result of running this program should be `num-val 10`.

Here are some variants:

```
let x = newref(0)
in begin
     for i=1 to 1 (
      setref(x,-(deref(x),-1))
     );
     deref(x)
   end
```

This returns `num-val 1` because it performs one iteration. However, the following program:

```
let x = newref(0)
in begin
     for i=10 to 1 (
      setref(x,-(deref(x),-1))
     );
     deref(x)
   end
```

should return `num-val 0` since it performs no iterations.

Some more examples:

```
let l = 1
in
let u = 10
in
let x = newref(0)
in begin
     for i=l to u (
      setref(x,-(deref(x),-1))
     );
     deref(x)
   end
```

should return `numval-10` and

```
let l = 1
in
let u = 10
in
let x = newref(0)
in begin
     for i=l to -(u,1) (
      setref(x,-(deref(x),-1))
     );
     deref(x)
   end
```

should return `num-val 9`.

4.1 Modifying the Language Grammar

Add the following to `lang.scm`:

```
(expression
        ("for" identifier "=" expression "to" expression "("
    expression ")")
        for-exp)
```

4.2 Modifying the Interpreter

Add a new case in the definition of `value-of` to handle the for loop. Your solution **must** use the `for-each` construct available in Racket for implementing the behavior of this extension.

5 Submission instructions

Submit a file named `HW4_<SURNAME>.zip` through Canvas. It should include all the files required to run the interpreter. Please include your name and pledge in the `top.scm` source file.