Problem Set 09

OUT: Monday 3/16/2015, 9pm EST
DUE: Monday 3/23/2015, 9pm EST

Last updated: Wed, 18 Mar 2015 19:38:36 -0400

Preliminaries

- You must use the `Intermediate Student with lambda` language to complete this assignment. Select it via the `Choose Language...` menu located at the bottom-left of the DrRacket window.
- Put all your solution files in a directory named `set09` in your repository.
- Download extras.rkt to this directory (right-click and choose "Save As"; don't copy and paste) and commit it as well.
- Use `begin-for-test` and `rackunit` to define your examples and tests.
- Don't forget to tell us how many hours you spent working on the assignment. This should be a global variable called `TIME-ON-TASK` in each file. For example:

```
(define TIME-ON-TASK 10.5) ; hours
```

- So each solution file must have at least the following at the top:

```
(require "extras.rkt")
(require rackunit)
(define TIME-ON-TASK <number-of-hours-you-spent>)
```

- After you've submitted your solution, use a web browser to go to https://github.ccs.neu.edu/ and check that your repository contains the following files:
  - `set09/eval.rkt`
  - `set09/extras.rkt`
- **Git Commit Requirement**: For this assignment, you must have at least three well-labeled git commits (including the final commit). A well-labeled git commit accurately and succinctly describes the changes since the previous commit. Something like `"commit2"`, or `"home work 3"` is not an acceptable git commit label. Failure to meet this requirement will result in loss of points.
- Be sure to state appropriate invariants and follow the proper style if you use any accumulators.
- Be sure to state a termination argument for any functions that follow the generative recursion strategy. In other words, generative recursive functions should either explain why the function always terminates, or indicate which inputs result in non-termination.
- **ADDITIONAL REQUIREMENT**: Submit a call graph of the functions in your program.

This helps you (and us) better understand the structure of your program. You may use any tool to draw your graph. Here is an example for a program similar to Problem Set 07.

1 Interpreter

1.1 Additional Preliminaries

Save your solutions for this problem to a file named `eval.rkt`.

Run the following expression (you must have required extras.rkt) to check that your file is properly named and is in the proper directory:

```
(check-location "09" "eval.rkt")
```

Add these additional provides at the top of your file (below the requires), so that we can test your solution:

```
(provide eval)

(provide lambda?)

(provide errstr?)

(provide subst)

(provide expr->expr/no-var)

(provide expr=?)
```

1.2 Problem Description

Although you are having a great time programming with Racket, you've decided to create your own programming language, called PDPLang . Your task for this assignment is to implement an interpreter for this language.

Note: Though this assignment is self-contained, you may find it helpful by reviewing Chapter 24 of the textbook.

Specifically, design the following function:

```
; eval : Program -> ListOf<Result>

; Evaluates a PDPLang program to a list of Results.
; Specifically, evaluates the Exprs in p, in the context of the given Defs.
; WHERE: A function may be called before it is defined.
; WHERE: The produced results are in the same relative order as their
; originating Exprs in p.
(define (eval p) ...)
```

To help us test, please also implement and provide the following predicates:

```
; lambda? : Expr -> Boolean
; Returns true if e is a PDPLang lambda expression.
(define (lambda? e) ...)
; errstr? : Expr -> Boolean
; Returns true if e is a PDPLang ErrString expression.
(define (errstr? e) ...)
```

Assume that PDPLang programs (i.e., information) have already been parsed to the following *data* representations.

```
; A UniqueListOf<X> is a ListOf<X>
; WHERE: none of the elements of the list are equal? to each other

; A 2ListOf<X> is a (cons X (cons X ListOf<X>))
; Represents a list of 2 or more elements.

; A Program is a:
; - empty
; - (cons Expr Program)
; - (cons Def Program)
; Represents a PDPLang program consisting of function defs and expressions.
; WHERE: no two Defs have the same name

; An Expr is one of:
; - Number
; - Boolean
; - Var
; - ErrString
; - Lambda
; - (make-arith ArithOp 2ListOf<Expr>) ; an arithmetic expression
; - (make-bool BoolOp 2ListOf<Expr>)   ; a boolean expression
; - (make-cmp CmpOp 2ListOf<Expr>)     ; a comparison expression
; - (make-if-exp Expr Expr Expr) ; an if conditional
; - (make-call Expr ListOf<Expr>) ; a function call
; Represents a PDPLang expression.
(define-struct arith (op args))
(define-struct bool (op args))
(define-struct cmp (op args))
(define-struct if-exp (test branch1 branch2))
(define-struct call (fn args))

; A Var is a Symbol, representing PDPLang variable.

; An ErrString is a String, representing a PDPLang error message.

; A Lambda is a (make-lam UniqueListOf<Param> Expr)
; Represents a lambda expression in PDPLang
(define-struct lam (params body))

; A Param is a Var, representing a function parameter.

; An ArithOp is one of:
; - '+
; - '-
; - '*
; - '/
; Represents an arithmetic operation in PDPLang
; A BoolOp is one of:
; - 'and
; - 'or
; Represents a boolean operation in PDPLang

; A CmpOp is one of:
; - '=
; - '<
; - '>

; Represents a comparison operation in PDPLang

; A Def is a (make-def FnName UniqueListOf<Param> Expr)
; Represents the definition of a function with the specified parameters.
(define-struct def (name params body))

; A FnName is a Var, representing a function name.

; A Result is a:
; - Number
; - Boolean
; - ErrString
; - Lambda
```

Complete the Data Design by adding templates and data examples.

Here’s how to evaluate a PDPLang Program.

- A PDPLang Program evaluates to a list of Results, where the list of Results are result of evaluating the Exprs in the program.
- Exprs in a PDPLang Program are evaluated assuming that all the Defs in the program are already defined.

Here’s how to evaluate a PDPLang Expr.

- An Expr that is already a Result evaluates to itself.
- A `Var`, if it represents a defined `Def`, should evaluate to an equivalent lambda Expr. Otherwise, a `Var` should evaluate to an appropriate `ErrStr`.

  ```
  (define F (make-def 'f '(x) 'x))
  (check-pred lambda? (first (eval (list F 'f))) "result is lambda")
  (check-pred errstr? (first (eval (list F 'g))) "err: undefined var")
  (check-pred errstr? (first (eval (list 'x))) "err: undefined var")
  ```

- Arithmetic and boolean operations evaluate to the obvious Result.
- An if expression:
  - first evaluates its test expression.
  - The result of evaluating an if expression is the result of evaluating the first branch if the test expression evaluates to `true`.
  - The result of evaluating an if expression is the result of evaluating the second branch if the test expression evaluates to `false`.

  - Otherwise, the result of evaluating the if expression should be an appropriate error message.

- Evaluate a function call by:

1. evaluating its function subexpression; if the result is not a `Lambda`, the result of the function call is an appropriate `ErrString`;

2. evaluating its arguments;

3. replacing each parameter reference in the function body with its corresponding argument and evaluating the resulting expressions; a function call evaluates to an appropriate `ErrString` if the number of arguments does not match the number of function parameters.

4. To help you with function call evaluation, implement, provide and use the following function:

```
; subst : Result Var Expr -> Expr
; Replaces references to x in e with r.
; Does not replace x with r if x occurs in the body of a lambda

; that shadows x.
; WHERE: r has no unbound variables
(begin-for-test
  (check-equal? (subst 4 'x 'x) 4 "x matches")
  (check-equal? (subst 4 'y 'x) 'x "y doesnt match")
  (check-equal?
   (subst 4 'x (make-arith '+ '(x 5)))
   (make-arith '+ '(4 5))
   "subst in arith")
  (check-equal?
   (subst 4 'x (make-lam '(y) (make-arith '+ '(x y))))
   (make-lam '(y) (make-arith '+ '(4 y)))
   "subst in lambda")
  (check-equal?
   (subst 4 'x (make-lam '(x) (make-arith '+ '(x 5))))
   (make-lam '(x) (make-arith '+ '(x 5)))
   "dont subst shadowed vars in lambdas"))
(define (subst r x e) ...)
```

- Any expressions whose subexpressions evaluate to the wrong kind of Result should evaluate to an `ErrString`.

```
(check-true
 (ormap
  errstr?
  (eval (list (make-arith '+ (list 2 true))
              (make-bool 'and (list 2 true))
              (make-if-exp 3 1 2))))
 "wrong kind of args")
```

UPDATE 2015-03-18: Finally, implement `expr->expr/no-var` and `expr=?` (this part of problem set is based on exercise 451 of the textbook).

```
; An ExprNoVar is one of:
; - Number
; - Boolean
; - StaticDist

; - ErrString
; - LamNoVar
; - (make-arith ArithOp 2ListOf<ExprNoVar>) ; an arithmetic expression
; - (make-bool BoolOp 2ListOf<ExprNoVar>)   ; a boolean expression
; - (make-cmp CmpOp 2ListOf<ExprNoVar>)     ; a comparison expression
; - (make-if-exp ExprNoVar ExprNoVar ExprNoVar) ; an if conditional
; - (make-call ExprNoVar ListOf<ExprNoVar>) ; a function call
; Represents an Expr without explicit variables.

; A StaticDist is a (list Depth Index)
; Represents a variable reference
; where depth is number of additional lambdas between this var ref and the
; lambda for which this variable is a parameter,
; and index is the (0-based) position of this variable in that lambda's
; parameter list.

; A Depth is a Natural
; An Index is a Natural

; A LamNoVar is a (make-lam/no-var ExprNoVar)
(define-struct lam/no-var (body))

; expr->expr/no-var : Expr -> ExprNoVar
; Replaces Var in e with StaticDist.
; WHERE: there are no unbound variables in e.
(begin-for-test
  (check-equal?
   (expr->expr/no-var (make-lam '(x) 'x))
   (make-lam/no-var '(0 0))
   "basic lambda")
  (check-equal?
   (expr->expr/no-var (make-lam '(x y) (make-lam '(z) (make-call 'x '(y z)))))
   (make-lam/no-var (make-lam/no-var (make-call '(1 0) '((1 1) (0 0)))))
   "nested lambdas"))
(define (expr->expr/no-var e) ...)

; expr=? : Expr Expr -> Boolean
; Returns true if e1 and e2 are structurally equivalent, up to some

; renaming of variable names.
(begin-for-test
  (check
   expr=?
   (make-lam '(x) 'x)
   (make-lam '(y) 'y)
   "equivalent basic lambdas")
  (check-false
   (expr=?
    (make-lam '(x y) (make-call 'x '(y)))
    (make-lam '(y x) (make-call 'x '(y))))
   "not equivalent")
  (check
   alpha=?
   (make-lam '(y) (make-lam '(x) (make-call 'y '(x))))
   (make-lam '(x) (make-lam '(y) (make-call 'x '(y))))
   "equivalent nested-lambdas"))
(define (expr=? e1 e2) ...)
```