Homework 5. Scheme code difference analyzer

The problem

Your employer Litigious Data Analysts Inc. (LDA) is suing Software Verification Modules Inc. (SVM), claiming that SVM stole large bodies of LDA's code and incorporated it into their data mining product, while tring to hide the fact that the code was stolen by renaming identifiers and by replacing `lambda` with λ and vice versa, and also making other minor changes. As part of the legal discovery process, LDA has obtained copies of SVM's data miner and wants to compare it to LDA's to find evidence of unauthorized copying. About 5% of both data miners are written in Scheme. Your team has been assigned the job of comparing the Scheme parts.

Your boss suggested that you prototype a procedure `expr-compare` that compares two Scheme expressions *x* and *y*, and produces a difference summary of where the two expressions are the same and where they differ. Your boss wants the difference summary to be easily checkable, in case there is a bug in `expr-compare` itself. So you decide to have the difference summary also be a Scheme expression which, if executed in an environment where the Scheme variable `%` is true, has the same behavior as *x*, and otherwise has the same behavior as *y*. You prefer to have a shorter summary expression, so you decide that the summary should use λ in places where one input expression used a `lambda` expression and the other used a λ expression (however, the summary should use `lambda` in places where both input expressions used `lambda`). You also want the summary expression to use the same identifiers as the two input expressions where they agree, and that if *x* declares the bound variable *X* in the same place where *y* declares the bound variable *Y*, the summary expression should declare a bound variable *X*`!`*Y* and use it consistently thereafter wherever the input expressions use *X* and *Y* respectively. (A *bound variable* is one that is declared in an expression by a binding construct such as `lambda`.)

To keep things simple your prototype need not handle arbitrary Racket expressions; it can be limited to the subset of expressions that consists of literal constants, identifiers, function calls, the special form `(quote` *s-exp*`)`, the special form `(lambda` *formals expr*`)` where *formals* is a list of identifiers, the equivalent special form `(λ` *formals body*`)`, and the special-form conditional `(if` *test-expr expr expr*`)`. To avoid confusion the input Scheme expressions cannot contain any symbols which contain the `%` or `!` characters. Your prototype need not check that its inputs are valid; it can have undefined behavior if given inputs outside the specified subset. For example, your prototype need not worry about inputs like `(if)` or `(lambda . 0)`, expressions that Racket rejects with the diagnostic “bad syntax”; conversely, your prototype should handle valid Scheme expressions like `((lambda (if lambda quote) (* quote if lambda 2)) 3 5 7)` where the first `lambda` is a keyword and the other symbols are ordinary identifiers.

Assignment

First, write a Scheme procedure `(expr-compare` *x y*`)` that implements the specification described above. Your implementation must be free of side effects; for example you cannot use the `set!` special form. Returned values should share storage with arguments when possible; they should not copy their arguments.

The output expression should use `if` expressions and identifiers containing `!` to represent differences whenever the two input expressions disagree, attempting to minimize the size of the subexpressions under the generated `if`s. As a special case, it should use `%` to represent a subexpression that is `#t` in LDA's version and `#f` in SVM's version, and should use `(not %)` to represent the reverse situation. Here are some examples and what they should evaluate to.

```
(expr-compare 12 12)  ⇒  12
(expr-compare 12 20)  ⇒  (if % 12 20)
(expr-compare #t #t)  ⇒  #t
(expr-compare #f #f)  ⇒  #f
```

```
(expr-compare #t #f)  ⇒  %
(expr-compare #f #t)  ⇒  (not %)
(expr-compare 'a '(cons a b))  ⇒  (if % a (cons a b))
(expr-compare '(cons a b) '(cons a b))  ⇒  (cons a b)
(expr-compare '(cons a lambda) '(cons a λ))  ⇒  (cons a (if % lambda λ))
(expr-compare '(cons (cons a b) (cons b c))
              '(cons (cons a c) (cons a c)))
  ⇒ (cons (cons a (if % b c)) (cons (if % b a) c))
(expr-compare '(cons a b) '(list a b))  ⇒  ((if % cons list) a b)
(expr-compare '(list) '(list a))  ⇒  (if % (list) (list a))
(expr-compare ''(a b) ''(a c))  ⇒  (if % '(a b) '(a c))
(expr-compare '(quote (a b)) '(quote (a c)))  ⇒  (if % '(a b) '(a c))
(expr-compare '(quoth (a b)) '(quoth (a c)))  ⇒  (quoth (a (if % b c)))
(expr-compare '(if x y z) '(if x z z))  ⇒  (if x (if % y z) z)
(expr-compare '(if x y z) '(g x y z))
  ⇒ (if % (if x y z) (g x y z))
(expr-compare '((lambda (a) (f a)) 1) '((lambda (a) (g a)) 2))
  ⇒ ((lambda (a) ((if % f g) a)) (if % 1 2))
(expr-compare '((lambda (a) (f a)) 1) '((λ (a) (g a)) 2))
  ⇒ ((λ (a) ((if % f g) a)) (if % 1 2))
(expr-compare '((lambda (a) a) c) '((lambda (b) b) d))
  ⇒ ((lambda (a!b) a!b) (if % c d))
(expr-compare ''((λ (a) a) c) ''((lambda (b) b) d))
  ⇒ (if % '((λ (a) a) c) '((lambda (b) b) d))
(expr-compare '(+ #f ((λ (a b) (f a b)) 1 2))
              '(+ #t ((lambda (a c) (f a c)) 1 2)))
  ⇒ (+
     (not %)
     ((λ (a b!c) (f a b!c)) 1 2))
(expr-compare '((λ (a b) (f a b)) 1 2)
              '((λ (a b) (f b a)) 1 2))
  ⇒ ((λ (a b) (f (if % a b) (if % b a))) 1 2)
(expr-compare '((λ (a b) (f a b)) 1 2)
              '((λ (a c) (f c a)) 1 2))
  ⇒ ((λ (a b!c) (f (if % a b!c) (if % b!c a)))
     1 2)
(expr-compare '((lambda (lambda) (+ lambda if (f lambda))) 3)
              '((lambda (if) (+ if if (f λ))) 3))
  ⇒ ((lambda (lambda!if) (+ lambda!if (if % if lambda!if) (f (if % lambda!if λ)))) 3)
(expr-compare '((lambda (a) (eq? a ((λ (a b) ((λ (a b) (a b)) b a))
                                    a (lambda (a) a))))
                (lambda (b a) (b a)))
              '((λ (a) (eqv? a ((lambda (b a) ((lambda (a b) (a b)) b a))
                                a (λ (b) a))))
                (lambda (a b) (a b))))
  ⇒ ((λ (a)
       ((if % eq? eqv?)
        a
        ((λ (a!b b!a) ((λ (a b) (a b)) (if % b!a a!b) (if % a!b b!a)))
         a (λ (a!b) (if % a!b a)))))
     (lambda (b!a a!b) (b!a a!b)))
```

(When testing your code, please note that Racket read–eval–print loop quotes its results unless they are self-quoting, so that, for example, although `12` prints as itself, `(if % 12 20)` prints as `'(if % 12 20)`.)

Second, write a Scheme procedure `(test-expr-compare` *x y*`)` that tests your implementation of `expr-compare` by using `eval` to evaluate the expression *x*, and to evaluate the expression returned by `(expr-compare` *x y*`)` in the same context except with `%` bound to `#t`, and which checks that the two expressions yield the same value. Similarly, it should check that *y* evaluates to the same value that the output of `expr-compare` evaluates to with `%` bound to `#f`. The `test-expr-compare` function should return a true value if both tests succeed, and `#f` otherwise.

Third, define two Scheme variables `test-expr-x` and `test-expr-y` that contain data that can be interpreted as Scheme expressions that test `expr-compare` well. Your definitions should look like this:

```
(define test-expr-x '(+ 3 ((lambda (a b) (list a b)) 1 2)))
(define test-expr-y '(+ 2 ((lambda (a c) (list a c)) 1 2)))
```

except that your definitions should attempt to exercise all the specification in order to provide a single test case for this complete assignment.

Submit

Submit a file `expr-compare.ss` containing the definitions of `expr-compare`, `test-expr-compare`, `test-expr-x`, and `test-expr-y` along with any other auxiliary definitions needed. Make sure that your definitions work with `racket`, the Scheme implementation installed on SEASnet.

$Id: hw5.html,v 1.67 2020/05/20 21:41:08 eggert Exp $