```
;; recon-test.scm
;;
;; Test the type reconstructor for Scheme/R
;;
;; You must load Scheme+ and recon.scm before running this code.
;;
;;
;; To run the test suite, execute
;;
;; (run-tests)
;;
;;

;; Modified from FL test suite by BJR

(define test-suite '())
(define test-counter 0)
(define test-recon-failed (list 'recon-failed))
(define halt-on-error #t)

(define-datatype test-case
(test-case int sexp type-sexp))

(define (atest-case-n atest-case)
(match atest-case
((test-case n _ _) n)))

(define (atest-case-exp atest-case)
(match atest-case
((test-case _ exp _) exp)))

(define (atest-case-type-sexp atest-case)
(match atest-case
((test-case _ _ tsexp) tsexp)))

(define (add-test! sexp result)
(set! test-counter (1+ test-counter))
(set! test-suite (cons (test-case test-counter sexp result)
                           test-suite))
unspecific)

(define (run-tests)
(let ((passed #t))
(for-each
(lambda (atest-case)
(newline)
```

```
(for-each display
        (list "Running test " (atest-case-n atest-case) " ..."))
(run-test atest-case
        (lambda (passed? val)
        (if passed?
        (display " OK!")
        (begin (set! passed #f)
            (test-failed atest-case val))))))
(reverse test-suite))
(newline)
(if passed
(for-each display (list "Test Suite passed -- "
                (length test-suite)
                " test cases.")))
unspecific))


(define (test-failed atest-case val)
(match atest-case
((test-case n sexp result)
(let ((msg (apply error-string
        (string-append
            "\nTest Case " (number->string n) " Failed:")
        ""
        (list sexp result val))))
(if halt-on-error
    (error msg)
    (display msg))))
))

(define (run-test test return)
;; Returns two values:
;; * A boolean that indicates whether the actual value matched the expected one.
;; * The actual value of the test.
(if (eq? (atest-case-type-sexp test) test-recon-failed)
(let ((bool&val
    (call-with-current-continuation
    (lambda (k)
        (fluid-let ((standard-error-hook
            (lambda (condition)
                (k (cons #t test-recon-failed)))))
        (cons #f (check (atest-case-prog test))))))))
(return (car bool&val) (cdr bool&val)))
(let ((bool&val
    (call-with-current-continuation
    (lambda (k)
                    i.  (fluid-let ((standard-error-hook
                              1.  (lambda (condition)
                                        a.  (k (cons #f (with-output-to-string
                                                  i.  (lambda ()
                                                 ii.  (write-condition-report
                                                            1.  condition
                                                            2.  (current-output-port)))))))))
                   ii.  (let* ((ignore (reset-tvariable-counter!))
                              1.  (recon-type (reconstruct (parse (atest-case-exp test))
                                                            1.  standard-type-environment)))
                  iii.  (cons (compare-types
                              1.  recon-type
                              2.  (instantiate-schema
                              3.  (parse-schema (atest-case-type-sexp test))))
                              4.  recon-type)))))))
59. (return (car bool&val) (cdr bool&val)))))


60. (define (compare-types t1 t2)
61. (call-with-current-continuation
62. (lambda (k)
63. (fluid-let ((standard-error-hook
                    i.  (lambda (condition)
                   ii.  (k #f))))
64. (begin (unify! t1 t2)
        a.  #t)))))
```

```
65. (add-test! '(let ((g (lambda (x) x)))
        a.  (if (g #t) (g 1) (g 2)))
        b.  'int)

66. (add-test! '(lambda (g)
        a.  (if (g #t) (g 1) (g 2)))
        b.  test-recon-failed)

67. (add-test! '(lambda (f)
        a.  (let ((g f))
                i.  (if (g #t) (g 1) (g 2))))
        b.  test-recon-failed)

68. (add-test! '(lambda (f)
        a.  (let ((g (lambda (x) (f x))))
                i.  (if (g #t) (g 1) (g 2))))
        b.  test-recon-failed)


69. (add-test! '(letrec ((fact (lambda (n)
                        1.  (if (= n 0)
                                a.  1
                                b.  (* n (fact (- n 1)))))))
        b.  fact)
        c.  '(-> (int) int))

70. (add-test!
71. '(letrec ((map (lambda (p l)
                i.  (if (null? l)
               ii.  (null)
              iii.  (cons (p (car l))
                        1.  (map p (cdr l)))))))
72. map)
73. '(generic (?t-17 ?result-16)
74. (-> ((-> (?t-17) ?result-16) (list-of ?t-17)) (list-of ?result-16))))

75. (add-test! '(lambda (x y)
        a.  (letrec ((map (lambda (p l)
                        1.  (if (null? l)
                                a.  (null)
                                b.  (cons (p (car l))
                                        i.  (map p (cdr l)))))))
               ii.  (append x
                        1.  (map (lambda (y-elt)
                        a.  (if y-elt 1 0))
                    2.  y))))
        b.  '(-> ((list-of int) (list-of bool)) (list-of int)))

76. (add-test! '(letrec ((^ (lambda (p n)
                    1.  (if (= n 0)
                    2.  (lambda (x) x)
                    3.  (lambda (x)
                        a.  (p ((^ p (- n 1)) x)))))))
        b.  ^)
        c.  '(generic (?x-11) (-> ((-> (?x-11) ?x-11) int) (-> (?x-11) ?x-11))))


77. ; Functions defined by letrec can be used polymorphically in the body
78. ; of the letrec.

79. (add-test! '(letrec ((g (lambda (x) x)))
        a.  (if (g #t) (g 1) (g 2)))
        b.  'int)

80. ; ... but letrec definitions aren't polymorphic over themselves.

81. ;Should fail
82. (add-test! '(letrec ((g (lambda (x) x))
                i.  (h (lambda () (if (g #t) (g 1) (g 2)))))
        b.  (if (g #t) (g 1) (g 2)))
        c.  test-recon-failed)

83. ; Should fail
84. (add-test! '(letrec ((f (lambda (x) x))
                i.  (g (lambda () (f 1))))
        b.  (f #t))
        c.  test-recon-failed)

85. ; A number of potential LETREC bugs are found by the following simple test,
86. ; which should fail.
87. (add-test! '(letrec ((a (lambda () 3))
                i.  (b (lambda () (if (a) 1 2))))
        b.  (b))
        c.  test-recon-failed)

88. ; Self-application should fail ...
89. (add-test! '(lambda (f) (f f))
        a.  test-recon-failed)

90. ; ... unless we know what we're self-applying
```

```
91. ;
92. (add-test! '(let ((twice (lambda (f) (lambda (x) (f (f x))))))
        a.  (twice twice))
        b.  '(generic (?result-6)
        c.  (-> ((-> (?result-6) ?result-6)) (-> (?result-6) ?result-6))))

93. (add-test!
94. '(let ((g (lambda (x) x)))
95. (if ((g g) #t) ((g g) 1) ((g g) 2)))
96. 'int)

97. ; Infinite loops match any type
98. ;
99. (add-test! '(letrec ((loop (lambda () (loop))))
        a.  (lambda (x)
                i.  (if x 3 (loop))))
        b.  '(-> (bool) int))

100.        (add-test! '(letrec ((loop (lambda () (loop))))
        a.  (lambda (x)
                i.  (if x "three" (loop))))
        b.  '(-> (bool) string))

101.        ; Type clash: string vs. int
102.        (add-test! '(letrec ((loop (lambda (b) (if b 1 (loop b)))))
        a.  (lambda (x)
                i.  (if x "three" (loop x))))
        b.  test-recon-failed)


103.        ;;; Hairy examples with identity

104.        ; type clash (-> (bool) bool) (-> (int) bool)
105.        (add-test! '(lambda (x)
        a.  (let ((id (lambda (a) a)))
                i.  (let ((id2 (if #t
                                    a.  id
                                    b.  x)))
               ii.  (if (id2 #f) (id2 3) 4))))
        b.  test-recon-failed)

106.        (add-test! '((lambda (x)
                i.  (let ((id (lambda (a) a)))
               ii.  (if #t
              iii.  id
               iv.  x)))
        b.  (lambda (z) z))
        c.  '(generic (?z-2) (-> (?z-2) ?z-2)))


107.        (add-test! '(lambda (x)
        a.  (begin (set! x 3)
                i.  x))
        b.  '(-> (int) int))

108.        (add-test! '((lambda (+ *)
                i.  (primop + 1 2))
        b.  (lambda (x) x)
        c.  (lambda (x) (* x x)))
        d.  'int)
```