Лабораторная работа №2

Типы данных. Юнит-тестирование.

Цели работы

- На практике ознакомиться с системой типов языка Scheme.
- На практике ознакомиться с юнит-тестированием.
- Разработать свои средства отладки программ на языке Scheme.
- На практике ознакомиться со средствами метапрограммирования языка Scheme.

Вопросы для допуска к работе

1. Определение понятия “тип данных”.
2. Что такое “предикат типа”? Почему и зачем предикаты типа используются в ЯП Scheme?
3. Какими составные типы ЯП Scheme являются встроенными?
4. Значение какого типа является результатом вычисления специальной формы `quote`?
5. Что из себя представляет выражение на языке Scheme?
6. Каково назначение процедуры `eval`? Приведите примеры.
7. Гигиенические макросы в ЯП Scheme: назначение, определение в программе, составные части определения макроса, как это работает (на примере нерекурсивного макроса с одним правилом типа `swap!`, `when`, `unless`).

Задания

1. Реализуйте макрос `trace` для трассировки. Трассировка — способ отладки, при котором отслеживаются значения переменных или выражений на каждом шаге выполнения программы. Необходимость и вывести значение в консоль, и вернуть его в программу нередко требует существенной модификации кода, что может стать источником дополнительных ошибок. Реализуйте макрос, который позволяет ценой небольшой вставки, не нарушающей декларативность кода, выполнить и вывод значения в консоль с комментарием в виде текста выражения, которое было вычислено, и возврат его значения в программу.

   Код без трассировки:

   ```
   (define (zip . xss)
     (if (or (null? xss)
             (null? (car xss))) ; Надо отслеживать значение (car xss) здесь...
         '()
         (cons (map car xss)
               (apply zip (map cdr xss))))) ; ...и значение xss здесь.
   ```

   Код с трассировкой:

   ```
   (load "trace.scm")

   (define (zip . xss)
     (if (or (null? xss)
             (null? (trace-ex (car xss)))) ; Здесь...
         '()
   ```

```
      (cons (map car xss)
            (apply zip (map cdr (trace-ex xss))))))) ; ... и здесь
```

Консоль:

```
> (zip '(1 2 3) '(one two three))
(car xss) => (1 2 3)
xss => ((1 2 3) (one two three))
(car xss) => (2 3)
xss => ((2 3) (two three))
(car xss) => (3)
xss => ((3) (three))
(car xss) => ()
((1 one) (2 two) (3 three))
```

Вычисление значения выражения осуществляется после вывода цитаты этого выражения в консоль. Таким образом, в случае аварийного завершения программы из-за невозможности вычисления значения, вы всегда сможете определить, в каком выражении возникает ошибка.

В дальнейшем используйте этот макрос при отладке своих программ на языке Scheme.

2. Юнит-тестирование — способ проверки корректности отдельных относительно независимых частей программы. При таком подходе для каждой функции (процедуры) пишется набор тестов — пар “выражение — значение, которое должно получится”. Процесс тестирования заключается в вычислении выражений тестов и автоматизированном сопоставлении результата вычислений с ожидаемым результатом. При несовпадении выдается сообщение об ошибках.

   Реализуйте свой каркас для юнит-тестирования. Пусть каркас включает следующие компоненты:

   - Макрос `test` — конструктор теста вида `(выражение ожидаемый-результат)`.
   - Процедуру `run-test`, выполняющую отдельный тест. Если вычисленный результат совпадает с ожидаемым, то в консоль выводятся выражение и признак того, что тест пройден. В противном случае выводится выражение, признак того, что тест не пройден, а также ожидаемый и фактический результаты. Функция возвращает `#t`, если тест пройден и `#f` в противном случае. Вывод цитаты выражения в консоль должен выполняться до вычисления его значения, чтобы при аварийном завершении программы последним в консоль было бы выведено выражение, в котором произошла ошибка.
   - Процедуру `run-tests`, выполняющую серию тестов, переданную ей в виде списка. Эта процедура должна выполнять все тесты в списке и возвращает `#t`, если все они были успешными, в противном случае процедура возвращает `#f`.

   Какой предикат вы будете использовать для сравнения ожидаемого результата с фактическим? Почему?

   Пример:

```
; Пример процедуры с ошибкой
;
(define (signum x)
  (cond
    ((< x 0) -1)
    ((= x 0)  1) ; Ошибка здесь!
```

```
    (else     1)))

; Загружаем каркас
;
(load "unit-test.scm")

; Определяем список тестов
;
(define the-tests
  (list (test (signum -2) -1)
        (test (signum  0)  0)
        (test (signum  2)  1)))

; Выполняем тесты
;
(run-tests the-tests)
```

Пример результата в консоли:

```
(signum -2) ok
(signum 0) FAIL
  Expected: 0
  Returned: 1
(signum 2) ok
#f
```

Используйте разработанные вами средства отладки для выполнения следующих заданий этой лабораторной работы и последующих домашних заданий.

3. Реализуйте процедуру доступа к произвольному элементу последовательности (правильного списка, вектора или строки) по индексу. Пусть процедура возвращает `#f` если получение элемента не возможно. Примеры применения процедуры:

```
(ref '(1 2 3) 1) ⇒ 2
(ref #(1 2 3) 1) ⇒ 2
(ref "123" 1)    ⇒ #\2
(ref "123" 3)    ⇒ #f
```

Реализуйте процедуру “вставки” произвольного элемента в последовательность, в позицию с заданным индексом (процедура возвращает новую последовательность). Пусть процедура возвращает `#f` если вставка не может быть выполнена. Примеры применения процедуры:

```
(ref '(1 2 3) 1 0)   ⇒ (1 0 2 3)
(ref #(1 2 3) 1 0)   ⇒ #(1 0 2 3)
(ref #(1 2 3) 1 #\0) ⇒ #(1 #\0 2 3)
(ref "123" 1 #\0)    ⇒ "1023"
(ref "123" 1 0)      ⇒ #f
(ref "123" 3)        ⇒ #f
```

Попробуйте предусмотреть все возможные варианты.

4. Разработайте наборы юнит-тестов для выполнения домашних заданий “Разложение на множители” и “Символьное дифференцирование”. Начните работу над этими задачами, используя прием разработки через тестирование.