Лекция 9а. Ввод-вывод в языке Scheme

Коновалов А. В.

17 октября 2022 г.

Мы будем рассматривать сегодня ввод-вывод в языке Scheme R^5RS. Язык Scheme R^5RS — не промышленный, а академический, поэтому средства ввода-вывода в нём довольно ограничены.

Для сравнения, Common Lisp — промышленный язык, в нём средства ввода-вывода более обширны.

Для абстракции ввода-вывода в Scheme R5RS используется понятие порта. Есть порт ввода и порт вывода по умолчанию, можно создавать новые порты, связанные с файлами.

Порты ввода-вывода, открытие и закрытие

Порты ввода-вывода

Порт ввода по умолчанию связан с клавиатурой (`stdin`, в терминах языка Си), порт вывода — с экраном (`stdout`, в терминах языка Си). Эти порты по умолчанию можно переназначать.

Предикат типа «порт»:

```
(port? x)                          → #t или #f
```

Порты по умолчанию:

```
(current-input-port)               → port
(current-output-port)              → port
```

Открытие и закрытие портов

Создание порта:

```
(open-input-file "имя файла")      → port
(open-output-file "имя файла")     → port
```

Предусловие для `open-output-port`: файл существовать не должен (иначе ошибка).

После использования порты, связанные с файлами, нужно закрывать:

```
(close-input-port port)
(close-output-port port)
```

Перенаправление портов

Временное перенаправление портов:

```
(with-output-to-file "имя файла" proc)      ≡ (proc)
(with-input-from-file "имя файла" proc)     ≡ (proc)
```

Здесь `proc` — процедура без параметров, во время выполнения этой процедуры порты вывода и ввода, соответственно, по умолчанию будут перенаправлены.

Возвращаемое значение у этих процедур то же, что у вызванной процедуры.

```
(with-output-to-file "D:\\test.txt"
  (lambda ()
    (display 'hello)))
```

В файл `D:\test.txt` будет записана строчка `hello`.

Открытие портов с автоматическим закрытием

Открытие портов с автоматическим закрытием:

```
(call-with-input-file "имя файла" proc)      ≡ (proc port)
(call-with-output-file "имя файла" proc)     ≡ (proc port)
```

Здесь процедура `proc` будет принимать порт в качестве параметра:

```
(call-with-output-file "D:/test2.txt"
  (lambda (port)
    (display 'hello port)))
```

Порт, переданный в процедуру, будет закрыт при завершении вызова самой процедуры.

Символьный ввод-вывод (1)

Для чтения и записи используются процедуры `read-char`, `peek-char`, `write-char`:

```
(read-char)                    → char | eof-object
(read-char port)               → char | eof-object

(peek-char)                    → char | eof-object
(peek-char port)               → char | eof-object

(write-char char)
(write-char char port)
```

Если порт не указан, то используется порт по умолчанию.
Процедуры `read-char` и `peek-char` возвращают либо литеру, либо признак конца файла (end-of-file).

Символьный ввод-вывод (2)

Проверка прочитанного на конец файла делается предикатом:

```
(eof-object? obj)                    → #t | #f
```

Процедура `read-char` читает литеру и забирает его из источника, процедура `peek-char` — читает литеру и оставляет её в источнике.

Символьный ввод-вывод (3)

Вызов

```
(let* ((x (read-char))
       (y (read-char))
       (z (read-char)))
  (list x y z))
```

вернёт три последовательных символа из порта. Вызов

```
(let* ((x (peek-char))
       (y (peek-char))
       (z (peek-char)))
  (list x y z))
```

построит список из трёх одинаковых литер, причём литера останется во входом порту — её можно будет получить следующим вызовом `read-char` или `peek-char`.

Ввод-вывод выражений (1)

Можно читать и записывать целые s-выражения, синтаксический анализ будет выполнен библиотекой, а мы получим готовое выражение.

```
(write expr)
(write expr port)
```

Процедура `write` выписывает в порт выражение в машиночитаемом виде. Т.е. выписанное выражение однозначно понятно. Для чтения машиной записанного выражения используется процедура

```
(read)                    → expr | eof-object
(read port)               → expr | eof-object
```

То, что мы записали при помощи `write`, мы можем потом прочитать при помощи `read`. Однако, не все данные поддаются чтению обратно.

Ввод-вывод выражений (2)

Снова прочитать мы можем только данные следующих типов:

- `#t`, `#f`,
- числа: `100500`, `2/3`, `3.1415926`, `7+2i`,
- строки: `"Hello!"`,
- литеры: `#\H`, `#\!`, `#\newline`,
- символы: `'hello`,
- списки и вектора всего вышеперечисленного.

Снова прочитать мы можем только объекты, для которых существуют литералы. Собственно, процедура `write` и выписывает данные в виде литералов, а процедура `read` их разбирает.

Ввод-вывод выражений (3)

Снова прочитать мы не можем объекты, существующие при выполнении конкретного процесса: процедуры (`lambda`), порты, продолжения (continuations). Для двух последних литералов не существует. Синтаксис `(lambda …)` можно считать литералом для процедуры, но процедура — вещь существенно динамическая, т.к. захватывает переменные из своего окружения (см. идиому статических переменных).

Прочий вывод (1)

Здесь рассмотрим вывод человекочитаемых данных, он представлен двумя процедурами:

```
(display)
(display port)
(newline)
(newline port)
```

Прочий вывод (2)

`display` выводит данные в человекочитаемом виде. Т.е. строки выводятся не как свои литералы, а с буквальной интерпретацией символов в них (перевод строки приведёт к печати перевода строки в файле, а не выдаче литер `\` и `n`). Вывод следующих трёх вызовов будет идентичен:

```
(display #\x)
(display "x")
(display 'x)
```

По выводу будет непонятно, что же было реальным аргументом. Но, когда пользуются процедурой `display`, это и не нужно.

Прочий вывод (3)

Если для отладки нужно выводить какое-то выражение, то его лучше выводить при помощи `write`, т.к. по выводу будет однозначно понятно содержимое. В частности, `write` следует использовать в макросе `trace-ex` и каркасе модульных тестов в ЛР3 для вывода выражений.

REPL (read-evaluate-print loop)

REPL (read-evaluate-print loop) — режим интерактивной работы с интерпретируемыми языками программирования. Пользователь вводит конструкцию языка (выражение, оператор), она тут же интерпретируется и результат выводится на экран. После чего пользователь может снова что-то ввести.

Впервые REPL появился для языка LISP, сейчас он поддерживается многими интерпретаторами языков программирования. Например, среда IDLE в Python, консоль JavaScript, доступная в любом браузере (часто вызывается по F12).

«Самодельный» REPL (1)

REPL можно реализовать в Scheme самостоятельно:

```
(define (print expr)
  (write expr)
  (newline))

(define (REPL)
  (let* ((e (read))                                    ; read
         (r (eval e (interaction-environment)))        ; eval
         (_ (print r)))                                ; print
    (REPL)))                                           ; loop
```

«Самодельный» REPL (2)

Добавим поддержку конца файла:

```
(define (REPL)
  (let* ((e (read)))                                    ; read
    (if (not (eof-object? e))
        (let* ((r (eval e (interaction-environment)))
                                                        ; eval
               (_ (print r)))                           ; print
          (REPL)))))                                    ; loop
```

Процедура `load`

Встроенная процедура `load` позволяет прочитать
и проинтерпретировать содержимое файла:

```
(load "trace.scm")
(load "unit-tests.scm")
```

Аналог процедура `load` можно написать самостоятельно:

```
(define (my-load filename)
  (with-input-from-file filename REPL))
```

«Самодельный» REPL (3)

Примечание. Чтобы среда DrRacket не печатала `#<void>` для конструкций без значения в нашем импровизированном REPL'е, функцию `print` можем уточнить:

```
(define the-void (if #f #f))

(define (print expr)
  (if (not (equal? expr the-void))
      (begin
        (write expr)
        (newline))))
```