[ИУ9] Основы информатики

Лекция 2. Языки семейства LISP. Язык программирования Scheme

LISP (от **LIS**t **P**rocessing) — язык программирования, созданный Джоном МакКарти в 1950-1960-е годы. Породил целое семейство языков со сходным синтаксисом и идеологией: Common Lisp, Scheme, Closure и т.д.

Scheme — язык семейства LISP, созданный Гаем Стилом и Джеральдом Сассманом в 1970-е годы. Отличается простотой и минималистичным дизайном.

Диалект Scheme используется в книге Абельсона и Сассмана «Структура и интерпретация компьютерных программ» (известна под аббревиатурой SICP). Первые две главы этой книги содержат основы программирования на Scheme (но без макросов), ими можно пользоваться в качестве учебника.

В нашем курсе мы будем использовать диалект Scheme R^{5}RS. Официальную спецификацию этого диалекта можно прочитать в PDF-ке r5rs.pdf.

Основные постулаты языков семейства Lisp:

1. Единство кода и данных.
2. Всё есть список.
3. Выражения являются списком, операция указывается в первом элементе.
4. Все выражения вычисляют значения.

Списки можно описать следующим образом:

- Список — последовательность термов (возможно пустая) в круглых скобках:

```
(терм терм терм…)
()  ;; это пустой список
```

- Терм — это либо атом, либо список.
- Атом — имя переменной, число, символ или строка. Подробнее разновидности атомов мы изучим позже.

Примеры списков:

- `(1 2 3 4)` — список из четырёх чисел,
- `()` — пустой список,

- `((a b c) (d e) (f g h i) (j))` — список из четырёх списков, в которых лежат имена,
- `(1 (a 7) 6.022e23 ((())) 3/2 "abcd")` — список, содержащий значения разных типов.

Грамматику списков можно (упрощённо!) следующим образом описать при помощи БНФ (формы Бэкуса-Наура):

```
<терм>   ::= <атом> | <список>
<список> ::= (<термы>)
<термы>  ::= <пусто> | <терм> <термы>
<атом>   ::= <переменная> | <число> | <символ> | <строка>
```

В выражениях языка Scheme после открывающей круглой скобки указывается операция. Операцией может быть либо применение процедуры, либо так называемая **особая форма.** В случае применения функции первым термом после скобок является имя процедуры или выражение, порождающее процедуру. В случае особой формы после открывающей круглой скобки располагается **ключевое слово.**

Примеры:

- `(define pi (* 4 (atan 1)))` — особая форма (определение переменной),
- `(if (> x 0) + -)` — особая форма (ветвление),
- `(atan 1)` — применение процедуры `atan`,
- `(* 4 (atan 1))` — тоже применение процедуры, теперь процедуры `*`,
- `((if (> x 0) + -) x)` — тоже применение процедуры, процедуры, которую нужно применить, вычисляется особой формой — ветвлением,
- `(1 2 3)` — с точки зрения интерпретатора — применение процедуры, который, однако, приведёт к ошибке во время выполнения, т.к. константа `1` — не процедура, а целое число, его применять нельзя.

Значения языка Scheme

- Числа: `1`, `1.0`, `6.022e23`, `1/3`...
- Строки: `"Scheme"`
- Логический тип: `#t`, `#f`
- Литерный (character) тип: `#\a` `#\newline`...
- Символьный (symbol) тип: `'x`, `'sin`
- Списки
- Вектора
- Процедурный тип — значение, которое можно применять, либо встроенная процедура, либо значение, порождаемое особой формой `(lambda …)`.
- Продолжения (continuations) — снимок состояния вычисления, его тоже можно применить, но оно восстановит предыдущее состояние интерпретатора.

- Отложенные вычисления с мемоизацией (promise, «обещание»).

Глобальные переменные

Переменные в Scheme определяются при помощи конструкции `define`. Её синтаксис:

```
(define ‹имя› ‹выражение›)  ;; определяет переменную

(define (‹имя› ‹параметр›…) ;; определяет функцию
  ‹вложенный define›…
  ‹выражение›…
  ‹выражение›)
```

Первая форма определяет переменную с именем `‹имя›` и связывает её со значением `‹выражения›`. (В Scheme принято говорить, что переменная *связывается со значением,* а не присваивается.)

Вторая форма определяет переменную с именем `‹имя›` и связывает её с процедурой. Тело процедуры состоит из нуля и более вложенных определений (т.е. особых форм `define`, определяющих переменные и процедуры) и одного и более выражений. Параметры процедуры — это имена переменных. Параметры и имена, определённые вложенными `define`, являются локальными, т.е. видимыми только внутри данной процедуры (и вложенных в неё процедур). Соответственно, при рекурсивных вызовах локальные переменные разных вызовов могут иметь разные значения.

Если выражений в теле процедуры несколько, то они вычисляются *последовательно* (аспект императивного программирования на Scheme). Возвращаемым значением функции (т.е. тем значением, которое примет выражение вызова функции) будет значение последнего выражения, значения предшествующих выражений игнорируются. При написании чистых функций (не использующих императивные возможности Scheme) все выражения, кроме последнего, писать бессмысленно. Их обычно пишут ради побочного эффекта.

Примеры:

```
;; Определения переменных:
(define pi (* 4 (atan 1)))
(define e (exp 1))

;; Определения процедуры
(define (my-abs x)
  ((if (> x 0) + -) x))

(define (circle-area r)
  (* pi r r))
```

```
(define (rect-area a b)
  (* a b))
```

Да, кстати, в Scheme комментарии однострочные и начинаются со знака `;`, для наглядности принято знак `;` ставить дважды.

Определение процедуры с помощью `define` является сокращённой записью для следующей конструкции:

```
;; запись
(define (‹имя› ‹парам›…)
  ‹тело›)
```

```
;; означает
(define ‹имя›
   (lambda (‹параметры›…)
     ‹тело›))
```

Особая форма `lambda` порождает новую процедуру, `define` связывает имя с процедурой, порождённой `lambda`.

Логические значения языка Scheme

Для представления логических значений в языке Scheme используются две константы `#t` — истина и `#f` — ложь. Процедуры, которые возвращают логическое значение (т.е. `#t` или `#f`), называются **предикатами**. Имена предикатов принято завершать знаком `?`, например `(even? x)` — `#t`, если `x` чётный, `#f`, если нечётный.

Однако, в управляющих конструкциях любое значение, кроме `#f`, считается истинным.

Ветвление — особая форма `if`

Базовой конструкцией для ветвления является особая форма `if`, имеющая вид

```
(if ‹условие›
    ‹выражение-для-истины›
    ‹выражение-для-лжи›)
```

Сначала вычисляется `‹условие›`, затем, в зависимости от его значения,

- если оно не равно `#f`, вычисляется `‹выражение-для-истины›` и значением этой особой формы становится значение этого выражения,
- если оно равно `#f`, вычисляется `‹выражение-для-лжи›`, значение которого становится значением особой формы `if`.

Выражение для лжи может отсутствовать:

```
(if ‹условие›
    ‹выражение-для-истины›)
```

Тогда, если условие ложное (равно `#f`), значение особой формы не определено. Такая разновидность `if` используется в императивном программировании, когда `‹выражение-для-истины›` вычисляется ради побочного эффекта.

Логические операции `and`, `or`, `not`.

`not` является обычной встроенной функцией языка, в то время как `and` и `or` являются особыми формами.

Семантику функции `not` можно условно описать следующей функцией:

```
(define (not value)
  (if value
      #f
      #t))
```

На самом деле это встроенная функция, среда DrRacket её переопределять не позволяет.

`and` и `or` — особые формы, имеют вид

```
(and ‹выражение›…)
(or ‹выражение›…)
```

Они не являются функциями, т.к. могут вычислять не все свои аргументы:

- Результат логического И истиннен, если являются истинными все его аргументы, если хотя бы один из них является ложным (равен `#f`) — весь результат ложь. После первого значения, равного `#f`, вычисление аргументов прекращается. Если все аргументы истинные (т.е. ни один из них не является `#f`), результатом особой формы `and` является результат последнего аргумента.
- Результат логического ИЛИ истиннен, если хотя бы один из аргументов является истинным. Поэтому особая форма `or` вычисляет свои аргументы до тех пор, пока не встретися первое значение, не равное `#f` и это значение становится значением всей особой формы — последующие значения не вычисляются. Если значения всех выражений оказались равны `#f`, то и результатом становится `#f`.

Таким образом, следующие функции никогда не дадут ошибку деления на ноль:

```
(define (f1 x)
  (or (= x 0) (/ 1 x)))

(define (f2 x)
  (and (> x 0) (/ 2 x)))
```

Ветвление — особые формы `cond` и `case`

Синтаксис:

```
(cond
  (‹условие-1› ‹выражение›)
  (‹условие-2› ‹выражение›)
  …
  (else ‹выражение›))
```

По очереди вычисляются выражения `‹условие-1›`, `‹условие-2›` и т.д., пока не обнаружится первое условие, значение которого истинное (т.е. опять не равное `#f`), после чего вычисляется выражение в соответствующей ветке. Значение этого выражения становится значением всей конструкции `cond`. Если ни одно из выражений не оказалось истинным, то вычисляется выражение в ветке `else`.

Если ветка `else` отсутствует, и все условия ложные, то результат не определён. Так обычно пишут в парадигме императивного программирования, когда выражения в ветках используются только ради побочного эффекта.

Существует вот такой синтаксис:

```
(cond
  …
  (‹условие-i› -> ‹выражение›)
  …
  (else ‹выражение›))
```

В этом случае выражение должно вычислять процедуру (например, быть именем процедуры). Тогда, если условие истинное, вызывается `(‹выражение› ‹значение-условия›)`, т.е. соответствующая процедура вызывается с аргументом — неложным значением условия.

Особая форма `case` оставляется на самостоятельное изучение.

Как писать не надо

```
(and <усл>
     <выраж>)            ; корректная запись
(if <усл>
    <выраж>
    #f)                  ; некорректная запись
```

```
(or <усл> <выраж>)       ; корректно
(if <усл>
```

```
    #t
    <выраж)                ; некорректно
```

```
(and (not <усл>) <выраж>) ; корректно
(if <усл>
    #f
    <выраж>)               ; некорректно
```

```
(or (not <усл>) <выраж>)  ; корректно
(if <усл>
    <выраж>
    #t)                    ; некорректно
```