Лекция 4.

Строки, последовательности, хэш-таблицы, структуры

СТРОКИ И ЗНАКИ

Строки – это векторы, состоящие из знаков. Строкой принято называть набор знаков, заключенный в двойные кавычки. Одиночный знак, например `c`, задается так: `#\c`.

Каждый знак соответствует определенному целому числу, как правило, (хотя и не обязательно) в соответствии с ASCII. В большинстве реализаций есть функция `char-code`, которая возвращает связанное со знаком число, и функция `code-char`, выполняющая обратное преобразование.°

Для сравнения знаков используются следующие функции: `char<` (меньше), `char<=` (меньше или равно), `char=` (равно), `char>=` (больше или равно), `char>` (больше) и `char/=` (не равно). Они работают так же, как и функции сравнения чисел, которые рассматриваются на стр. 157.

```
> (sort "elbow" #'char<)
"below"
```

Поскольку строки – это массивы, то к ним применимы все операции с массивами. Например, получить знак, находящийся в конкретной позиции, можно с помощью aref:

```
> (aref "abc" 1)
#\b
```

Однако эта операция может быть выполнена быстрее с помощью специализированной функции char:

```
> (char "abc" 1)
#\b
```

Функция char, как и aref, может быть использована вместе с setf для замены элементов:

```
> (let ((str (copy-seq "Merlin")))
    (setf (char str 3) #\k)
    str)
"Merkin"
```

Чтобы сравнить две строки, можно воспользоваться известной вам функцией `equal`, но есть также и специализированная `string-equal`, которая к тому же не учитывает регистр букв:

```
> (equal "fred" "fred")
T
> (equal "fred" "Fred")
NIL

> (string-equal "fred" "Fred")
T
```

Есть несколько способов создания строк. Самый общий – с помощью функции `format`. При использовании `nil` в качестве ее первого аргумента `format` вернет строку, вместо того чтобы ее напечатать:

```
> (format nil "~A or ~A" "truth" "dare")
"truth or dare"
```

Но если вам нужно просто соединить несколько строк, можно воспользоваться `concatenate`, которая принимает тип результата и одну или несколько последовательностей:

```
> (concatenate 'string "not " "to worry")
"not to worry"
```

ПОСЛЕДОВАТЕЛЬНОСТИ

Тип *последовательность* (*sequence*) в Common Lisp включает в себя списки и векторы (а значит, и строки). Многие функции из тех, которые мы ранее использовали для списков, на самом деле определены для любых последовательностей. Это, например, `remove`, `length`, `subseq`, `reverse`, `sort`, `every`, `some`. Таким образом, функция, определенная нами на стр. 62, будет работать и с другими видами последовательностей:

```
> (mirror? "abba")
T
```

Мы уже знаем некоторые функции для доступа к элементам последовательностей: `nth` для списков, `aref` и `svref` для векторов, `char` для строк. Доступ к элементу последовательности любого типа может быть осуществлен с помощью `elt`:

```
> (elt '(a b c) 1)
B
```

Специализированные функции работают быстрее, и использовать `elt` рекомендуется только тогда, когда тип последовательности заранее не известен.

С помощью `elt` функция `mirror?` может быть оптимизирована для векторов:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (do ((forward 0 (+ forward 1))
              (back (- len 1) (- back 1)))
             ((or (> forward back)


                  (not (eql (elt s forward)
                            (elt s back))))
              (> forward back))))))
```

Эта версия по-прежнему будет работать и со списками, однако она более приспособлена для векторов. Регулярное использование последовательного доступа к элементам списка довольно затратно, а непосредственного доступа к нужному элементу они не предоставляют. Для векторов же стоимость доступа к любому элементу не зависит от его положения.

Многие функции, работающие с последовательностями, имеют несколько аргументов по ключу:

Параметр	Назначение	По умолчанию
:key	Функция, применяемая к каждому элементу	identity
:test	Предикат для сравнения	eql
:from-end	Если t, работа с конца	nil
:start	Индекс элемента, с которого начинается выполнение	0
:end	Если задан, то индекс элемента, на котором следует остановиться	nil

Одна из функций, которая принимает все эти аргументы, – position. Она возвращает положение определенного элемента в последовательности или nil в случае его отсутствия. Посмотрим на роль аргументов по ключу на примере position:

```
> (position #\a "fantasia")
1
> (position #\a "fantasia" :start 3 :end 5)
4
```

Во втором случае поиск выполняется между четвертым и шестым элементом. Аргумент :start ограничивает подпоследовательность слева, :end ограничивает справа или же не ограничивает вовсе, если этот аргумент не задан.

Задавая параметр :from-end:

```
> (position #\a "fantasia" :from-end t)
7
```

мы получаем позицию элемента, ближайшего к концу последовательности. Но позиция элемента вычисляется как обычно, то есть от начала списка (однако поиск элемента производится с конца списка).

Параметр :key определяет функцию, применяемую к каждому элементу перед сравнением его с искомым:

```
> (position 'a '((c d) (a b)) :key #'car)
1
```

В этом примере мы поинтересовались, car какого элемента содержит a.

Параметр :test определяет, с помощью какой функции будут сравниваться элементы. По умолчанию используется eql. Если вам необходимо сравнивать списки, придется воспользоваться функцией equal:

```
> (position '(a b) '((a b) (c d)))
NIL
> (position '(a b) '((a b) (c d)) :test #'equal)
0
```

Аргумент :test может быть любой функцией от двух элементов. Например, с помощью < можно найти первый элемент, больший заданного:

```
> (position 3 '(1 0 7 5) :test #'<)
2
```

С помощью subseq и position можно разделить последовательность на части. Например, функция

```
(defun second-word (str)
  (let ((p1 (+ (position #\  str) 1)))
    (subseq str p1 (position #\  str :start p1))))
```

возвращает второе слово в предложении:

```
> (second-word "Form follows function.")
"follows"
```

Поиск элементов, удовлетворяющих заданному предикату, осуществляется с помощью position-if. Она принимает функцию и последовательность, возвращая положение первого встреченного элемента, удовлетворяющего предикату:

```
> (position-if #'oddp '(2 3 4 5))
1
```

Эта функция принимает все вышеперечисленные аргументы по ключу, за исключением :test.

Также для последовательностей определены функции, аналогичные member и member-if. Это find (принимает все аргументы по ключу) и find-if (принимает все аргументы, кроме :test):

```
> (find #\a "cat")
#\a
> (find-if #'characterp "ham")
#\h
```

В отличие от member и member-if, они возвращают только сам найденный элемент.

Вместо find-if иногда лучше использовать find с ключом :key. Например, выражение:

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

будет выглядеть понятнее в виде:

```
(find 'complete lst :key #'car)
```

Функции remove и remove-if работают с последовательностями любого типа. Разница между ними точно такая же, как между find и find-if. Связанная с ними функция remove-duplicates удаляет все повторяющиеся элементы последовательности, кроме последнего:

```
> (remove-duplicates "abracadabra")
"cdbra"
```

Эта функция использует все аргументы по ключу, рассмотренные в таблице выше.

Функция reduce сводит последовательность в одно значение. Она принимает функцию, по крайней мере, с двумя аргументами и последовательность. Заданная функция первоначально применяется к первым двум элементам последовательности, а затем последовательно к полученному результату и следующему элементу последовательности. Последнее полученное значение будет возвращено как результат reduce. Таким образом, вызов:

```
(reduce #'fn '(a b c d))
```

будет эквивалентен

```
(fn (fn (fn 'a 'b) 'c) 'd)
```

Хорошее применение reduce – расширение набора аргументов для функций, которые принимают только два аргумента. Например, чтобы получить пересечение трех или более списков, можно написать:

```
> (reduce #'intersection '((b r a d 's) (b a d) (c a t)))
(A)
```

Пример: разбор дат

В качестве примера операций с последовательностями в этом разделе приводится программа для разбора дат. Мы напишем программу, которая превращает строку типа "16 Aug 1980" в целые числа, соответствующие дню, месяцу и году.

Программа на рис. 4.2 содержит некоторые функции, которые потребуются нам в дальнейшем. Первая, tokens, выделяет знаки из строки. Функция tokens принимает строку и предикат, возвращая список подстрок, все знаки в которых удовлетворяют этому предикату. Приведем пример. Пусть используется функция alpha-char-p – предикат, справедливый для буквенных знаков. Тогда получим:

```
> (tokens "ab12 3cde.f" #'alpha-char-p 0)
("ab" "cde" "f")
```

Все остальные знаки, не удовлетворяющие данному предикату, рассматриваются как пробельные.

```
(defun tokens (str test start)
 (let ((p1 (position-if test str :start start)))
   (if p1
       (let ((p2 (position-if #'(lambda (c)
                                  (not (funcall test c)))
                              str :start p1)))
         (cons (subseq str p1 p2)
               (if p2
                   (tokens str test p2)
                   nil)))
       nil)))

(defun constituent (c)
  (and (graphic-char-p c)
       (not (char= c #\ ))))
```

Рис. 4.2. Распознавание символов

Функция constituent будет использоваться в качестве предиката для tokens. В Common Lisp к *печатным знакам* (*graphic characters*) относятся все знаки, которые видны при печати, а также пробел. Вызов tokens с функцией constituent будет выделять подстроки, состоящие из печатных знаков:

```
> (tokens "ab12 3cde.f
          gh" #'constituent 0)
("ab12" "3cde.f" "gh")
```

На рис. 4.3 показаны функции, выполняющие разбор дат.

```
(defun parse-date (str)
  (let ((toks (tokens str #'constituent 0)))
    (list (parse-integer (first toks))
          (parse-month (second toks))
          (parse-integer (third toks)))))

(defconstant month-names
  #("jan" "feb" "mar" "apr" "may" "jun"
    "jul" "aug" "sep" "oct" "nov" "dec"))

(defun parse-month (str)
  (let ((p (position str month-names
                     :test #'string-equal)))
    (if p
        (+ p 1)
        nil)))
```

Рис. 4.3. Функции для разбора дат

Функция parse-date принимает дату, записанную в указанной форме, и возвращает список целых чисел, соответствующих ее компонентам:

```
> (parse-date "16 Aug 1980")
(16 8 1980)
```

Эта функция делит строку на части и применяет parse-month и parse-integer к полученным частям. Функция parse-month не чувствительна к регистру, так как сравнивает строки с помощью string-equal. Для преобразования строки, содержащей число, в само число, используется встроенная функция parse-integer.

Однако если бы такой функции в Common Lisp изначально не было, нам пришлось бы определить ее самостоятельно:

```
(defun read-integer (str)
  (if (every #'digit-char-p str)
      (let ((accum 0))
        (dotimes (pos (length str))
          (setf accum (+ (* accum 10)
                         (digit-char-p (char str pos)))))
        accum)
      nil))
```

Определенная нами функция `read-integer` показывает, как в Common Lisp преобразовать набор знаков в число. Она использует особенность функции `digit-char-p`, которая проверяет, является ли аргумент цифрой, и возвращает саму цифру, если это так.

Хэш-таблицы

Для достаточно больших массивов данных (начиная уже с 10 элементов) использование хеш-таблиц существенно увеличит производительность. Хеш-таблицу можно создать с помощью функции `make-hash-table`, которая не требует обязательных аргументов:

```
> (setf ht (make-hash-table))
#<Hash-Table BF0A96>
```

Хеш-таблицы, как и функции, при печати отображаются в виде `#<...>`.

Хеш-таблица, как и ассоциативный список, – это способ ассоциирования пар объектов. Чтобы получить значение, связанное с заданным ключом, достаточно вызвать gethash с этим ключом и таблицей. По умолчанию gethash возвращает nil, если не находит искомого элемента.

```
> (gethash 'color ht)
NIL
NIL
```

Здесь мы впервые сталкиваемся с важной особенностью Common Lisp: выражение может возвращать несколько значений. Функция `gethash` возвращает два. Первое значение ассоциировано с ключом. Второе значение, если оно `nil` (как в нашем примере), означает, что искомый элемент не был найден. Почему мы не можем судить об этом из первого `nil`? Дело в том, что элементом, связанным с ключом `color`, может оказаться `nil`, и `gethash` вернет его, но в этом случае в качестве второго значения — `t`.

Чтобы сопоставить новое значение какому-либо ключу, используем `setf` вместе с `gethash`:

```
> (setf (gethash 'color ht) 'red)
RED
```

Теперь `gethash` вернет вновь установленное значение:

```
> (gethash 'color ht)
RED
T
```

Второе значение подтверждает, что `gethash` вернул реально имеющийся в таблице объект, а не значение по умолчанию.

Объекты, хранящиеся в хеш-таблицах, могут иметь любой тип. Например, при желании сопоставить каждой функции ее краткое описание можно создать таблицу, в которой ключами будут функции, а значениями – строки:

```
> (setf bugs (make-hash-table))
#<Hash-Table BF4C36>
> (push "Doesn't take keyword arguments. "
        (gethash #'our-member bugs))
("Doesn't take keyword arguments. ")
```

Так как по умолчанию `gethash` возвращает `nil`, вызов `push` эквивалентен `setf`, и мы просто кладем нашу строку на пустой список.

Хеш-таблицы также можно использовать вместо списков для представления множеств. Они существенно ускоряют поиск значений и их удаление в случаях больших объемов данных. Чтобы добавить элемент в множество, представленное в виде хеш-таблицы, используйте `setf` вместе с `gethash`:

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash 'apricot fruit) t)
T
```

Проверка на принадлежность элемента множеству выполняется с помощью gethash:

```
> (gethash 'apricot fruit)
T
T
```

По умолчанию gethash возвращает nil, поэтому вновь созданная хеш-таблица представляет собой пустое множество.

Чтобы удалить элемент из множества, можно воспользоваться remhash:

```
> (remhash 'apricot fruit)
T
```

Возвращая t, remhash сигнализирует, что искомый элемент был найден и успешно удален.

Для итерации по хеш-таблице существует maphash, которой необходимо передать функцию двух аргументов и саму таблицу. Эта функция будет вызвана с каждой имеющейся в таблице парой ключ-значение в произвольном порядке.

```
> (setf (gethash 'shape ht) 'spherical
        (gethash 'size ht) 'giant)
GIANT
> (maphash #'(lambda (k v)
               (format t "~A = ~A~%" k v))
           ht)
SHAPE = SPHERICAL
SIZE = GIANT
COLOR = RED
NIL
```

Функция maphash всегда возвращает nil, однако вы можете сохранить данные, если передадите функцию, которая, например, будет накапливать результаты в списке.

Хеш-таблицы могут накапливать любое количество вхождений, так как способны расширяться в процессе работы, когда места для хранения элементов перестанет хватать. Задать исходную емкость таблицы можно с помощью ключа `:size` функции `make-hash-table`. Используя этот параметр, вам следует помнить о двух вещах: большой размер таблицы позволит избежать ее частых расширений (это довольно затратная процедура), однако создание таблицы большого размера для малого набора данных приведет к необоснованным затратам памяти. Важный момент:

параметр `:size` определяет не количество хранимых объектов, а количество пар ключ-значение. Выражение:

```
(make-hash-table :size 5)
```

вернет хеш-таблицу, которая сможет вместить пять вхождений до того, как ей придется расширяться.

Как и любая структура, подразумевающая возможность поиска элементов, хеш-таблица может использовать различные предикаты проверки эквивалентности. По умолчанию используется `eql`, но также можно применить `eq`, `equal` или `equalp`; используемый предикат указывается с помощью ключа `:test`:

```
> (setf writers (make-hash-table :test #'equal)
#<Hash-Table C005E6>
> (setf (gethash '(ralph waldo emerson) writers) t)
T
```

Это один из компромиссов, с которым нам приходится мириться ради получения эффективных хеш-таблиц. Работая со списками, мы бы воспользовались функцией `member`, которой можно каждый раз сообщать различные предикаты проверки. При работе с хеш-таблицами мы должны определиться с используемым предикатом заранее, в момент ее создания.

С необходимостью жертвовать одним ради другого в Лисп-разработке (да и в жизни в целом) вам придется столкнуться не раз. Это часть философии Лиспа: первоначально вы миритесь с низкой производительностью ради удобства разработки, а по мере развития программы можете пожертвовать ее гибкостью ради скорости выполнения.

Структуры

Структура может рассматриваться как более продвинутый вариант вектора. Предположим, что нам нужно написать программу, отслеживающую положение набора параллелепипедов. Каждое такое тело можно представить в виде вектора, состоящего из трех элементов: высота, ширина и глубина. Программу будет проще читать, если вместо простых `svref` мы будем использовать специальные функции:

```
(defun block-height (b) (svref b 0))
```

и так далее. Можете считать структуру таким вектором, у которого все эти функции уже заданы.

Определить структуру можно с помощью `defstruct`. В простейшем случае достаточно задать имена структуры и ее полей:

```
(defstruct point
  x
  y)
```

Мы определили структуру `point`, имеющею два поля, `x` и `y`. Кроме того, неявно были заданы функции: `make-point`, `point-p`, `copy-point`, `point-x`, `point-y`.

В разделе 2.3 мы упоминали о способности Лисп-программ писать другие Лисп-программы. Это один из наглядных примеров: при вызове `defstruct` самостоятельно определяет все необходимые функции. Научившись работать с макросами, вы сами сможете делать похожие вещи. (Вы даже смогли бы написать свою версию `defstruct`, если бы в этом была необходимость.)

Каждый вызов `make-point` возвращает вновь созданный экземпляр структуры `point`. Значения полей могут быть изначально заданы с помощью соответствующих аргументов по ключу:

```
> (setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
```

Функции доступа к полям структуры определены не только для чтения полей, но и для задания значений с помощью `setf`:

```
> (point-x p)
0
> (setf (point-y p) 2)
2
> p
#S(POINT X 0 Y 2)
```

Определение структуры также приводит к определению одноименного типа. Каждый экземпляр `point` принадлежит типу `point`, затем `structure`, затем `atom` и `t`. Таким образом, использование `point-p` равносильно проверке типа:

```
> (point-p p)
T
> (typep p 'point)
T
```

Функция `typep` проверяет объект на принадлежность к заданному типу.

Также можно задать значения полей по умолчанию, если заключить имя соответствующего поля в список и поместить в него выражение для вычисления этого значения.

```
(defstruct polemic
  (type (progn
          (format t "What kind of polemic was it? ")
          (read)))
  (effect nil))
```

Вызов `make-polemic` без дополнительных аргументов установит исходные значения полей:

```
> (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC TYPE SCATHING EFFECT NIL)
```

Кроме того, можно управлять такими вещами, как способ отображения структуры и префикс имен функций для доступа к полям. Вот более развитый вариант определения структуры `point`:

```
(defstruct (point (:conc-name p)
                  (:print-function print-point))
  (x 0)
  (y 0))

(defun print-point (p stream depth)
  (format stream "#<~A, ~A>" (px p) (py p)))
```

Аргумент `:conc-name` задает префикс, с которого будут начинаться имена функций для доступа к полям структуры. По умолчанию он равен `point-`, а в новом определении это просто `p`. Отход от варианта по умолчанию делает код менее читаемым, поэтому использовать более короткий префикс стоит, только если вам предстоит постоянно пользоваться функциями доступа к полям.

Параметр `:print-function` – это *имя* функции, которая будет вызываться для печати объекта, когда его нужно будет отобразить (например, в top-level). Такая функция должна принимать три аргумента: сам объект; поток, куда он будет напечатан; третий аргумент обычно не требуется и может быть проигнорирован[1]. С потоками ввода-вывода мы познакомимся подробнее в разделе 7.1. Сейчас достаточно сказать, что второй аргумент, поток, может быть передан функции `format`.

Функция `print-point` будет отображать структуру в такой сокращенной форме:

```
> (make-point)
#<0,0>
```