Лекция 3.

Списки, точечные пары, ассоциативные списки, массивы, строки, структуры

Списки, которые могут быть построены с помощью list, называются *правильными списками* (*proper list*). Правильным списком считается либо nil, либо cons-ячейка, cdr которой – также правильный список. Таким образом, можно определить предикат, который возвращает истину только для правильного списка[1]:

```
(defun proper-list? (x)
  (or (null x)
      (and (consp x)
           (proper-list? (cdr x)))))
```

Оказывается, с помощью cons можно создавать не только правильные списки, но и структуры, содержащие ровно два элемента. При этом car соответствует первому элементу структуры, а cdr – второму.

```
> (setf pair (cons 'a 'b))
(A . B)
```

Оказывается, с помощью cons можно создавать не только правильные списки, но и структуры, содержащие ровно два элемента. При этом car соответствует первому элементу структуры, а cdr – второму.

```
> (setf pair (cons 'a 'b))
(A . B)
```

Поскольку эта cons-ячейка не является правильным списком, при отображении ее car и cdr разделяются точкой. Такие ячейки называются *точечными парами* (рис. 3.10).

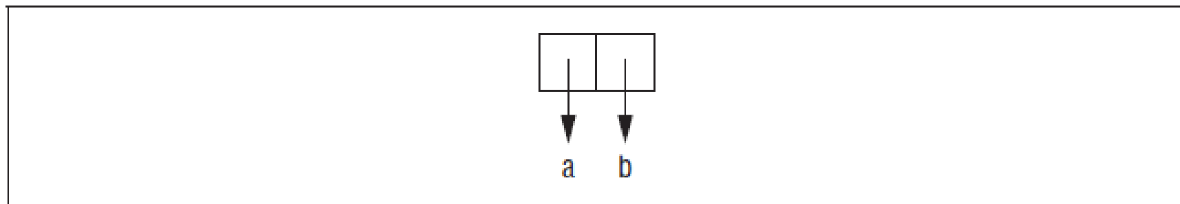

Рис. 3.10. *Ячейка как точечная пара*

Правильные списки можно задавать и в виде набора точечных пар, но они будут отображаться в виде списков:

```
> '(a . (b . (c . nil)))
(A B C)
```

Обратите внимание, как соотносятся ячеечная и точечная нотации – рис. 3.2 и 3.10.

Допустима также смешанная форма записи:

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

Структура такого списка показана на рис. 3.11.

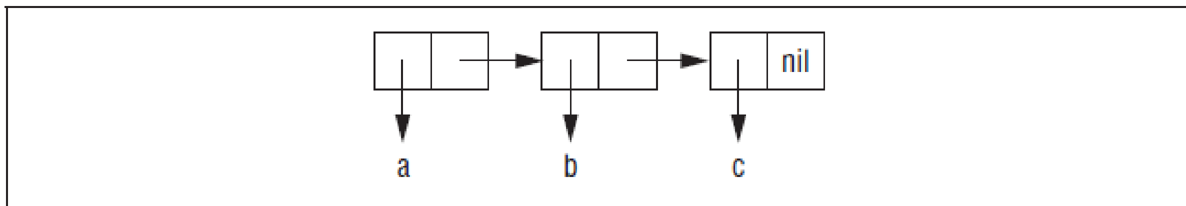

Рис. 3.2. *Список из трех ячеек*

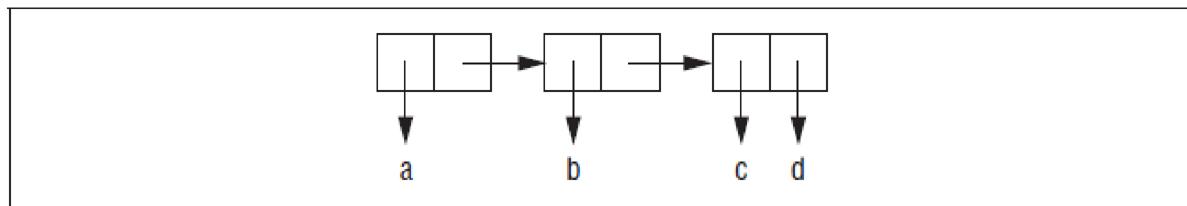

Рис. 3.11. Список точечных пар

Таким образом, список (a b) может быть записан аж четырьмя способами:

```
(a . (b . nil))
(a . (b))
(a b . nil)
(a b)
```

и при этом Лисп отобразит их одинаково.

АССОЦИАТИВНЫЕ СПИСКИ

Также вполне естественно задействовать cons-ячейки для представления отображений. Список точечных пар называется *ассоциативным списком* (*assoc-list*, *alist*). С помощью него легко определить набор каких-либо правил и соответствий, к примеру:

```
> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

Ассоциативные списки медленные, но они удобны на начальных этапах работы над программой. В Common Lisp есть встроенная функция assoc для получения по ключу соответствующей ему пары в таком списке:

```
> (assoc '+ trans)
(+ . "add")
> (assoc '* trans)
NIL
```

Если assoc ничего не находит, возвращается nil.

Попробуем определить упрощенный вариант функции assoc:

```
(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
         (if (eql key (car pair))
             pair
             (our-assoc key (cdr alist))))))
```

Как и member, реальная функция assoc принимает несколько аргументов по ключу, включая :test и :key. Также Common Lisp определяет assoc-if, которая работает по аналогии с member-if.

ПОИСК КРАТЧАЙШЕГО ПУТИ НА ЛИСП

На рис. 3.12 показана программа, вычисляющая кратчайший путь на графе (или сети). Функции `shortest-path` необходимо сообщить начальную и конечную точки, а также саму сеть, и она вернет кратчайший путь между ними, если он вообще существует.

В этом примере узлам соответствуют символы, а сама сеть представлена как ассоциативный список элементов вида (*узел . соседи*).

Небольшая сеть, показанная на рис. 3.13, может быть записана так:

```
(setf min '((a b c) (b c) (c d)))
```

Найти узлы, в которые можно попасть из узла `a`, поможет функция `assoc`:

```
> (cdr (assoc 'a min))
(B C)
```

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
      nil
      (let ((path (car queue)))
        (let ((node (car path)))
          (if (eql node end)
              (reverse path)
              (bfs end
                   (append (cdr queue)
                           (new-paths path node net))
                   net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda (n)
              (cons n path))
          (cdr (assoc node net))))
```

Рис. 3.12. Поиск в ширину

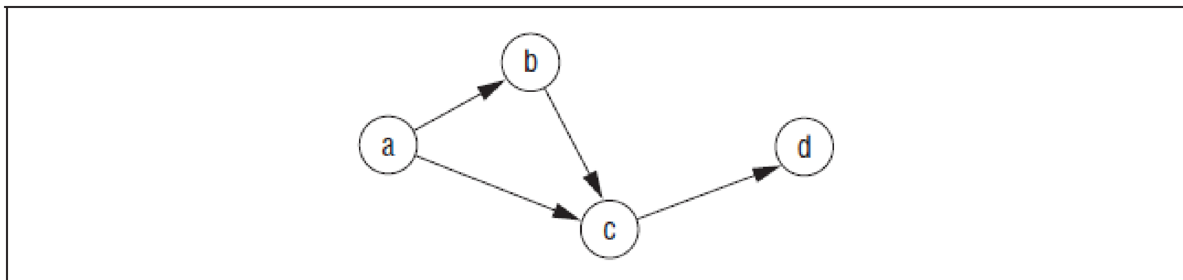

Рис. 3.13. Простейшая сеть

Программа на рис. 3.12 реализует *поиск в ширину* (*breadth-first search*). Каждый слой сети исследуется поочередно друг за другом, пока не будет найден нужный элемент или достигнут конец сети. Последовательность исследуемых узлов представляется в виде очереди.

Приведенный на рис. 3.12 код слегка усложняет эту идею, позволяя не только прийти к пункту назначения, но еще и сохранить запись о том, как мы туда добрались. Таким образом, мы оперируем не с очередью узлов, а с очередью пройденных путей.

Поиск выполняется функцией `bfs`. Первоначально в очереди только один элемент – путь к начальному узлу. Таким образом, `shortest-path` вызывает `bfs` с `(list (list start))` в качестве исходной очереди.

Первое, что должна сделать bfs, – проверить, остались ли еще непройденные узлы. Если очередь пуста, bfs возвращает nil, сигнализируя, что путь не был найден. Если же еще есть непроверенные узлы, bfs берет первый из очереди. Если car этого узла содержит искомый элемент, значит, мы нашли путь до него, и мы возвращаем его, предварительно развернув. В противном случае мы добавляем все дочерние узлы в конец очереди. Затем мы рекурсивно вызываем bfs и переходим к следующему слою.

Так как bfs осуществляет поиск в ширину, то первый найденный путь будет одновременно самым коротким или одним из кратчайших, если есть и другие пути такой же длины:

```
> (shortest-path 'a 'd min)
(A C D)
```

А вот как выглядит соответствующая очередь во время каждого из вызовов bfs:

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

В каждой следующей очереди второй элемент предыдущей очереди становится первым, а первый элемент становится хвостом (cdr) любых новых элементов в конце следующей очереди.

Разумеется, приведенный на рис. 3.12 код далек от полноценной эффективной реализации. Однако он убедительно показывает гибкость и удобство списков. В этой короткой программе мы используем списки тремя различными способами: список символов представляет путь, список путей представляет очередь[1] при поиске по ширине, а ассоциативный список изображает саму сеть целиком.

МАССИВЫ

В Common Lisp массивы создаются с помощью функции `make-array`, первым аргументом которой выступает список размерностей. Создадим массив 2×3:

```
> (setf arr (make-array '(2 3) :initial-element nil))
#<Simple-Array T (2 3) BFC4FE>
```

Многомерные массивы в Common Lisp могут иметь по меньшей мере 7 размерностей, а в каждом измерении поддерживается хранение не менее 1023 элементов[1].

Аргумент `:initial-element` не является обязательным. Если он используется, то устанавливает начальное значение каждого элемента массива. Поведение системы при попытке получить значение элемента массива, не инициализированного начальным значением, не определено.

Чтобы получить элемент массива, воспользуйтесь `aref`. Как и большинство других функций доступа в Common Lisp, `aref` начинает отсчет элементов с нуля:

```
> (aref arr 0 0)
NIL
```

Новое значение элемента массива можно установить, используя `setf` вместе с `aref`:

```
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
```

Как и списки, массивы могут быть заданы буквально с помощью синтаксиса `#na`, где *n* – количество размерностей массива. Например, текущее состояние массива `arr` может быть задано так:

```
#2a((b nil nil) (nil nil nil))
```

Для создания одномерного массива можно вместо списка размерностей в качестве первого аргумента передать функции make-array целое число:

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

Одномерный массив также называют *вектором*. Создать и заполнить вектор можно с помощью функции vector:

```
> (vector "a" 'b 3)
#("a" B 3)
```

Как и массив, который может быть задан буквально с помощью синтаксиса #*n*a, вектор может быть задан буквально с помощью синтаксиса #().

Хотя доступ к элементам вектора может осуществить aref, для работы с векторами есть более быстрая функция svref:

```
> (svref vec 0)
NIL
```

БИНАРНЫЙ ПОИСК

В этом разделе в качестве примера показано, как написать функцию поиска элемента в отсортированном векторе. Если нам известно, что элементы вектора расположены в определенном порядке, то поиск нужного элемента может быть выполнен быстрее, чем с помощью функции `find` (стр. 80). Вместо того чтобы последовательно проверять элемент за элементом, мы сразу перемещаемся в середину вектора. Если средний элемент соответствует искомому, то поиск закончен. В противном случае мы продолжаем поиск в правой или левой половине в зависимости от того, больше или меньше искомого значения этот средний элемент вектора.

На рис. 4.1 приведена программа, которая работает подобным образом. Она состоит из двух функций: `bin-search`[2] определяет границы поиска и передает управление функции `finder`, которая ищет соответствующий элемент между позициями `start` и `end` вектора `vec`.

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
         (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

Рис. 4.1. *Поиск в отсортированном векторе*

Когда область поиска сокращается до одного элемента, возвращается сам элемент в случае его соответствия искомому значению `obj`, в противном случае – `nil`. Если область поиска состоит из нескольких элементов, определяется ее средний элемент – `obj2` (функция `round` возвращает ближайшее целое число), который сравнивается с искомым элементом `obj`. Если `obj` меньше `obj2`, поиск продолжается рекурсивно в левой половине вектора, в противном случае – в правой половине. Остается вариант `obj = obj2`, но это значит, что искомый элемент найден и мы просто его возвращаем.

Если вставить следующую строку в начало определения функции `finder`,

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

мы сможем наблюдать за процессом отсечения половин на каждом шаге:

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

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.

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 toplevel). 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>
```