17

Пример: объекты

В этой главе мы собираемся реализовать внутри Лиспа свой собственный объектно-ориентированный язык. Такие программы называют *встроенными языками* (*embedded language*). Встраивание объектно-ориентированного языка в Лисп – это идеальный пример. Он не только демонстрирует типичное использование Лиспа, но еще и показывает, как естественно ложатся основные понятия объектно-ориентированного программирования на фундаментальные абстракции Лиспа.

17.1. Наследование

В разделе 11.10 объяснялось, чем обобщенные функции отличаются от передачи сообщений. В модели передачи сообщений объекты:

1. Имеют свойства.
2. Реагируют на сообщения.
3. Наследуют свойства и методы от предков.

CLOS, как известно, реализует модель обобщенных функций. Но в этой главе мы заинтересованы в написании простейшей объектной системы, не претендующей на соперничество с CLOS, поэтому будем использовать более старую модель.

Лисп предоставляет несколько способов хранения наборов свойств. Первый способ – представление объекта как хеш-таблицы и хранение его свойств в качестве ее элементов. В этом случае мы можем получить доступ к конкретным свойствам с помощью `gethash`:

```
(gethash 'color obj)
```

Поскольку функции – это объекты данных, мы также можем хранить их как свойства. Это означает, что мы также можем хранить и методы.

Чтобы вызвать определенный метод объекта, необходимо применить `funcall` к свойству с именем метода:

```
(funcall (gethash 'move obj) obj 10)
```

Используя такой подход, можно определить синтаксис передачи сообщений в стиле Smalltalk:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

Теперь, чтобы приказать объекту передвинуться на 10, мы можем сказать:

```
(tell obj 'move 10)
```

По сути, единственный ингредиент, которого не хватает голому Лиспу, – это поддержка наследования. Мы можем реализовать простейший ее вариант с помощью определения рекурсивной версии `gethash`, как показано на рис. 17.1. (Имя `rget` – это сокращение от «recursive get», рекурсивное получение.) Теперь с помощью восьми строк кода мы получаем все три вышеперечисленные элемента объектной ориентированности.

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

***Рис. 17.1.** Наследование*

Давайте опробуем этот код в деле. Применим его к рассмотренному ранее примеру, создав два объекта, один из которых является потомком другого:

```
> (setf circle-class                (make-hash-table)
        our-circle                  (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

Объект `circle-class` будет содержать метод `area` для всех кругов. Это будет функция одного аргумента – объекта, которому посылается сообщение:

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

Теперь можно запросить площадь `our-circle`. Она будет вычисляться в соответствии с только что определенным для класса методом: `rget` считывает свойство, а `tell` применяет метод:

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

Прежде чем браться за усовершенствование данной программы, разберемся с тем, что мы уже сделали. Те самые восемь строчек кода превратили голую версию Лиспа без CLOS в объектно-ориентированный язык. Как нам удалось решить столь непростую задачу? Должно быть, тут скрыт какой-то трюк: реализовать объектную ориентированность восемью строчками кода!

Да, трюк здесь определенно присутствует, но он скрыт не в нашей программе. Дело в том, что Лисп в некотором смысле уже был объектно-ориентированным или скорее даже более общим языком. Все, что нам понадобилось, – это добавить новый фасад вокруг абстракций, которые в нем уже имелись.

17.2. Множественное наследование

Пока что мы умеем осуществлять лишь одиночное наследование, при котором объект может иметь только одного родителя. Но можно реализовать и множественное наследование, сделав `parent` списком и переопределив `rget`, как показано на рис. 17.2.

```
(defun rget (prop obj)
  (dolist (c (precedence obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse
                           (gethash :parents x)))))
    (delete-duplicates (traverse obj))))
```

***Рис. 17.2.** Множественное наследование*

При одиночном наследовании, если мы хотим получить некоторое свойство объекта, нам достаточно рекурсивно пройтись по всем его предшественникам. Если сам объект не содержит достаточной информации об интересующем нас свойстве, мы переходим к его родителю, и т. д. При

подобном поиске в случае множественного наследования наша работа несколько усложняется, потому что предшественники объекта образуют не простое дерево, а граф, и обычный поиск в глубину здесь не поможет. Множественное наследование позволяет реализовать, например, такую иерархию, как на рис. 17.3: к `a` мы можем спуститься от `b` и `c`, а к ним – от `d`. Поиск в глубину (здесь скорее в высоту) даст следующий порядок обхода: `a`, `b`, `d`, `c`, `d`. Если желаемое свойство имеется в `d` и `c`, мы получим значение из `d`, а не из `c`, что не будет соответствовать правилу, по которому значение из подкласса приоритетнее, чем из его родителей.

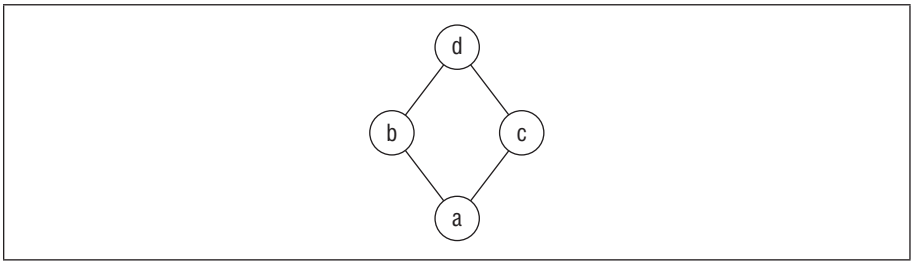

Рис. 17.3. *Несколько путей к одному суперклассу*

Чтобы соблюсти это правило, нам необходимо четко следить, чтобы никакой объект не обрабатывался раньше всех его потомков. В нашем случае порядок обхода должен быть таким: `a`, `b`, `c`, `d`. Как это гарантировать? Простейший способ – собрать список из объекта и всех его предков, следующих в правильном порядке, а затем проверять по очереди каждый элемент этого списка.

Такой список возвращает функция `precedence`. Она начинается с вызова `traverse`, выполняющего поиск в глубину. Если несколько объектов имеют общего предка, то он встретится в списке несколько раз. Если из всех наборов повторений мы сохраним только последний, то в результате получится список предшествования в естественном для CLOS порядке. (Удаление всех повторений, кроме последнего, соответствует правилу 3 на стр. 192.) Такое поведение закреплено за встроенной в Common Lisp функцией `delete-duplicates`. Поэтому если мы передадим ей результат поиска в глубину, она вернет корректный список предшествования. После того как этот список создан, `rget` ищет в нем первый объект с заданным свойством.

Используя идею предшествования, мы можем сказать, например, что патриотичный негодяй является прежде всего негодяем и лишь потом патриотом:[1]

[1] То есть служит (serves) сначала себе (self) и лишь затем стране (country), что и продемонстрировано в данном примере. – *Прим. перев.*

```
> (setf scoundrel           (make-hash-table)
        patriot             (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash :parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

На этом этапе мы имеем очень мощную, но уродливую и неэффективную программу. Пора приступать ко второму этапу – приведению наброска программы к виду, пригодному для использования.

17.3. Определение объектов

Во-первых, нам нужен нормальный способ создания объектов, скрывающий их истинное устройство. Если мы определим функцию для построения объектов, одним вызовом которой можно создать объект и определить его предков, то сможем строить список предшествования для объекта лишь однажды – при его создании, избегая тем самым повторения этой дорогостоящей операции каждый раз, когда нужно найти свойство или метод.

Если мы собираемся хранить список предшествования, вместо того чтобы каждый раз его воссоздавать, нужно учесть возможность его устаревания. Нашей стратегией будет хранение списка всех существующих объектов, а при модификации родителей какого-либо из них мы переопределим список предшествования всех объектов, на которые это влияет. Да, на перестройку списка будут затрачиваться некоторые ресурсы, но мы все равно останемся в выигрыше, так как обращение к свойствам – намного более частая операция, чем добавление новых предков. Кроме того, это нисколько не уменьшит гибкость нашей программы; мы просто переносим затраты с часто используемой операции на редко используемую.

На рис. 17.4. показан новый код.˚ Глобальная переменная `*objs*` является списком всех объектов. Функция `parents` получает предков заданного объекта; `(setf parents)` не только назначает новых предков, но и вызывает `make-precedence` для перестройки любого списка предшествования, который также мог поменяться. Списки, как и прежде, строятся с помощью `precedence`.

Теперь для создания объектов вместо `make-hash-table` пользователи могут вызывать `obj`, которая создает новый объект и определяет всех его предков за один шаг. Мы также переопределили `rget`, чтобы воспользоваться преимуществом хранимых списков предшествования.

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash ipreclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

***Рис. 17.4.** Создание объектов*

17.4. Функциональный синтаксис

Усовершенствовать стоит и синтаксис передачи сообщений. Использование `tell` не только загромождает запись, но и лишает нас возможности читать программу в обычной для Лиспа префиксной нотации:

```
(tell (tell obj 'find-owner) 'find-owner)
```

Мы можем избавиться от `tell`, определяя имена свойств как функции. Определим для этого макрос `defprop` (рис. 17.5). Необязательный аргумент `meth?`, будучи истинным, сигнализирует, что свойство должно считаться методом. В противном случае оно будет считаться слотом, и значение, получаемое `rget`, будет просто возвращаться.

Теперь, когда мы определили имя свойства,

```
(defprop find-owner t)
```

можно сослаться на него через вызов функции, и наш код снова станет читаться как Лисп:

```
(find-owner (find-owner obj))
```

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for "A." name obj))))
```

Рис. 17.5. Функциональный синтаксис

Теперь наш предыдущий пример станет более читаемым:

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot)   'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5. Определение методов

До сих пор мы определяли методы следующим образом:

```
(defprop area t)

(setf circle-class (obj))

(setf (aref circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

Вызывая первый метод из `cdr` списка предшествования, мы можем получить внутри метода эффект встроенного оператора `call-next-method`. К примеру, если мы захотим печатать что-либо во время вычисления площади круга, мы скажем:

```
(setf grumpy-circle (obj circle-class))

(setf (area grumpy-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
```

```
        (funcall (some #'(lambda (x) (gethash 'area x))
                       (cdr (gethash :preclist c)))
                 c)))
```

Данный `funcall` эквивалентен `call-next-method`, но он открывает больше своего внутреннего устройства, чем следовало бы.

Макрос `defmeth` на рис. 17.6 предоставляет более удобный способ создания методов, а также облегчает вызов внутри них следующего метода.

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             (labels ((next () (get-next ,gobj ',name)))
               #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

***Рис. 17.6.** Определение методов*

Вызов `defmeth` раскрывается в `setf`, внутри которого находится `labels`-выражение, задающее функцию `next` для получения следующего метода. Она работает наподобие `next-method-p` (стр. 182), но возвращает объект, который может быть вызван, то есть служит еще и как `call-next-method`.° Теперь предыдущие два метода можно определить следующим образом:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))

(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

Обратите внимание, каким образом в определении `defmeth` используется захват символов. Тело метода вставляется в контекст, в котором локально определена функция `next`.

17.6. Экземпляры

До сих пор мы не делали различий между классами и экземплярами, используя для них общий термин: *объект*. Конечно, такая унификация дает определенную гибкость и удобство, но она крайне неэффективна. В большинстве объектно-ориентированных приложений граф наследования тяготеет к низу. Например, в задаче симуляции трафика типы транспортных средств представляются менее чем десятью классами, а вот объектов, представляющих отдельные автомобили, сотни. Посколь-

ку у всех них будут одинаковые списки предшествования, создавать и хранить их по отдельности – пустая трата ресурсов времени и памяти.

На рис. 17.7 представлен макрос `inst`, производящий экземпляры. Экземпляры подобны объектам (которые мы теперь можем называть классами), но имеют лишь одного предка и не содержат списка предшествования. Кроме того, они не включаются в список `*objs*`. Теперь наш предыдущий пример может быть несколько изменен:

```
(setf grumpy-circle (inst circle-class))
```

Поскольку некоторые объекты отныне не содержат списков предшествования, мы переопределили `rget` и `get-next`, чтобы они могли сразу получать доступ к единственному предку. Прирост производительности никак не сказался на гибкости. Мы можем делать с экземплярами все, что могли делать с любыми объектами, в том числе создавать их экземпляры и переназначать предков. В последнем случае `(setf parents)` будет эффективно преобразовывать объект в «класс».

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
  obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
        (multiple-value-bind (val in) (gethash prop obj)
          (if in
              (values val in)
              (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
        (get-next (gethash obj :parents) name))))
```

***Рис. 17.7.** Создание экземпляров*

17.7. Новая реализация

Пока что ни одно из наших улучшений не сказалось отрицательно на гибкости программы. Обычно на последних этапах разработки Лисп-программа жертвует частью своей гибкости ради эффективности. И наша

программа не исключение. Представив все объекты в виде хеш-таблиц, мы не только получаем избыточную гибкость, но платим за это слишком высокую цену. В этом разделе мы перепишем нашу программу, чтобы представить объекты в виде простых векторов.

Да, мы отказываемся от возможности добавлять новые свойства на лету – до сих пор мы могли определить новое свойство любого объекта, просто сославшись на него. Но мы по-прежнему сможем делать это с помощью переопределения объектов. При создании класса нам потребуется передать ему список новых свойств, а при создании экземпляров они будут получать свойства через наследование.

В предыдущей реализации, по сути, не было разделения между классами и экземплярами. Экземпляром считался класс, имеющий лишь одного предка, и переназначение предков превращало экземпляр в класс. В новой реализации у нас будет полноценное деление между этими двумя типами объектов, но мы больше не сможем преобразовывать экземпляры в классы.

Код на рис. 17.8–17.10 полностью представляет новую реализацию. Рисунок 17.8 содержит новые операторы для определения классов и экземпляров. Классы и экземпляры представляются в виде векторов. Первые три элемента содержат информацию, необходимую самой программе, а первые три макроса на рис. 17.8 позволяют ссылаться на нее:

1. Поле `parents` аналогично записи `:parents` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список родительских классов, для экземпляра – один родительский класс.
2. Поле `layout` (размещение) будет содержать вектор с именами свойств, который определяет их размещение в векторе самого класса или экземпляра, начиная с его четвертого элемента[1].
3. Поле `preclist` аналогично записи `:preclist` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список предшествования, для экземпляра – `nil`.

Так как эти операторы – макросы, то они могут быть первыми аргументами `setf` (см. раздел 10.6).

Макрос `class` предназначен для создания классов. Он принимает необязательный список суперклассов, за которым следуют ноль или более имен свойств. Макрос возвращает объект, представляющий класс. В новом классе будут объединены его собственные свойства и свойства, унаследованные от суперклассов.

```
> (setf *print-array* nil
        geom-class   (class nil area)
        circle-class (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

[1] Первые 3 элемента одинаковы для всех объектов. Это родители, собственно `layout` и список предшествования. – *Прим. перев.*

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
                          :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

***Рис. 17.8.** Реализация с помощью векторов: создание объектов*

Здесь мы создали два класса: `geom-class` не имеет суперклассов и содержит только одно свойство `area`; `circle-class` является подклассом `geom-class` и имеет дополнительное свойство `radius`.[1] Функция `layout` позволяет увидеть имена последних двух из пяти его полей:[2]

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

[1] При отображении классов `*print-array*` должен быть `nil`. Первый элемент в списке `preclist` любого класса сам является классом, поэтому попытка отображения такой структуры приведет к попаданию в бесконечный цикл.

[2] Вектор приводится к списку для просмотра его содержимого. Если `*print-array*` установлен в `nil`, содержимое вектора не отображается.

Макрос class – это просто интерфейс к функции class-fn, которая выполняет основную работу. Она вызывает inherit-props для сборки списка свойств всех предков нового объекта, создает вектор необходимой длины и устанавливает три первых его элемента. (preclist строится с помощью функции precedence, которую мы практически не изменили.) Остальные поля класса устанавливаются в :nil, указывая на то, что они не инициализированы. Чтобы получить свойство area класса circle-class, мы можем сказать:

```
> (svref circle-class
         (+ (position 'area (layout circle-class)) 3))
:NIL
```

Позже мы определим функции доступа, которые будут делать это автоматически.

Наконец, функция inst используется для создания экземпляров. Она не обязана быть макросом, так как принимает лишь один аргумент:

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

Полезно сравнить inst и class-fn, которые выполняют похожие задачи. Так как экземпляры имеют лишь одного родителя, то нет необходимости определять, какие свойства наследуются. Экземпляр может просто скопировать размещение родительского класса. Кроме того, нет необходимости строить список предшествования, поскольку у экземпляра его попросту нет. Таким образом, создание экземпляров намного быстрее создания классов. Но так и должно быть, ведь, как правило, новые экземпляры создаются намного чаще, чем новые классы.

Теперь, чтобы построить иерархию классов и экземпляров, нам потребуются функции чтения и записи свойств. Первой функцией на рис. 17.9 является новая версия rget. По форме она напоминает rget с рис. 17.7. Она имеет две ветки – для классов и экземпляров соответственно.

1. Если объект является классом, мы обходим его список предшествования до тех пор, пока не найдем объект, имеющий значение искомого свойства, не равное :nil. Если такой объект не найден, возвращаем :nil.
2. Если объект является экземпляром, то ищем лишь локально, среди его свойств, а если не находим, то рекурсивно вызываем rget.

У rget появился новый третий аргумент – next?, предназначение которого будет объяснено позже. Сейчас достаточно сказать, что если он nil, то rget ведет себя как обычно.

Функция lookup и обратная ей играют роль gethash в старой версии rget. Они ищут в объекте свойство с заданным именем или устанавливают новое. Этот вызов функции эквивалентен предыдущей версии с svref:

```
> (lookup 'area circle-class)
:NIL
```

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))

(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

Рис. 17.9. *Реализация с помощью векторов: доступ*

Поскольку `setf` для `lookup` также определено, мы можем определить метод `area` для `circle-class`:

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

В этой программе, как и в ее предыдущей версии, нет четкого разделения между классами и экземплярами, а «метод» – это просто функция в поле объекта. Но вскоре это будет скрыто за более удобным фасадом.

Рисунок 17.10 содержит оставшуюся часть нашей новой реализации.[1] Этот код не добавляет программе никаких новых возможностей, но делает ее более простой в использовании. Макрос `defprop`, по сути, остался тем же, просто теперь он вызывает `lookup` вместо `gethash`. Как и раньше, он позволяет ссылаться на свойства в функциональном стиле:

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
```

[1] Данный листинг содержит исправление авторской опечатки, которая была обнаружена Тимом Мензисом (Tim Menzies) и опубликована в новостной группе comp.lang.lisp (19.12.2010). Эта опечатка не внесена автором в официальный список опечаток книги. – *Прим. перев.*

```
> (setf (radius our-circle) 2)
2
```

Если необязательный второй аргумент макроса `defprop` истинен, его вызов раскрывается в `run-methods`, который сохранился практически без изменений.

Наконец, функция `defmeth` предоставляет удобный способ определения методов. В этой версии есть три новых момента: она неявно вызывает `defprop`, использует `lookup` вместо `gethash` и вызывает `rget` вместо `get-next` (стр. 282) для получения следующего метода. Теперь мы видим причину добавления необязательного аргумента в `rget`: эта функция так похожа на `get-next`, что мы можем объединить их в одну за счет добавления дополнительного аргумента. Если этот аргумент истинен, `rget` выполняет роль `get-next`.

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ',name ,gobj t)))
               #'(lambda ,parms ,@body))))))
```

Рис. 17.10. *Реализация с помощью векторов: макросы интерфейса*

Теперь мы можем достичь того же эффекта, что и в предыдущем примере, но с помощью существенно более понятного кода:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

Обратите внимание, что вместо вызова `rget` мы можем сразу вызвать `radius`, так как она была определена как функция при вызове `defprop`.

Благодаря неявному вызову `defprop` внутри `defmeth` мы можем аналогичным образом вызвать `area`, чтобы найти площадь `our-circle`:

```
> (area our-circle)
12.566370614359173
```

17.8. Анализ

Теперь у нас есть встроенный язык, приспособленный для написания реальных объектно-ориентированных приложений. Он простой, но для своего размера достаточно мощный. Кроме того, он будет достаточно быстрым для типичных приложений, в которых операции с экземплярами обычно выполняются намного чаще, чем с классами. Основной задачей усовершенствования кода было именно повышение производительности при работе с экземплярами.

В нашей программе создание новых классов происходит медленно и производит большое количество мусора. Но если классы не создаются в те моменты, когда критична скорость, это вполне приемлемо. Что действительно должно происходить быстро, так это доступ и создание экземпляров. В плане скорости доступа к экземплярам наша программа достигла предела, и дальнейшие усовершенствования могут быть связаны с оптимизацией при компиляции.° То же самое касается и создания экземпляров. И никакие операции с экземплярами не требуют выделения памяти, за исключением, разумеется, создания вектора, представляющего сам экземпляр. Сейчас такие векторы размещаются в памяти динамически, что вполне естественно, но при желании можно избежать динамического выделения памяти, применив стратегию, описанную в разделе 13.4.

Наш встроенный язык является характерным примером программирования на Лиспе. Уже тот факт, что он встроенный, говорит об этом. Но показательно и то, как Лисп позволил нам развивать программу от небольшой ограниченной версии к мощной, но неэффективной, а затем к эффективной, но слегка ограниченной.

Репутация Лиспа как медленного языка происходит не из его природы (еще с 1980-х годов компиляторы Лиспа способны генерировать не менее быстрый код, чем компиляторы C), а из того, что многие программисты останавливаются на втором этапе развития программы. Как писал Ричард Гэбриел:

> «В Лиспе очень легко написать программу с плохой производительностью; в C это практически невозможно».°

Да, это так, но это высказывание может пониматься как за, так и против Лиспа:

1. Приобретая гибкость ценой скорости, вы облегчаете процесс написания программ в Лиспе; в C вы не имеете такой возможности.
2. Если не оптимизировать ваш Лисп-код, очень легко получить медленную программу.

Какая из интерпретаций описывает вашу программу, зависит всецело от вас. Но в Лиспе у вас как минимум есть возможность пожертвовать скоростью выполнения ради экономии вашего времени на ранних этапах создания программы. Для чего наш пример точно *не* подходит, так это для демонстрации модели работы CLOS (за исключением, пожалуй, прояснения механизма работы `call-next-method`). Да и как можно говорить о сходстве слонопотамного CLOS и нашего 70-строчного комарика. Действительно, различия между этими двумя программами более показательны, чем сходства. Во-первых, мы увидели, насколько широким понятием может быть объектная ориентированность. Наша программа будет помощнее, чем некоторые другие реализации объектно-ориентированного программирования, но тем не менее она обладает лишь частью мощи CLOS.

Наша реализация отличается от CLOS тем, что методы являются методами *определенного* объекта. Такая концепция методов приравнивает их к функциям, поведение которых управляется их первым аргументом. Обобщенные функции CLOS могут управляться любыми из своих аргументов, а методами считаются компоненты обобщенной функции. Если они будут специфицированы только по первому аргументу, то можно создать иллюзию, что методы принадлежат определенным классам. Но представление о CLOS в терминах модели передачи сообщений лишь запутает вас, так как CLOS значительно превосходит ее.

Отметим один из недостатков CLOS: ее огромный размер и проработка скрывают то, насколько объектно-ориентированное программирование является лишь перефразировкой Лиспа. Пример, приведенный в данной главе, по крайней мере, проясняет этот момент. Если бы нас удовлетворяла старая модель передачи сообщений, нам бы хватило немногим более страницы кода для ее реализации. Объектно-ориентированное программирование – лишь одна из вещей, на которые способен Лисп. А теперь более интересный вопрос: на что еще он способен?