Когда использовать макросы

Как определить должно ли данное действие быть реализовано в виде функции или в виде макроса? Чаще всего существует чёткое различие между случаями в которых лучше использовать макросы и случаями в которых они не нужны. По умолчанию мы должны использовать функции: не элегантно использовать макрос там, где можно использовать функцию. Мы должны обращаться к макросам только в тех случаях, в которых они дают нам специфические преимущества.

Когда же макросы дают преимущества? Это предмет обсуждения данной главы. Обычно вопрос не столько в преимуществе, сколько в необходимости. Большую часть вещей, которую мы делаем с помощью макросов невозможно сделать с помощью функций. Раздел 8.1 перечисляет виды операторов, которые могут быть реализованы только как макросы. Однако, существует небольшой (но интересный) класс пограничных случаев, где оператор справедливо может быть написан и как функция, и как макрос. Для таких ситуаций, в разделе 8.2 приводятся аргументы за и против макросов. В итоге, рассмотрев то, что можно сделать с помощью макросов, мы обратимся в разделе 8.3 к взаимосвязанному вопросу: что люди делают с помощью них?

Когда нет другого выбора

Общий принцип правильного проектирования таков, что если вы находите похожий код в нескольких местах программы вы должны написать подпрограмму и заменить похожие участки вызовом этой подпрограммы. Когда мы применяем этот принцип к Lisp программам, мы должны решить, должна ли быть «подпрограмма» функцией или макросом.

В некоторых случаях проще принять решение писать макрос вместо функции, так как только макрос может сделать то, что

нужно. Функция похожая на 1+ потенциально могла быть написана как функция и как макрос:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

Но WHILE, из секции 7.3, может быть определена только как макрос:

```
(defmacro while (test &body body)
    `(do ()
         ((not ,test))
         ,@body))
```

Не существует способа воспроизвести поведение этого макроса при помощи функции. Определение `while` вклеивает выражения переданные как `body` в `do`, где они вычисляются только если `test` выражение возвращает `nil`. Ни одна функция не может сделать этого; в вызове функции все аргументы вычисляются ещё до того как вызовется сама функция.

Когда вам понадобился макрос, что вы хотите получить от него? Макросы могут делать две вещи, которые не могут делать функции: они могут контролировать (или предотвратить) вычисление своих аргументов, а также они разворачиваются непосредственно в вызывающий контекст. Любое приложение, нуждающееся в макросах, в конечном счёте испытывает потребность в одном или обоих этих свойствах.

Неформальное объяснение, что «макросы не вычисляют свои аргументы» слегка неверно. Точнее будет сказать, что макросы *контролируют* вычисление своих аргументов в вызове макроса. В зависимости от того, куда аргумент помещён в развёртке макроса, он может быть вычислен однажды, много раз или совсем никогда. Макросы производят такой контроль четырьмя способами:

1) *Трансформация.* Макрос `setf` один из класса макросов, ко-

торые разделяют свои аргументы на части до их вычисления. Для встроенной функции доступа всегда есть парная, задача которой установить то, что вернула функция доступа. Обратная функция `car` это `rplaca`, `cdr` — `rplacd` и так далее. С помощью `setf` мы можем обращаться к таким функциям доступа так, как будто они являются переменными, например `(setf (car x) 'a)`, который раскрывается в `(progn (rplaca x 'a) 'a)`.

Чтобы осуществить такой трюк, `setf` должна заглянуть внутрь своего первого аргумента. Чтобы понять, что в текущем случае нужна `rplaca`, `setf` должна увидеть, что первый аргумент начинается с `car`. Таким образом `setf` и любой другой оператор, преобразующий свои аргументы, должны быть реализованы в виде макросов.

2) *Связывание.* Лексические переменные должны находиться непосредственно в коде. Первый аргумент для `setq` например не вычисляется, поэтому всё что построено на основе `setq` должно раскрываться в `setq`, нежели быть функцией, которой её вызывает. Аналогично для операторов таких как `let`, чьи аргументы присутствуют как параметры в `lambda` выражении, для макросов как `do`, раскрывающихся в `lets` и так далее. Любой новый оператор, изменяющий лексические привязки своих аргументов должен быть написан как макрос.

3) *Вычисление при определённом условии.* Все аргументы функции вычисляются. В конструкциях наподобие `when` мы хотим, чтобы часть аргументов вычислялась только при определённых условиях. Такая гибкость возможна только с помощью макросов.

4) *Многократное вычисление.* Все аргументы функций вычисляются, и вычисляются они только один раз. Нам необходим

макрос для определения конструкции наподобие `do`, где нужные нам аргументы вычисляются многократно.

Также существует несколько способов воспользоваться свойством раскрытия макроса в коде. Важно обратить внимание на то, что раскрытие таким образом появляется в лексическом контексте вызова макроса, так как два из трёх случаев использования макросов полагается на этот факт. Вот они:

5) *Использование контекста вызова.* Макрос может создать раскрытие, содержащее переменную, чья привязка определяется контекстом вызова. Поведение следующего макроса:

```
(defmacro foo (x)
    '(+ ,x y))
```

зависит от привязки `y` там, где вызывается `foo`.

Такой тип лексического взаимодействия обычно рассматривается скорее как источник вреда, чем пользы. Чаще всего это будет плохим стилем написания подобного макроса. Идеал функционального программирования также применим и к макросам: предпочтительный способ общения с макросом — через его параметры. Действительно, настолько редко требуется использовать контекст вызова, что чаще всего это происходит именно по ошибке. (Смотрите Главу 9.) Из всех макросов в этой книге, только макрос передачи продолжения (continuation) (Глава 20) и некоторые части `ATN` компилятора (Глава 23) используют вызывающее окружение таким образом.

6) *Создание окружения-обёртки.* Макрос также может заставить свои аргументы выполняться в новом лексическом окружении. Классический пример `let`, который может быть реализован как макрос над `lambda` (страница 144). Внутри

тела выражения такого как `(let ((y 2)) (+ x y))`, `y` будет ссылаться на новую переменную.

7) *Сохранение вызовов функций.* Третье последствие встраивания раскрытия макроса заключается в том, что в скомпилированном коде нет издержек на вызов макроса. К времени выполнения программы вызов макроса будет заменён его раскрытием. (Тот же принцип справедлив и для функции объявленной `inline`.)

Особенно важно, что случаи 5 и 6, при их неумышленном использовании, создают проблему захвата переменной — самое неприятное, чего стоит опасаться создателю макроса. Захват переменной обсуждается в главе 9.

Вместо семи способов использования макроса, правильнее было бы сказать шесть с половиной. В идеальном мире, все компиляторы Common Lisp будут следовать объявлениям `inline` и экономия на вызовах функций будет задачей встраиваемых функций, не макросов. Идеальный мир оставлен читателю в качестве упражнения.

Макрос или функция?

Предыдущий раздел касался простых случаев. Любой оператор, которому необходим доступ к параметрам до их вычисления должен быть написан как макрос, потому что других вариантов нет. Что же насчёт тех операторов, которые могут быть написаны и тем и другим способом? К примеру возьмём оператор `avg`, который возвращает среднее от всех аргументов. Он может быть определён как функция

```
(defun avg (&rest args)
    (/ (apply #'+ args) (length args)))
```

но также его вполне обоснованно можно реализовать как макрос

```
(defmacro avg (&rest args)
    `(/ (+ args) ,(length args)))
```

потому, что версия с функцией повлечёт ненужные вызовы функции `length` при каждом вызове `avg`. Во время компиляции мы можем не знать значения аргументов, но мы знаем их количество, так что вызов `length` с тем же успехом может быть сделан на этом этапе. Ниже приведено несколько размышлений о вещах, которые нужно брать в расчёт когда мы сталкиваемся с таким выбором:

The Pros

1) *Вычисление на этапе компиляции*. Вызов макроса разбит на два вычисления: когда макрос раскрывается и когда раскрытие вычисляется. Все раскрытия в Lisp программе происходят при компиляции и каждый бит, вычисленный во время компиляции, это бит, который не будет тормозить программу во время её работы. Если оператор может быть написан так, что часть своей работы будет выполняться на этапе раскрытия макроса, тогда эффективнее реализовывать его макросом, потому, что ту работу, с которой не справился умный компилятор должна будет делать функция во время выполнения. Глава 13 описывает макросы аналогичные `avg`, которые делают часть своей работы при раскрытии.

2) *Интеграция с Lisp*. Иногда, использование макросов вместо функций делает программу более интегрированной с Lisp. Вместо написания программы для решения определённой проблемы, вы можете использовать макрос для преобразования проблемы в ту, которую Lisp уже умеет решать. Такой подход, когда возможен, обычно сокращает код программы, а также делаёт её более эффективной: сокращает код, так как Lisp делает часть работы за вас, делает эффективнее, так как промышленные Lisp системы в большинстве выжимают всё до

последнего в плане эффективности по сравнению с пользовательскими программами. Это преимущество появляется чаще всего во встроенных языках, о которых пойдёт речь начиная с главы 19.

3) *Экономия вызовов функций*. Вызов макроса раскрывается непосредственно в код, где он встречается. Так что если вы какую-то часть кода как макрос, вы сэкономите вызов функции на каждом вызове. В ранних диалектах Lisp, программисты пользовались этим преимуществом макросов для экономии вызовов функции во время выполнения. В Common Lisp, эта работа перекладывается на функции объявленные как `inline`.

   Объявляя функцию как `inline`, вы просите компилятор скомпилировать её непосредственно в вызываемый код, также как макрос. Однако, здесь есть расхождение между теорией и практикой; CLTL2 (стр. **??**) говорит нам «компилятор волен игнорировать декларацию» и некоторые Common Lisp компиляторы это и делают. Поэтому до сих пор может быть оправдано использование макросов для сокращения количества вызовов функций, если вы вынуждены использовать такой компилятор.

В некоторых случаях комбинированное преимущество от эффективности и тесной интеграции с Lisp может дать серьёзный повод для использования макросов. В компиляторе запросов главы 19, количество вычислений, которые можно перераспределить на этап компиляции настолько большое, что оправдывает превращение всей программы в один гигантский макрос. Сделанное для увеличения скорости, изменение также делает код теснее интегрированным с Lisp: в новой версии проще использовать Lisp выражения — например арифметические выражения — внутри запроса.

The Cons

4) *Функции — данные*, в то время как макросы больше похожи на инструкции компилятору. Функции могут быть переданы как аргументы (например в `apply`), возвращены другими функциями, либо сохранены в структурах. Ни что из перечисленного не возможно сделать макросами.

   В некоторых случаях, вы можете достичь того, что хотите обернув вызов макроса ламбда функцией. Это работает, например, если вы хотите вызвать `apply` или `funcall` над конкретным макросом:

   ```
   > (funcall #'(lambda (x y) (avg x y)) 1 3)
   2
   ```

   Однако, это неудобно. К тому же это не всегда сработает: даже если также как `avg`, макрос имеет `&rest` параметр, не существует способа передать изменяемое количество аргументов.

5) *Ясность исходного кода.* Макро объявления могут быть тяжелее в чтении, чем эквивалентные объявления функций. Так что если написание чего-то как макрос сделает программу незначительно лучше, лучше использовать функцию.

6) *Ясность во время выполнения.* Макросы иногда труднее отлаживать чем функции. Если вы ловите ошибку времени выполнения в коде, содержащем большое количество вызовов макросов, код, который вы видете в backtrace, может состоять из раскрытий всех этих макросов, и быть мало похожим на исходный код который вы написали.

   И по той же причине, что макросы исчезают после раскрытия, их вызовы нельзя учесть. Обычно вы не можете использовать `trace`, чтобы увидеть как вызывался макрос. Если даже это сработало, `trace` покажет вызов функции раскрытия макроса, но не сам вызов макроса.

7) *Рекурсия*. Использование рекурсии в макросах не такое простое как в функциях. Несмотря на то, что функция раскрытия макроса может быть рекурсивной, само раскрытие таковым быть не может. Раздел 10.4 касается вопроса рекурсии в макросах.

Все эти рассуждения должны быть сбалансированы относительно друг друга при решении о использовании макросов. Только опыт может подсказать, какое из них будет доминирующим. Однако, примеры макросов, появляющиеся в последующих главах покрывают большую часть ситуаций в которых макросы полезны. Если проектируемый макрос аналогичен одному из приведённых здесь, тогда в большой вероятностью реализовывать его будет безопасно.

В заключении, нужно отметить, что ясность (пункт 6) редко становится проблемой. Отладка кода, использующего большое количество макросов будет не такой сложной как может показаться. Если объявления макросов занимают несколько сотен строк, то их было бы неприятно отлаживать во время выполнения. Но утилиты, в конечном счёте, стремятся к написанию небольших, надёжных слоев. В общем случае их объявления занимают не больше 15 строк. Так что даже на случай сосредоточенного обдумывания логов обратной трассировки, таки макросы не сильно затуманят ваш взгляд.

Применения макросов

Зная возможности макросов следующий вопрос, который требует ответа, это: в каком типе приложений мы можем их использовать? Наиболее общий ответом использования макросов можно считать их применение для синтаксических преобразований. Это не наводит на мысль, что область применения макросов ограничена.

Так как Lisp программы составлены из списков[1], которые есть Lisp структуры данных, «синтаксическое преобразование» может означать по истине большие возможности. В главах 19-24 представлены программы, чья задача может быть описана как синтаксическое преобразование и которые в сущности все являются макросами.

Область применения макросов занимает всё пространство между небольшими макросами общего назначения, таких как `while` и большими, специализированными под задачу макросами, которые приведены в последующих главах. С одной стороны находятся *утилиты*, макросы напоминающие те, которые встроены в каждый Lisp. Они обыкновенно небольшие, наиболее общие и написаны вне какого-либо контекста. Однако, вы можете писать утилиты и для специфического класса программ, когда например у вас есть коллекция макросов для использования в графической программе, они становятся очень похожими на язык для графики. С другой стороны, макросы помогают вам писать целые программы на языке, отличном от Lisp. Макросы используемые таким образом называются реализацией *встроенного языка* (embedded languages).

Утилиты — первый результат стиля разработки снизу-вверх. Даже если программа слишком мала для построения из слоёв, она всё же может получить преимущество от добавления нижнего слоя, самого Lisp. Утилита `nil!`, устанавливающая аргумент в `nil`, не может быть по-другому реализована, кроме как макросом:

```
(defmacro nil! (x)
    `(setf ,x nil))
```

Посмотрев на `nil!`, у кто-нибудь может захочет сказать, что он вообще то ничего не делает, кроме сокращения времени набора. Это правда, но всё что делают макросы в действительности сокращение набора машинного языка. Значение утилит не должно быть недооценено, потому что эффект кумулятивный: несколько слоёв

[1] В смысле являются входными данными для компилятора. Функции более не представляют из себя списки, как было в ранних диалектах.

специальных макросов могут стать различием между элегантной программой и никуда не годной.

Большинство утилит — это материализованные шаблоны. Когда вы замечает шаблон в своём коде, подумайте над превращением его в утилиту. Шаблоны — это как раз то, в чём компьютеры очень хороши. Зачем вы должны утруждать себя их повторять, когда вы можете заставить делать это программу? Представьте, что при написании некоторой программы вы обнаруживаете за собой, что во многих местах используете `do` циклы одинакового вида:

```
(do ()
    ((not <condition>))
    . <body of code>)
```

Когда вы видите повторяющийся шаблон в коде, у него обычно есть имя. Имя этого шаблона — `while`. Если мы хотим сделать из него новую утилиту, мы должны будем использовать макрос, так как нам необходимо вычисление по условию и повторяющееся вычисление. Если мы определим `while` используя определения со страницы **??**:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
       ,@body))
```

тогда мы можем переписать все шаблоны как:

```
(while <condition>
    . <body of code>)
```

Сделав это мы получим более лаконичный код, точнее описывающий то, что он делает.

Возможность преобразовывать аргументы делает макросы полезными для написания интерфейсов. Подходящий макрос позволит набирать более короткие выражения, там где потребовалось бы объёмное. Хотя графические интерфейсы снижают необходимость писать таковые для конечных пользователей, программисты

используют такой тип макросов постоянно. Самый простой пример — `defun`, который делает привязку функций по виду похожей на описание функций в таких языках как Pascal и C. В главе 2 упомянуты следующие два выражения, имеющий одинаковый эффект:

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
      #'(lambda (x) (* x 2)))
```

Так что `defun` может быть реализован как макрос, превращающий первое в последнее. Мы можем представить, что он написан следующим образом:

```
(defmacro our-defun (name params &body body)
  `(progn
     (setf (symbol-function ',name)
           #'(lambda ,params (block , name ,@body)))
      ',name))
```

Макросы такие как `while` и `nil!` можно описать как утилиты общего назначения. Любая Lisp программа может их использовать. Но конкретные области как правило могут иметь свои утилиты. Нет причин предполагать, что Lisp база единственный уровень на котором вы можете расширять язык программирования. Если вы пишете, например, CAD программу, лучших результатов можно достичь при написании в два слоя: язык (либо если вы предпочитаете более скромный термин, набор инструментов) для CAD программ и вышестоящий слой — ваша программа.

Lisp размывает границы, которые в других языках воспринимаются как должные. В других языках, существуют концептуальные различия между временем компиляции и работой программы, программой и данными, языком и программой. В Lisp, эти ограничения существуют только как устные соглашения. Не существует разделяющей линии, например, между языком и программой. Вы

```
(defun move-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (incf (obj-x o) dx)
      (incf (obj-y o) dy))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
      (dolist (o objs)
        (setf (obj-dx o) (* (obj-x o) factor)
              (obj-dy o) (* (obj-y o) factor)))
      (multiple-value-bind (xa ya xb yb) (bounds objs)
        (redraw (min x0 xa) (min y0 ya)
                (max x1 xb) (max y1 yb)))))
```

Рис. 1 — Исходные `move` и `scale`

можете провести линию так, чтобы это соответствовало поставленной задаче. Так что это не более чем вопрос терминологии называть ли подложенный слой кода инструментарием или языком. Одно из удобств считать его языком в том, что это подсказывает вам, что его можно расширять так же как вы расширяете Lisp утилитами.

Возьмём к примеру интерактивную графическую программу для редактирования изображений. Для простоты, мы предположим, что единственные объекты, с которыми имеет дело программа — линии, представленные начальной точкой `<x, y>` и вектором `<dx, dy>`. Одной из возможностей, которая такая программа должна будет обладать, это перемещение группы объектов. Это задача функции `move-objs` (Рисунок. 1). Для эффективности, мы не хотим перери-

совывать весь экран после каждой операции, только те части, которые изменились. Следовательно два вызова функции `bounds`, которая возвращает четыре координаты (`min x, min y, max x, max y`) представляющие прямоугольник в котором находятся объекты. Часть отвечающая за операцию заключена между двумя вызовами `bounds`, посредством которых находятся прямоугольные области до смещения и после, и затем перерисовывается затронутая часть изображения.

Функция `scale-objs` предназначена для изменения размеров группы объектов. В виду того, что изменяемая часть может увеличиться или сжаться в зависимости от масштабного коэффициента, эта функция также должна производить все операции между двумя вызовами `bounds`. По мере того как мы будем писать подобные программы, мы увидим больше случаев появления этого шаблона: в функциях поворота, отображения, транспонирования и так далее.

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    `(let ((,gob ,objs))
       (multiple-value-bind (,x0 ,y0 ,x1 ,y1)
                            (bounds ,gob)
         (dolist (,var ,gob) ,@body)
         (multiple-value-bind (xa ya xb yb)
                              (bounds ,gob)
             (redraw (min ,x0 xa) (min ,y0 ya)
                     (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
    (with-redraw (o objs)
        (incf (obj-x o) dx)
        (incf (obj-y o) dy)))

(defun scale-objs (objs dx dy)
    (with-redraw (o objs)
        (setf (obj-dx o) (* (obj-x o) factor)
              (obj-dy o) (* (obj-y o) factor))))
```

Рис. 2 — Выделенные `move` и `scale`

С макросом мы можем абстрагировать подобный код таких функций. Макрос `with-draw` на рисунке 2 является единым каркасом, используемым в функциях. Как результат, они могут быть написаны в четыре строки, как в конце рисунка 2. Уже на этих двух функциях новый макрос показал свою лаконичность. И на сколько более ясные стали две функции после абстрагирования деталей отрисовки.

С одной стороны можно рассматривать `with-redraw` как конструкцию языка для написания интерактивных графических программ. После того как мы разработаем больше подобных макросов, они в действительности начнут быть похожими на язык программирования, а сама наша программа станет более элегантной, и можно подумать, что она была написана на специально созданном для неё языке.

Другим вариантом использования макросов является реализация встроенных языков. Lisp особенно хорош для написания языков программирования, так как Lisp программы могут быть выражены в виде списков, и Lisp обладает встроенным парсером (`read`) и компилятором (`compile`) для такого представления. Чаще даже вам не понадобится вызывать `compile`; вы можете иметь свой встроенный язык, компилируемый неявно, компилированием кода совершающего преобразования (стр. 25).

Встроенный язык этот такой язык, который не написан поверх Lisp, а смешан с ним, так что синтаксис — это смесь из Lisp и конструкций специфического языка. Наивным способом создать такой язык является написание для него интерпретатора (на Lisp). Более хорошим способом, если это возможно, являются преобразования языка: преобразовать каждое выражение в Lisp код, который затем интерпретатор считает и выполнит. Та задача в которой макросы выходят на сцену. Работа макросов в точности работа по преобразованию одного типа выражений в другие, так что они становятся естественным выбором для написания встроенных языков.

В общем случае, чем большая часть встроенного языка может быть реализована с помощью макросов, тем лучше. Во-первых, это уменьшение работы. Если в новом языке есть арифметика, например, вы не сталкиваетесь со сложностями представления и манипулирования числовыми значениями. Если возможности Lisp достаточно для ваших целей, тогда вы можете просто преобразовать ваши арифметические выражения в эквивалентные выражения на Lisp, всё остальное за вас сделает Lisp.

Использование преобразований также как правило сделает ваш встроенные язык быстрее. Интерпретаторы обладают неудобствами связанными с скоростными характеристиками. Например, если код встречается в цикле, то интерпретатор будет совершать работу на каждой итерации, в отличие от скомпилированного когда, в котором она сделана единожды. Встроенный язык имеющий свой интерпретатор в любом случае будет медленным, даже если он сам скомпилирован. Но если выражения нового языка преобразованы в Lisp, тогда окончательный код будет скомпилирован Lisp компилятором. Язык реализованный таким образом не будет иметь издержек интерпретации. Сократив время на написание компилятора для вашего языка, макросы оставят преимущество быстрого выполнения. В действительности, макросы преобразующие новый язык могут рассматриваться как компилятор для него, используюемый для основной части работы компилятор Lisp.

Мы не будем касаться никаких примеров встроенного языка, в виду того, что главы 19-25 полностью посвящены этой теме. Глава 19 связана с темой разницы между интерпретацией и преобразованием встроенного языка, приводит один и тот же язык реализованный двумя способами.

В одной из книг по Common Lisp утверждается, что область применения макросов ограничена, доказывая это тем фактом, что из всех операторов, определённых в CLTL1, меньше 10% занимают макросы. Такое утверждение, равносильно утверждению, что если

наш дом сделан из кирпичей, то наша мебель также должна быть из них сделана. Пропорция макросов в Common Lisp программе будет полностью зависеть от её назначения. Некоторые программы не будет содержать макросов. Некоторые программы будут полностью состоять из них.