Когда использовать макросы

Как мы можем понять когда данная функция должна действительно быть функцией, а не макросом? Почти всегда существует чёткое различие между случаями которые имеют склонность к реализации с помощью макросов и которые такой склонности не имеют. По умолчанию мы должны использовать функции: неопрятно помещать макрос там, где место функции. Мы должны использовать макросы только там, где 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 не может быть осуществлена функциями. Секция 8.1 перечисляет характерные операции, которые могут быть реализованы только как макросы. Однако, существует небольшой (но интересный) класс пограничных классов, где оператор справедливо может быть написан и в виде функции и в виде макроса. Для таких ситуаций, в секции 8.2 приводятся аргументы за и против макросов. В итоге, ознакомившись с тем, с чем могут справиться макросы, мы переходим в секции 8.3 к связанному вопросу (related question): какого рода вещи люди с помощью них делают?

Когда больше ничего не помогает

Общий принцип хорошего дизайна 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 повторить 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 потребовался макрос, что вы хотите получить от него? Макросы могут сделать две вещи, которые не могут сделать функции: они могут контролировать (или предотвратить) вычисление своих аргументов, а также они разворачиваются непосредственно в вызываемый 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 это узнать, 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. Каждый новый оператор, изменяющий лексические привязки своих аргументов должны быть написаны как макросы.

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.* Кроме того факта, что все аргументы функции вычисляется, есть ещё один — они вычисля-

ются только один раз. Нам необходим макрос для определения конструкции `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. Продолжим:

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 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 вычесленный 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 будет более эффективным 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, котороую Lisp уже умеет решать. Такой подход, когда возможен, обычно сделает программу как меньше, так и более эффективнее: меньше, так как Lisp делает часть работы за вас, эффективнее, так как промышленные Lisp системы в большинстве выжимают всё до последнего в плане эффективности по сравнению с пользовательсими программами. Это

преимущество появляется чаще всего в 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 с 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 обхявления 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 этой же причине, так как 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 итоге, 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 размером на 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 применения 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.

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