Искусство метапрограммирования, Часть 1: Введение в метапрограммирование

Источник: [IBM developerWorks Россия](http://www-128.ibm.com/developerworks/ru/library/l-metaprog1/index.html)

Джонатан Бартлет

**Оглавление**

* [Различные применения метапрограммирования](http://www.interface.ru/home.asp?artId=2559#N10058)
* [Основные текстовые макроязыки](http://www.interface.ru/home.asp?artId=2559#N1006A)
* [Чувствительное к языку макропрограммирование с Scheme](http://www.interface.ru/home.asp?artId=2559#N1023C)
* [Резюме](http://www.interface.ru/home.asp?artId=2559#N1034A)

Одной из самых неиспользуемых технологий программирования является написание программ, генерирующих программы или части программ. Узнайте, почему метапрограммирование необходимо, и рассмотрите некоторые его компоненты (текстовые макроязыки, специализированные генераторы кода). Узнайте, как создать генератор кода, и познакомьтесь ближе с чувствительным к языку макропрограммированием в Scheme.

Генерирующие код программы часто называют метапрограммами ; написание этих программ называется метапрограммированием. Создание программ, генерирующих код, имеет многочисленные применения.

Данная статья объясняет, почему вам желательно знать метапрограммирование, и рассматривает некоторые компоненты этого искусства - мы детально рассмотрим текстовые макроязыки, исследуем специализированные генераторы кода и обсудим, как их создать, проанализируем чувствительное к языку макропрограммирование с использованием Scheme.

**Различные применения метапрограммирования**

Во-первых, вы можете писать программы, которые будут предварительно генерировать таблицы данных для использования во время исполнения программы. Например, если вы создаете игру и хотите выполнять быстрый поиск в таблице синусов для всех 8-битных целых чисел, то можете либо вычислять каждый синус и закодировать это таким образом, чтобы программа строила таблицу при запуске, либо написать программу для создания специализированного кода для таблицы перед компиляцией. Хотя для такого маленького набора чисел может иметь смысл создание таблицы во время исполнения, другие подобные задачи могут чрезмерно замедлить время запуска программы. В таких случаях написание программы построения статических данных обычно является наилучшим вариантом.

Во-вторых, если у вас есть большое приложение, в котором множество функций включает длинный стереотипный код, вы можете создать мини-язык, который будет создавать стереотипный код вместо вас и даст вам возможность кодировать только важные части программы. Здесь, если вы сможете, лучше всего абстрагировать стереотипные фрагменты в функцию. Но часто эти фрагменты не столь приятны. Возможно есть список переменных, которые нужно объявить в каждом экземпляре, возможно есть необходимость зарегистрировать обработчики ошибок, возможно существует несколько стереотипных фрагментов, которые должны включать код в определенных обстоятельствах. Все это делает создание простой функции невозможным. Часто в таких ситуациях хорошей идеей является создание мини-языка, что позволит вам работать с таким кодом более простым способом. Этот мини-язык затем конвертируется в исходный код на обычном языке программирования перед компиляцией.

Наконец, многие языки программирования вынуждают вас писать многословные операторы для выполнения простых задач. Генерирующие код программы позволяют вам сократить эти операторы и сохранить много времени, расходуемого на набор кода, что также предохраняет от многих ошибок из-за уменьшения шанса возникновения опечаток.

По мере приобретения языком дополнительных возможностей, генерирующие код программы становятся менее привлекательными. То, что является доступным как стандартная возможность в одном языке, может быть доступно в другом только при использовании генерирующей код программы. Однако, неадекватный дизайн языка является не единственной причиной необходимости в программах, генерирующих код. Важную роль имеет еще и более легкое обслуживание.

**Основные текстовые макроязыки**

Генерирующие код программы дают вам возможность разрабатывать и использовать маленькие, специфические для конкретной области языки, на которых легче писать и поддерживать программы, чем писать их на целевом языке.

Средства для создания этих предметных языков обычно называются макроязыками. В этой статье рассматриваются некоторые типы макроязыков, а также обсуждается, как вы можете использовать их для улучшения вашего кода.

**Препроцессор C (CPP)**

Сначала давайте посмотрим на метапрограммирование, в котором используются текстовые макроязыки. Текстовым макросом является макрос, который непосредственно влияет на текст, написанный на языке программирования, и при этом не знает языка или не имеет отношения к его смыслу. Наиболее широко используемыми текстовыми макросистемами являются препроцессор C и макропроцессор M4.

Если вы работали с языком C, то, возможно, имели дело с макросом#define. Текстовые макрорасширения хотя и не идеальный, но простой способ выполнить метапрограммирование на начальном уровне во многих языках, которые не имеют более развитых возможностей генерирования кода. В листинге 1 приведен пример макроса#define:

**Листинг 1. Простой макрос для перестановки двух значений**

|  |
| --- |
| #define SWAP(a, b, type) { type \_\_tmp\_c; c = b; b = a; a = c; } |

Этот макрос дает вам возможность переставить два значения данного типа. Эту операцию лучше всего делать в макросе по нескольким причинам:

* Вызов функции приводит к большим накладным расходам для простых операций.
* Вы должны были бы передать адреса переменных в функцию вместо их значений (это не так уж и плохо, но передача адресов делает вызов функции менее изящным и не дает компилятору хранить значения в регистрах).
* Вы должны были бы кодировать различные функции для каждого типа переменных, значения которых хотели бы переставлять.

В листинге 2 приведен пример используемого макроса:

**Листинг 2. Использование макроса SWAP:**

|  |
| --- |
| #define SWAP(a, b, type) { type \_\_tmp\_c; c = b; b = a; a = c; }  int main()  {  int a = 3;  int b = 5;  printf("a is %d and b is %d\n", a, b);  SWAP(a, b, int);  printf("a is now %d and b is now %d\n", a, b);  return 0;  } |

Препроцессор C во время исполнения дословно изменяет текстSWAP(a, b, int)на{ int \_\_tmp\_c; \_\_tmp\_c = b; b = a; a = \_\_tmp\_c; }.

Текстовая подстановка - это полезная, но довольно ограниченная возможность. С ней есть следующие проблемы:

* Текстовая подстановка может стать очень запутанной при использовании в комбинации с другими выражениями.
* Препроцессор C разрешает использовать только фиксированное количество аргументов макросов.
* Из-за механизма работы с типами языка C вам часто требуются различные макросы для различных типов аргументов, или, по крайней мере, вы должны передавать типы параметров как аргументы макроса.
* Поскольку выполняется только текстовая подстановка, язык C недостаточно интеллектуален для переименования нашей временной переменной, если она конфликтует с одним из переданных в макрос аргументов. Наш макрос наверняка выдаст ошибку, если ему передать переменную с названием\_\_tmp\_c.

Проблема комбинирования макросов с выражениями делает написание макросов довольно трудным делом. Допустим, вы имеете следующий макрос с названиемMIN, который возвращает меньшее из двух чисел:

**Листинг 3. Макрос, возвращающий минимум из двух значений**

|  |
| --- |
| #define MIN(x, y) ((x) > (y) ? (y) : (x)) |

Возможно вы удивитесь, почему используется так много скобок. Из-за старшинства операторов. Например, если вы запишетеMIN(27, b=32), без этих скобок макрос расширился бы в27 > b = 32 ? b = 32 : 27, что привело бы к ошибке компиляции, поскольку выражение27 > bвыполнилось бы раньше из-за старшинства операций. Если опять поставить скобки, то все будет работать так, как ожидалось.

К сожалению, еще есть и вторая проблема. Любая функция, вызванная как параметр, будет вызываться каждый раз, когда она появляется с правой стороны. Помните, препроцессор C не знает ничего о языке C и только выполняет текстовую подстановку. Следовательно, если вы выполните макросMIN(do\_long\_calc(), do\_long\_calc2()), то он расширится в( (do\_long\_calc()) > (do\_long\_calc2()) ? (do\_long\_calc2()) : (do\_long\_calc())). Его выполнение займет длительное время, поскольку как минимум одно вычисление будет выполнено дважды.

Еще хуже, когда одно из этих вычислений имеет побочный эффект (например, распечатка, изменение глобальной переменной и т.д.), поскольку этот побочный эффект выполнится дважды. Эта проблема "множественных вызовов" может даже повлиять на возврат из макроса ошибочного значения, если одна из функций возвращает различное значение в каждом вызове.

Более подробная информация по макропрограммированию препроцессора C доступна в "Справочном руководстве по CPP".

**Чувствительное к языку макропрограммирование с Scheme**

Хотя генераторы кода немного понимают целевой язык, они обычно не являются полными синтаксическими анализаторами и не могут работать с другим целевым языком без перезаписи компилятора.

Однако эта ситуация могла бы упроститься, если бы существовал язык, представленный простой структурой данных. В языке программирования Scheme сам язык представляется как связанный список, и он создан для обработки списков! Это делает Scheme идеальным (почти) языком создания программ, которые преобразуются - для синтаксического анализа программы не нужно ее объемного разбора. Scheme сам по себе является языком обработки списков.

На самом деле возможности Scheme по выполнению преобразований выходят за эти рамки. Стандарт Scheme определяет макроязык, специально созданный для облегчения создания дополнений к языку. Большинство реализаций Scheme предоставляют дополнительные возможности для помощи в создании генерирующих код программ.

Давайте еще раз рассмотрим проблемы наших C-макросов. Для макросаSWAPвы, во-первых, должны явно указать типы меняемых значений, и, во-вторых, вы должны использовать такое имя для временной переменной, которое, вы уверены, не используется где-либо еще. Рассмотрим, как выглядит эквивалент на языке Scheme, и как Scheme решает эти проблемы:

**Листинг 14. Макрос обмена значений в Scheme**

|  |
| --- |
| ;;Определить SWAP как макрос  (define-syntax SWAP  ;;Мы используем метод syntax-rules для создания макроса  (syntax-rules ()  ;;Rule Group  (  ;;Это шаблон, соответствие которому мы проверяем  (SWAP a b)  ;;Во что мы его преобразовываем  (let (  (c b))  (set! b a)  (set! a c)))))  (define first 2)  (define second 9)  (SWAP first second)  (display "first is: ")  (display first)  (newline)  (display "second is: ")  (display second)  (newline) |

Это макросsyntax-rules. В Scheme существует несколько макросистем, ноsyntax-rulesявляется стандартом.

В макросе syntax-rules

define-syntax является ключевым словом, используемым для определения преобразования. После ключевого словаdefine-syntaxуказывается имя определяемого макроса, а затем указывается преобразование.

syntax-rules- это тип применяемого преобразования. Внутри скобок находятся любые другие специфичные для макроса символы, отличные от имени самого макроса (в данном случае их нет).

Затем указывается последовательность правил преобразования. Программа преобразования синтаксиса проходит по каждому правилу и пытается найти совпадающий шаблон. После его нахождения программа запускает указанное преобразование. В данном случае существует только один шаблон:(SWAP a b).aиb- это переменные шаблона , которые соответствуют элементам кода в операторе вызова макроса и используются для перестановки во время преобразования.

С первого взгляда может показаться, что здесь имеются те же недостатки, что и в C-версии; однако есть несколько отличий. Во-первых, поскольку это язык Scheme, типы связаны с самими значениями, а не с именами переменных, поэтому абсолютно не надо беспокоиться о проблемах типов переменных, присутствующих в C-версии. Но нет ли здесь той же проблемы по именованию переменных, которая была ранее? То есть, если одна из переменных имеет имяc, не вызовет ли это конфликта?

На самом деле не должно быть никаких конфликтов. Макрос в Scheme, использующийsyntax-rules, является " гигиеническим ". Это означает, что все временные переменные, используемые в макросе, автоматически переименовываются перед подстановкой , для того чтобы предотвратить конфликт имен. Следовательно, в этом макросе с переименуется во что-нибудь еще перед подстановками, если имя одной из переменных для подстановки равноc. На самом деле, вероятнее всего, он будет переименован в любом случае. В листинге 15 представлен возможный результат макропреобразования программы:

**Листинг 15. Возможное преобразование макроса перестановки значений**

|  |
| --- |
| (define first 2)  (define second 9)  (let  (  (\_\_generated\_symbol\_1 second))  (set! second first)  (set! first \_\_generated\_symbol\_1))  (display "first is: ")  (display first)  (newline)  (display "second is: ")  (display second)  (newline) |

Как можно заметить, "гигиенический" макрос Scheme может предоставить вам преимущества других макросистем без многих их недостатков.

Иногда, однако, вы не захотите, чтобы макрос был "гигиеническим". Например, вы можете захотеть ввести в макрос связывания, доступные преобразуемому коду. Простое объявление переменной не действует, поскольку системаsyntax-rulesпросто переименует переменную. Поэтому большинство схем имеют также "негигиеническую" макросистему с названиемsyntax-case.

Макросыsyntax-caseписать труднее, но они намного мощнее, поскольку для ваших преобразований в определенной степени доступна вся исполняющая система Scheme. Макросыsyntax-caseне являются стандартом, но они реализованы на многих системах Scheme. Те же, которые не имеютsyntax-case, обычно имеют другие аналогичные системы.

Давайте рассмотрим основную форму макросаsyntax-case. Определим макрос с именемat-compile-time, который будет выполнять данную форму во время компиляции.

**Листинг 16. Макрос для генерирования значения или набора значений во время компиляции**

|  |
| --- |
| ;;Определить макрос  (define-syntax at-compile-time  ;;x - это синтаксический объект для преобразования  (lambda (x)  (syntax-case x ()  (  ;;Шаблон, аналогичный шаблону syntax-rules  (at-compile-time expression)  ;;with-syntax позволяет нам создавать синтаксические объекты  ;;динамически  (with-syntax  (  ;это - создаваемый нами синтаксический объект  (expression-value  ;после вычисления выражения преобразуем его в синтаксический объект  (datum->syntax-object  ;домен syntax  (syntax at-compile-time)  ;отметить значение кавычками, поскольку оно является литеральным значением  (list 'quote  ;вычислить значение преобразования  (eval  ;;преобразовать выражение из синтаксического представления  ;;в список  (syntax-object->datum (syntax expression))  ;;среда для вычисления  (interaction-environment)  )))))  ;;Просто возвратить сгенерированное значение как результат  (syntax expression-value))))))  (define a  ;;преобразовать в 5 во время компиляции  (at-compile-time (+ 2 3))) |

Данная операция выполнится во время компиляции. А именно, она выполнится во время расширения макроса, что не всегда совпадает с временем компиляции в системах Scheme. Любое выражение, разрешенное во время компиляции на вашей системе Scheme, будет доступно для использования в этом выражении. Теперь посмотрим, как это работает.

Вsyntax-caseвы, фактически, определяете функцию преобразования - lambda.x- это преобразуемое выражение.with-syntaxопределяет дополнительные синтаксические элементы, которые могут быть использованы в выражении преобразования.syntaxберет синтаксические элементы и комбинирует их вместе, следуя тем же правилам, что и программа преобразования вsyntax-rules. Давайте пошагово рассмотрим, что происходит:

1. Выражениеat-compile-timeсовпадает.
2. В самой внутренней части преобразованияexpressionпреобразуется в список представлений и вычисляется как обычный код схемы.
3. Результат объединяется с символом "кавычки" в список, для того чтобы Scheme обрабатывал его как литеральное значение, когда он станет кодом.
4. Эти данные преобразуются в синтаксический объект.
5. Синтаксическому объекту дается имяexpression-valueдля выражения его в результате работы.
6. Программа преобразования (syntax expression-value) указывает, чтоexpression-valueявляется нераздельным результатом из этого макроса.

С этой возможностью выполнять вычисления во время компиляции можно создать версию макросаTABLE, даже еще лучшую, чем при использовании языка C. В листинге 17 показано, как вы могли бы сделать это в Scheme с нашим макросомat-compile-time:

**Листинг 17. Создание таблицы квадратных корней в Scheme**

|  |
| --- |
| (define sqrt-table  (at-compile-time  (list->vector  (let build  (  (val 0))  (if (> val 20)  '()  (cons (sqrt val) (build (+ val 1))))))))  (display (vector-ref sqrt-table 5))  (newline) |

Его можно сделать даже еще более легким для использования, выполняя следующий макрос для построения таблицы, который будет очень похож на макрос языка C:

**Листинг 18. Макрос для создания таблиц преобразования во время компиляции**

|  |
| --- |
| (define-syntax build-compiled-table  (syntax-rules ()  (  (build-compiled-table name start end default func)  (define name  (at-compile-time  (list->vector  (let build  (  (val 0))  (if (> val end)  '()  (if (< val start)  (cons default (build (+ val 1)))  (cons (func val) (build (+ val 1))))))))))))  (build-compiled-table sqrt-table 5 20 0.0 sqrt)  (display (vector-ref sqrt-table 5))  (newline) |

Теперь у вас есть функция, позволяющая легко построить любой тип таблиц.

**Резюме**

Ух! Мы рассмотрели большой объем материала, поэтому давайте потратим минутку на обзор. Сначала мы обсудили тип проблем, которые лучше всего решаются генерирующими код программами. К ним относятся:

* Программы, нуждающиеся в предварительно сгенерированных таблицах данных
* Программы, имеющие большой объем стереотипного кода, который не может быть абстрагирован в функции
* Программы, использующие технологии, которые чрезмерно многословно выражаются на языке программы

Затем мы рассмотрели несколько систем метапрограммирования и примеры их использования. К ним относятся системы текстовой подстановки общего назначения.

Наконец, мы познакомились с Scheme и увидели, как он может решать проблемы, которые возникают в языке C, используя конструкции, являющиеся частью самого языка Scheme. Scheme является одновременно и языком, и генератором собственного кода. Поскольку эти технологии встроены в сам язык, то упрощается программирование, и исчезают многие проблемы, присущие другим рассмотренным технологиям. Это позволяет легко и просто добавить предметно-ориентированные расширения к языку Scheme в областях, традиционно занятых генераторами кода.

Во [второй части](http://www.interface.ru/home.asp?artId=2637) данной серии статей мы более детально рассмотрим программирование макросов в Scheme и то, как они могут значительно облегчить ваши широкомасштабные задачи программирования.

Искусство метапрограммирования, Часть 2: Метапрограммирование с использованием Scheme

**Оглавление**

* [Написание макросов syntax-case в Scheme](http://www.interface.ru/home.asp?artId=2637#N10084)
* [Знакомство с идентификаторами](http://www.interface.ru/home.asp?artId=2637#N10169)
* [Создание стереотипных макросов](http://www.interface.ru/home.asp?artId=2637#N10224)
* [Использование макросов для предметно-ориентированных языков](http://www.interface.ru/home.asp?artId=2637#N10285)
* [Заключение](http://www.interface.ru/home.asp?artId=2637#N10333)

В этой статье приводятся подробности программирования макросов Scheme и объясняется, как они могут значительно облегчить масштабные задачи программирования.

**Написание макросов syntax-case в Scheme**

Хотя макросыsyntax-caseне являются стандартной частью Scheme, они являются наиболее широко используемым типом макросов, позволяющим гигиенические и не гигиенические формы, и являются очень близкими к стандартным макросамsyntax-rules.

Макросыsyntax-caseимеют следующий формат (листинг 1):

**Листинг 1. Общий формат макросов syntax-case**

|  |
| --- |
| (define-syntax macro-name  (lambda (x)  (syntax-case x (другие ключевые слова, если имеются)  (  ;;Первый шаблон  (macro-name macro-arg1 macro-arg2)  ;;Расширение макроса (одна или несколько форм)  ;;(syntax - это зарезервированное слово)  (syntax (расширение макроса находится здесь))  )  (  ;;Второй шаблон - версия с одним аргументом  (macro-name macro-arg1)  ;;Расширение макроса  (syntax (расширение макроса находится здесь))  )  ))) |

Этот формат определяетmacro-nameкак ключевое слово, используемое для преобразования.lambda- это функция, используемая для преобразования выраженияxв его расширение.

syntax-caseпринимает выражениеxв качестве своего первого аргумента. Второй аргумент - это список ключевых слов, которые вставляются дословно в синтаксические шаблоны. Другие идентификаторы, используемые в шаблонах, будут работать как переменные шаблона. Затем вsyntax-caseуказывается последовательность комбинаций шаблон/преобразователь.syntax-caseобрабатывает каждую из них, пытаясь сопоставить входную форму шаблону и, при совпадении, выполняет соответствующее расширение.

Рассмотрим простой пример. Допустим, что мы хотим написать более подробную версию оператораif, чем предлагаемую в Scheme. И допустим, что мы хотим найти и возвратить большее из двух чисел. Код будет выглядеть примерно так:

(if (> a b) a b)

Для программистов, не работавших с Scheme, будет странным не видеть текстовых указаний ветви "then" и ветви "else". Чтобы решить этот вопрос, мы можем создать нашу собственную версию оператораif, в которой добавляются ключевые слова "then" и "else". Код будет выглядеть так:

(my-if (> a b) then a else b)

В листинге 2 приведен макрос для выполнения этой операции:

**Листинг 2. Макрос для определения расширенной версии оператора if**

|  |
| --- |
| ;;определить my-if как макрос  (define-syntax my-if  (lambda (x)  ;;установить, что "then" и "else" - это ключевые слова  (syntax-case x (then else)  (  ;;шаблон для соответствия  (my-if condition then yes-result else no-result)  ;;преобразователь  (syntax (if condition yes-result no-result))  )  ))) |

При выполнении этот макрос будет сопоставлять выражениеmy-ifс шаблоном следующим образом (другими словами, соответствие вызова макроса шаблону определения макроса):

|  |
| --- |
| (my-if (> a b) then a else b)  / / / / / /  / / / / / /  v v v v v v  (my-if condition then yes-result else no-result) |

В преобразующем выражении везде, где встретится словоcondition, оно будет заменено на(> a b). Не имеет значения, что(> a b)- это список. Это простой элемент, помещенный в список, и рассматривается он как отдельная сущность в шаблоне. Результирующее синтаксическое выражение просто переставляет каждую из этих частей в новом выражении.

Это преобразование происходит перед выполнением во время так называемого макрорасширения . Во многих основанных на компиляторах реализациях Scheme макрорасширение происходит во время компиляции. Это означает, что макросы выполняются только один раз, в начале программы или во время компиляции, и никогда не вычисляются повторно. Следовательно, наш операторmy-ifне вносит никаких накладных расходов - он преобразуется в простойifво время исполнения.

В следующем примере мы выполним известный макросswap!. Это будет простой макрос, предназначенный для перестановки значений двух идентификаторов. В листинге 3 приведен пример использования макроса.

**Листинг 3. Использование макроса swap! для перестановки значений идентификаторов**

|  |
| --- |
| (define a 1)  (define b 2)  (swap! a b)  (display "a is now ")(display a)(newline)  (display "b is now ")(display b)(newline) |

Следующий простой макрос (листинг 4) реализует перестановку путем определения новой временной переменной:

**Листинг 4. Определение вашего собственного макроса swap!**

|  |
| --- |
| ;;Определить новый макрос  (define-syntax swap!  (lambda (x)  ;;здесь мы не используем ключевых слов  (syntax-case x ()  (  (swap! a b)  (syntax  (let ((c a))  (set! a b)  (set! b c)))  )  ))) |

Здесь определяется новая переменная с именемс. Но что произойдет, если один из переставляемых аргументов будет называтьсяс?

syntax-caseрешает эту проблему, заменяя переменнуюсуникальным, неиспользуемым именем переменной при расширении макроса. Следовательно, синтаксический преобразователь сам обо все побеспокоится.

Обратите внимание на то, чтоsyntax-caseне заменяетlet, потому чтоlet- это определенный глобально идентификатор.

Техника замены имен переменных не конфликтующими именами называется гигиеной ; использующий эту технику макрос называется гигиеническим макросом . Гигиенические макросы могут безопасно использоваться везде без опасений конфликта с существующими именами переменных. Для широкого круга задач метапрограммирования эта возможность делает макросы более предсказуемыми и удобными в работе.

**Знакомство с идентификаторами**

Хотя гигиенические макросы делают представляемые в макросе имена переменных безопасными, существуют ситуации, когда вы хотите, чтобы макрос был не гигиеническим. Например, допустим, что вы хотите создать макрос, определяющий переменную в области видимости, которая используется вызывающей макрос программой. Такой макрос должен быть не гигиеническим, поскольку он вмешивается в пространство имен пользовательского кода. Однако во многих случаях эта способность является полезной.

В качестве простого примера, допустим, что вы хотите написать макрос, определяющий несколько математических констант для использования внутри макроса (да, это можно было бы сделать с использованием других средств, но я делаю именно так для примера). Допустим, мы хотим определить числаpiиe, используя следующий вызов макроса (листинг 5):

**Листинг 5. Вызов макроса определения математических констант**

|  |
| --- |
| (with-math-defines  (\* pi e)) |

Если бы мы попытались записать его как предыдущие макросы, он бы не работал:

**Листинг 6. Не работающий макрос определения математических констант**

|  |
| --- |
| (define-syntax with-math-defines  (lambda (x)  (syntax-rules x ()  (  (with-math-defines expression)  (syntax  (let ( (pi 3.14) (e 2.71828) )  expression))  )  ))) |

Этот фрагмент не работает. Причина этого, как упоминалось ранее, заключается в том, что Scheme будет переименовыватьpiиe, для того чтобы они не конфликтовали с другими именами в окружающих или вложенных областях видимости. Следовательно, они получат новое имя и код(\* pi e)будет ссылаться на неопределенные переменные. Нам необходим способ записывать литеральные символы, которые могут использоваться разработчиком, вызывающим макрос.

Для записи в макросе кода, который бы не модифицировался автоматически (гигиена) системой Scheme, код должен быть преобразован из списка символов в синтаксический объект , который можно было бы затем присвоить переменной шаблона и вставить в преобразованное выражение. Для этого мы будем использоватьwith-syntax, который по существу является оператором "let" для макросов. Он имеет аналогичный основной формат, но используется для присвоения синтаксических объектов переменным шаблона.

Для создания новой переменной шаблона вы должны быть способны транслировать символы и выражения в обоих направлениях между представлением в списке (способ записи синтаксиса) и более абстрактным представлением синтаксического объекта . Эти преобразования выполняют следующие функции:

* datum->syntax-objectпреобразует список в более абстрактное представление синтаксического объекта.
  + Первым параметром этой функции обычно является(syntax k)- немного магическая формула, помогающая преобразователю синтаксиса получить корректный контекст.
  + Вторым параметром является выражение, которое нужно преобразовать в синтаксический объект.
  + Результат - это синтаксический объект, который может быть присвоен переменной шаблона с использованиемwith-syntax.
* syntax-object->datumобратная процедураdatum->syntax-object. Она получает синтаксический объект и преобразует его в выражение, с которым можно работать, используя обычные функции Scheme для обработки списков.
* syntaxпринимает выражение преобразования, содержащее переменные шаблона, и константное выражение, а возвращает полученный синтаксический объект.

В данном примере для получения литерального значения в переменной шаблона вы должны использовать комбинацию синтаксиса иsyntax-object->datum. Затем вы могли бы поработать с выражением и использоватьdatum->syntax-objectдля получения его назад в виде синтаксического объекта, который можно присвоить переменной шаблона вwith-syntax. Затем в конечном выражении преобразования новая переменная шаблона может быть использована как любая другая переменная.

В сущности, вы преобразуете синтаксис Scheme в список, которым можно управлять, работаете с этим списком и преобразуете его назад в синтаксис выражения Scheme для вывода.

В листинге 7 приведено определение макроса, использующего эти функции для определения математических символов:

**Листинг 7. Работающий макрос определения математических констант**

|  |
| --- |
| (define-syntax with-math-defines  (lambda (x)  (syntax-case x ()  (  ;;Шаблон  (with-math-defines expression)  ;;with-syntax определяет новые переменные шаблона  (with-syntax  (  (expr ;;новая переменная шаблона  ;;преобразовать выражение в синтаксический объект  (datum->syntax-object  ;;syntax - местная магия  (syntax k)  ;;выражение для преобразования  `(let ( (pi 3.14) (e 2.72))  ;;Вставить код для переменной шаблона "expression"  ;;сюда.  ,(syntax-object->datum (syntax expression))))))  ;;Использовать новую созданную переменную шаблона "expr"  ;;как конечное выражение  (syntax expr))  )  ))) |

Если вы не знакомы с Scheme, обратная кавычка, называемая квазикавычкой , похожа на оператор "кавычка" за исключением того, что разрешает включать данные, не имеющие кавычек, если они начинаются с запятой (которая называется оператором закрытия кавычек ). Это позволяет нам соединить выражение с нашим фрагментом шаблонного кода, затем все это дело преобразовать обратно в синтаксический объект как конечное преобразование.

Поскольку мы явно соединили новые переменные с существующим синтаксическим объектом, они не могут быть переименованы. Также обратите внимание на то, что выражение(syntax k)вdatum->syntax-objectнеобходимо, но, по существу, бессмысленно. Оно используется для активизации маленькой "магии" в синтаксическом процессоре, для того чтобы функцияdatum->syntax-objectзнала, какой контекст выражения должен быть в ней обработан. Всегда записывается как(syntax k).

Проблема негигиенических макросов заключается в том, что определяемые переменные могут перезаписать другие переменные и быть перезаписанными другими переменными в коде. Это делает смешивание негигиенических макросов особенно опасным, поскольку макросы не будут знать, какие переменные используют другие макросы, и могут поменять значения переменных друг друга. Следовательно, негигиенические макросы должны использоваться только тогда, когда нет другого способа решить задачу с использованием обычных функций или гигиенических макросов, прчем в таких ситуациях символические определения макросов должны быть тщательно документированы.

**Заключение**

Метапрограммирование широко используется в программировании широкомасштабных проектов. В данной статье я коснулся инструментальных средств, необходимых для метапрограммирования на языке Scheme, а также привел несколько примеров. Технология метапрограммирования применялась в нескольких прикладных областях:

* Улучшение синтаксиса
* Автоматизация генерирования стереотипного кода

В Scheme вы можете использовать возможности макроязыка для определения практически любого типа предметно-ориентированного языка. Средство для этого существует. Просто нужно решить, какие возможности реализуются более легко и более понятно при помощи макрорасширений, а какие - при помощи обычного кода.