Монады в Clojure *

Jim Duey

19 августа 2011 г.

Clojure — многообещающий и перспективный язык программирования, работающий на платформе JVM и являющийся функциональным языком с Lisp-подобным синтаксисом.

Одной из примечательных идей функционального программирования (ФП) является идея монад. Однако, практически все руководства по монадам в сети (а их там множество) используют язык Haskell. Кроме того, во многих из них к монадам подходят со стороны теории категорий, поэтому может показаться, что понять монады возможно только обладая глубокими знаниями в теории ФП. В данной статье я постараюсь развеять эти опасения и показать как понимать и использовать монады в Clojure.

Konrad Hinsen написал великолепную реализацию монад на Clojure, и я не собираюсь делать этого заново. Она включена в пакет *clojure.contrib*, поэтому необходимо установить его в том случае, если у Вас возникнет желание использовать код, который будет приводиться далее. Весь код, используемый в статье, доступен здесь.

Konrad также написал статью по монадам в Clojure, которая доступна здесь.

1 Введение

Монада — это не более, чем способ композиции функций. Рассмотрим композицию функций с помощью *comp* из *clojure.core*:

```
; f1, f2 и f3 представляют собой функции, каждая из которых
; принимает один аргумент и возвращает некоторый результат.
(comp f1 f2 f3)
```

comp возвращает функцию, в которой результат работы функции *f3* передается функции *f2* в качестве аргумента, затем возвращаемое значение *f2* передается в *f1*, результат которой, наконец, возвращается пользователю. Следующий код эквивалентен ранее приведенному вызову *comp*:

```
(fn [x]
   (f1
     (f2
       (f3 x))))
```

*«Monads in Clojure»: http://intensivesystems.net/tutorials/monads_101.html.

А что если результат *f3* не соответствует аргументу, который требуется для *f2*? В этом случае, чтобы использовать их вместе, необходим некий дополнительный код. Пусть все функции *f1*, *f2* и *f3* принимают в качестве аргумента целое число, а возвращают список целых чисел. Чтобы получить результат композиции этих трех функций мы должны написать что-нибудь такое:

```
(fn [x]
   (mapcat f1
           (mapcat f2
                   (f3 x))))
```

Итак, *f3* получает в качестве аргумента x и возвращает список целых чисел. Затем внутренний *mapcat* применяет *f2* к каждому элементу из этого списка. Каждый вызов *f2* возвращает список целых чисел. Все эти списки от *f2* *mapcat* преобразовывает в единый список целых чисел. Затем внешний *mapcat* применяет *f1* к каждому целому числу из списка, полученного от внутреннего *mapcat*, и соединяет результат работы *f1* в один единый список, который и является результатом работы вышеприведенной функции. Идея, которая стоит за монадами, заключается в том, чтобы избавиться от этого дополнительного кода, скрыв таким образом сложность функции и оставив на поверхности только композицию функций.

2 Внутреннее устройство монады

А как абстрагировать этот дополнительный код? Для начала разберемся с терминологией. В статически типизированных языках (как Haskell) функция должна иметь сигнатуру, которая описывает тип каждого параметра и тип возвращаемого значения. Эта сигнатура может быть сама определена как тип, поэтому все функции, которые обладают одной и той же сигнатурой, имеют один и тот же тип. Clojure представляет собой динамически типизированный язык, поэтому функции не имеют сигнатур. Но мы можем сказать, что некоторые функции ожидают на вход вполне конкретные типы параметров и возвращают вполне конкретный тип результата. Мы это проделывали выше, говоря о том, что *f1*, *f2* и *f3* ожидают на вход целое число, а возвращают список целых чисел. Функции, для которых необходимо провести композицию с помощью монады, должны иметь один и тот же тип сигнатуры и называются «монадическими функциями». Значения, которые они возвращают, называются «монадическими значениями» и содержат или обертывают базисные значения. В выше приведенном примере монадическими значениями являются списки целых чисел (возвращаемые значения функции), а базисными значениями — целые числа (аргументы функции). Любая монадическая функция принимает базисное значение в качестве параметра. Если функции принимают монадические значения, получить их композицию можно просто воспользовавшись *comp*.

Теперь можно увидеть, что монадическая функция может быть вызвана обычным образом:

```
(f2 4)
```

А как передать монадической функции монадическое значение? Для этого необходима другая функция, которая принимает монадическое значение и монадическую функцию в качестве параметров и «выполняет все, что нужно». В нашем примере, такой функцией является *mapcat*. Однако, по соглашениям и для того чтобы

облегчить чтение кода, монадическое значение должно стоять перед монадической функцией в вызове функции. Поэтому, имея монадическое значение `[1 5 7]`, вместо

```
(mapcat f2 [1 5 7])
```

мы пишем

```
(m-bind [1 5 7] f2)
```

Стоит обратить внимание на две вещи. Во-первых, мы используем вектор в квадратных скобках вместо списка в круглых. Это идеологически более правильно для Clojure, а также проще читать, так как в круглые скобки заключается также вызов функции. Во-вторых, что такое *m-bind*?

m-bind — это стандартное имя для функции, которая применяет монадическую функции к монадическому значению. Необходимо, чтобы все монады имели функцию с таким именем. В нашем случае *m-bind* будет выглядеть следующим образом:

```
(defn m-bind [mv mf]
      (mapcat mf mv))
```

Возникает другой вопрос, что если мы хотим применить монадическую функцию к базисному значению? Если бы мы смогли преобразовать наше базисное значение в монадическое, мы далее просто могли бы использовать функцию *m-bind*. Каждая монада должна иметь такую функции и называться она должна *m-result*. В нашем примере мы должны написать следующий код

```
(m-result 6)
```

, который возвратит значение [6]. Поэтому *m-result* должна быть определена следующим образом:

```
(defn m-result [x]
      [x])
```

Итак, первый шаг к пониманию монад сделан. Монада — это способ композиции (здесь должны возникать понятия сигнатуры монадической функции, функции с именем *m-result*, преобразующей простое значение в монадическое, и функции с именем *m-bind*, применяющей монадическую функцию к монадическому значению). Монада может быть использована для композиции монадических функций, получая таким образом новые монадические функции.

3 Композиция

Как нам теперь создать функцию, являющуюся композицией функций *f1*, *f2* и *f3*, используя монаду? Можно так:

```
(defn m-comp [f1 f2 f3]
      (fn [x]
          (m-bind
              (m-bind
                  (m-bind
                      (m-result x)
                      f3)
                   f2)
               f1)))
```

На первый взгляд выглядит ужасно. Однако, стоит заметить, что эта функция независима от используемой монады. Поэтому мы можем использовать ее с любой монадой для композиции монадических функций, которые имеют такую же сигнатуру как монада, разработанная нами выше. *m-comp*, также как и *comp*, вызывается с функциями в качестве параметра:

```
(m-comp f1 f2 f3)
```

4 Нотация Do

Более подходящим способом для композиции функций является использование нотации *'do'*. Монада, которую мы определили выше, имеется в библиотеке *monads* и называется *'sequence-m'*. Монадическим значением для *sequence-m* является последовательность значений, а параметром монадической функции — некоторое базисное значение. Стоит заметить, что монадические значения не ограничены последовательностью целых чисел, это может быть последовательность значений любых типов. Композиция трех функций с использованием нотации *'do'* выглядит следующим образом:

```
(defn m-comp [f1 f2 f3]
      (fn [x]
          (domonad sequence-m
                   [a (f3 x)
                    b (f2 a)
                    c (f1 b)]
                   c)))
```

Этот код выглядит уже менее ужасно, чем код из предыдущей попытки. Конструкция *'domonad'* имеет три части. Первая часть представляет собой имя монады. Вторая — это вектор, заключенный в квадратные скобки и содержащий пары «имя выражения/выражение». И наконец, третья часть — это выражение для возвращаемого значения.

Здесь и происходит весь фокус. В парах «имя выражения/выражение» каждое выражение при выполнении возвращает монадическое значение. Т.е. выражение *'(f3 x)'* возвращает монадическое значение, зависящее от *x*. Однако, монадическое значение не связывается с *'a'*. Вместо этого, *'a'* используется в последующем выражении для доступа к базисным значениям, содержащимся внутри монадического значения, которое возвращает *'(f3 x)'*.

Итак, в вышеприведенном примере *'(f3 x)'* возвращает монадическое значение, основанное на *x*, а именно список значений. Затем *f2* применяется к каждому из значений из этого списка по порядку, производя при каждом вызове новый список, который собирается в единый список, элементы которого доступны через *'b'*. А затем этот процесс повторяется с *f1*, результат которого доступен через *'c'*.

Последний элемент в *'domonad'* представляет собой выражение для возвращаемого значения. Это выражение может включать любое имя из второй части *'domonad'* и используется для получения монадического значения, которое возвращается конструкцией *'domonad'*. Здесь нужно увидеть важную вещь: результатом *domonad* является монадическое значение, которое создает функция (монадическая), возвращаемая *m-comp*. Поэтому в дальнейшем *m-comp* может участвовать в композиции с другими монадическими функциями, таким образом образуя при помощи композиции дополнительные монадические функции. Композиция является одной из важной составляющей возможностей монад.

5 Раскрытие списков

В предыдущем разделе мы увидели важную вещь: выражения, связанные с некоторыми именами, возвращают монадические значения. А что будет, если выражения будут содержать просто списки?

Конструкция

```
(domonad sequence-m
         [letters ['a 'b 'c]
          numbers [1 2 3]]
         [letters numbers])
```

возвращает следующий результат

```
([a 1] [a 2] [a 3] [b 1] [b 2] [b 3] [c 1] [c 2] [c 3])
```

В выражениях не использовались монадические функции, но каждый элемент вектора *letters* образует пару с каждым элементов вектора *numbers*. Давайте взглянем на другое выражение Clojure, а именно макрос раскрытия списков (list comprehension):

```
(for
   [letters ['a 'b 'c]
    numbers [1 2 3]]
   [letters numbers])
```

, который возвращает следующий результат:

```
([a 1] [a 2] [a 3] [b 1] [b 2] [b 3] [c 1] [c 2] [c 3])
```

Этот результат показывает другой аспект монад, а именно то, что раскрытие списков является частным случаем конструкции *domonad* с *sequence-m* монадой. Или, если подойти с другой стороны, *sequence-m* монада является обобщением раскрытия списков и может иногда называться монадой раскрытия. Мы рассмотрим ее интересное приложении позже.

6 В чем преимущество?

Все это может быть и привлекательно, но в чем преимущество использования монад в коде? Одна из фундаментальных концепций информатики заключается в том, чтобы разделить задачу на подзадачи, решить эти подзадачи и затем использовать полученные решения для решения первоначальной проблемы. Используя монады, можно разбивать и разбивать проблему на маленькие части, пока каждая такая часть не будет решена с помощью монадической функции. Затем можно использовать комбинацию этих монадических функций, используя монадические комбинаторы, одним из которых является конструкция *domonad*, для решения основной задачи. В итоге мы получаем не только решение нашей основной задачи, но и набор решенных подзадач, который мы можем комбинировать для решения похожих проблем без написания нового кода. Это чем-то похоже на блоки Lego, которые могут использоваться для построения все больших и больших конструкций. Возможности набора монадических функций не возврастают линейно при добавлении новой функции. Они возрастают экспоненциально, потому что каждая новая монадическая функция может участвовать в композиции со всеми другими монадическими функциями. Мы рассмотрим конкретный пример этого в скором времени.

7 Монада *state-m*

Монада *sequence-m* является хорошим введением в использование монад. Существует еще несколько простых монад, которые за обманчивой простотой прячут в себе большие возможности. Мы вернемся к ним позже. А прямо сейчас рассмотрим другую составляющую великолепия монад.

Одной из наиболее мощных концепций ФП являются функции первого класса: с такими функциями можно проделывать все те вещи, что проделываются с целыми числами. Можно связать функцию с переменной, передать ее как параметр в другую функцию и возвратить ее как результат выполнения функции. Ее также можно использовать как монадическое значение.

Функции сами по себе могут быть монадическими значениями, которые отличаются от их монадических функций. Это означает, что монада может быть определена так, что монадическая функция возвращает функцию как монадическое значение. Чтобы понять эту концепцию, необходимо четко представлять себе различие между функцией, которая является монадическим значением, и функцией, которая является монадической функцией.

Давайте рассмотрим конкретный пример, а именно монаду *state-m*. Монадическим значением монады *state-m* является функция, которая принимает значение (состояние state) в качестве параметра и возвращают список, который содержит возвращаемое значение и новое значение состояния. Состояние может быть любым типом: отображением, строкой, целым числом и так далее. Монадическими функциями монады *state-m* являются функции, которые принимают значение и возвращают монадическое значение (функцию, которая принимает значение состояния и возвращает список, состоящий из возвращаемого значения и нового состояния).

Вот пример функции, которая является монадическим значением в монаде *state-m*:

```
(defn some-fn [state]
```

```
    ; do something
    ; and something else
    [return-value new-state])
```

Заметим, что функция сама по себе является монадическим значением, а не ее значение состояния.

Теперь давайте определим несколько таких функций, в которых состояние будет целым число, а их работа будет заключаться в простом увеличении этого состояния. Возвращаемым значением (первым элементом списка) этих функций будут константы.

```
(defn g1 [state-int]
      [:g1 (inc state-int)])

(defn g2 [state-int]
      [:g2 (inc state-int)])

(defn g3 [state-int]
      [:g3 (inc state-int)])
```

Теперь без определения каких-либо еще монадических функций для этой монады давайте произведем их композицию, используя выражение *domonad*, и привяжем результат *domonad* к некоторому имени. Напомню, что *g1*, *g2* и *g3* являются монадическими значениями и поэтому используются прямо в *domonad*, без всякого вызова для получения значения, которые они возвращают:

```
(def gs (domonad state-m
                 [a g1
                  b g2
                  c g3]
                 [a b c]))
```

Какое значение имеет *'gs'*? Мы знаем, что *domonad* возвращает монадическое значение, а для монады *state-m* монадическим значением является функция, которая принимает значение состояния. Поэтому *'gs'* является функцией от одного аргумента, которым является состояние.

А что эта функция возвращает? Монадическое значение монады *state-m* возвращает список с возвращаемым значением и новым состоянием, поэтому именно это вернет *'gs'*. Ее возвращаемое значение (первый элемент списка) определено выражением *'[a b c]'*, где переменные связаны с возвращаемыми значениями функций *g1*, *g2* и *g3* соответственно. А что за значение состояния вернет *'gs'*? Может быть это и не очевидно, но она вернет значение состояния, полученного от *g3*.

Поэтому

```
(gs 5)
```

вернет

```
([:g1 :g2 :g3] 8)
```

Другим интересным моментом монады *state-m* является наличие специальных функций. Например, функция

```
(defn fetch-state []
      (fn [state]
          [state state]))
```

вернет функцию, которая может быть использована для получения значения состояния в любой точке выражения *domonad*:

```
(def gs1
     (domonad state-m
              [a g1
               x (fetch-state)
               b g2]
              [a x b]))

(gs1 3)
```

вернет

```
([:g1 4 :g2] 5)
```

Имя выражения *'x'* будет связано со значением состояния, которое было в момент вызова *fetch-state*.

Противоположной для *fetch-state* функцией является *set-state*:

```
(defn set-state [new-state]
      (fn [old-state]
          [old-state new-state]))
```

Код

```
(def gs2
     (domonad state-m
              [a g1
               x (set-state 50)
               b g2]
              [a x b]))

(gs2 3)
```

вернет

```
([:g1 4 :g2] 51)
```

Заметим, что *fetch-state* и *set-state* не являются монадическими значениями. Они представляют собой функции, которые возвращают монадические значения для монады *state-m*, принимая состояние и возвращая список, состоящий из возвращаемого значения и нового значения состояния. При этом *set-state* является монадической функцией, так как она принимает аргумент, который является новым состоянием и возвращает функцию, которая при своем вызове устанавливает состояние в это значение.

Если мы заглянем внутрь монады *state-m*, то увидим следующее определение:

```
(defn m-result [v]
      (fn [s]
          (list v s)))

(defn m-bind [mv f]
      (fn [s]
          (let [[v ss] (mv s)]
               ((f v) ss))))
```

Функция *m-result* принимает значение и возвращает функцию, которая принимает состояние и возвращает значение и состояние единым списком. Никаких фокусов, так как можно вспомнить, что *m-result* должна возвращать монадическое значение, которое в данном случае является функцией.

Функция *m-bind* немного сложнее. Во-первых, заметим, что она, как и положено, принимает два параметра. Первым аргументом является монадическое значение, которое представляет собой функцию. Вторым аргументом является монадическая функция. А сейчас будет интересно: параметр, с которым работает эта монадическая функция не является монадическим значением. На самом деле это значение, которое возвращается, когда функция применяется к состоянию.

Итак, *m-bind* возвращает функцию, которая является монадическим значением и принимает в качестве аргумента значение состояния. Монадическое значение, которым является первый аргумент *m-bind*, применяется к этому значению состояния и возвращает некоторое новое значение и новое состояние. (Конструкция *'let'* используется для того, чтобы по отдельности использовать элементы возвращаемого списка от монадического значения.) Затем монадическая функция применяется к этому некоторому значению, возвращая функцию, которая является монадическим значением. Эта последняя функция применяется к новому состоянию, возвращая, наконец, конечные значение и состояние.

Все это сбивает с толку. Попытка выполнять примеры в REPL приводит к долгому осознанию того, как это все работает. Ведь нужно помнить, что для монады *state-m* монадическим значением является функция, которая принимает значение состояния и возвращает список из возвращаемого значения и нового состояния. Монадическая функция принимает значение и возвращает монадическое значение, которое является функций и так далее, и так далее. Но понять монады, стоит потраченных усилий.

Давайте на мгновение вернемся на шаг назад и вспомним, что состояние может быть любым. Например, оно может быть отображением, в котором значения ассоциированы с ключевыми символами (keyword). А функция, являющаяся монадическим значением, может изменять значения в этом отображении, имея таким образом глобальное изменяемое состояние и все еще сохраняя при этом все преимущества программирования без побочных эффектов. Также состояние может быть строкой, которая разбирается функцией (организация синтаксического анализатора), или целым числом, которое увеличивается или уменьшается на единицу (организация счетчика).

Также монада *state-m* может быть использована для организации многоуровневой абстракции в одном приложении. Для примера давайте рассмотрим хорошо известную игру Pac-Man. На одном уровне состоянием может быть текущий лабиринт, а монадическими значениями могут быть функции, которые обновляют позиции всех персонажей, проверяют коллизии, создают видео фреймы или проигрывают

звуки. Эти функции могут участвовать в композиции при помощи *m-bind*, образуя единственную функцию, принимающую текущее состояние лабиринта и возвращающую следующее.

Эта функция может быть использована более высоким уровнем абстракции, где состоянием будет являться список начальных состояний лабиринтов, по одному для каждого этапа игры. Монадическим значением этого уровня абстракции будет функция, позволяющая игроку играть на текущем лабиринте. Как только игрок проходит лабиринт, снова вызывается эта функция со списком оставшихся лабиринтов.

8 Законы

В нашем первом определении монады мы говорили о том, что составляющими любой монады являются сигнатура функции, функция под названием *m-result* и функция под названием *m-bind*. Но мы не упомянули, что *m-result* и *m-bind* не могут быть любыми функциями. Они должны работать вместе так, чтобы монадическая функция могла свободно участвовать в композиции, возвращая предсказуемый результат. Эти правила постулированы в трех законах монад, которые *m-result* и *m-bind* обязаны соблюдать для того, чтобы получилась монада. Вспомним, что *m-result* преобразует базисное значение в монадическое, а *m-bind* применяет монадическую функцию к базисному значению, извлеченному из монадического значения.

Первый закон

```
(m-bind (m-result x) f) эквивалентно (f x)
```

означает, что что бы *m-result* ни делало с *'x'*, превращая его в монадическое значение, *m-bind* преобразует его обратно, применяя *'f'* к *'x'*.

Второй закон:

```
(m-bind mv m-result) эквивалентно mv
```

, где *'mv'* представляет собой монадическое значение. Этот закон дополняет первый. Он гарантирует, что что бы *m-bind* ни делало для извлечения значения из монадического значения, монадическая функция *m-result* преобразует это значение обратно в монадическое значение.

При работе монады *m-result* и *m-bind* являются зависимыми функциями: каждая из них зависит от реализации другой, и два вышеприведенных закона четко описывают схему зависимости.

Третий закон:

```
(m-bind (m-bind mv f) g) эквивалентно (m-bind mv (fn [x] (m-bind (f x) g)))
```

, где *'f'* и *'g'* являются монадическими функциями, а *'mv'* — монадическим значением. Третий закон говорит о том, что не важно применяется ли *'f'* к *'mv'*, а затем *'g'* к полученному результату, или создается новая монадическая функция, являющаяся композицией *'f'* и *'g'*, которая затем применяется к *'mv'*. В любом случае результатом должно быть одно и то же монадическое значение.

Один интересный факт возникает из того, что все монады должны следовать этим трем законом: функции, которые написаны только с помощью *m-result* и *m-bind* будут работать для всех монад. Это повышает уровень абстракции, на котором строятся программы, скрывая частности, они делают структуру приложения простой и наглядной.

9 Нулевой элемент

До сих пор мы обсуждали только функции *m-result* и *m-bind*, так как это минимум, который обязана реализовывать монада, чтобы называться монадой. Если монада определит несколько больше функций, то это сделает ее более выразительной.

Есть много полезных вещей, которые могут быть выполнены при использовании только натуральных чисел (1, 2, 3 ...). Однако, когда добавляется концепция «отсутствия (nothing)» при помощи цифры 0, то такая система чисел становится более полезной. Точно таким же образом, когда монада добавляет концепцию «отсутствия», она становится полезной для решения большего круга проблем.

Стандартным именем для монадического значения «отсутствия» является *'m-zero'*, и оно должно соблюдать несколько законов. Первый закон

```
(m-bind m-zero f) возвращает m-zero
```

говорит о том, что любая попытка применить монадическую функцию к *m-zero* вернет *m-zero*.

Второй закон

```
(m-bind mv (fn [x] m-zero)) возвращает m-zero
```

гласит, что любая монадическая функция, которая возвращает *m-zero*, всегда будет возвращать значение *m-zero* независимо от того, к какому монадическому значению она применяется.

m-zero может быть полезна для определения ошибки или завершения дальнейшего выполнения *domonad*.

10 Операция сложения

Другим расширением монады, близко связанным с *m-zero*, является операция сложения, которая обычно определяется функцией с именем *m-plus*. Эта функция принимает два или более монадических значения и манипулирует ими определенным способом, образуя новое монадическое значение. Существует несколько законов, которые связывают *m-zero* и *m-plus*:

```
(m-plus mv m-zero) возвращает mv
(m-plus m-zero mv) возвращает mv
```

Эти два закона представляют собой два разных способа выразить одно и тоже: *m-plus* игнорирует любой параметр, имеющий *m-zero* значение, а остальными оперирует обычным образом.

11 Синтаксический анализ

Предыдущий материал являлся введением в монады. Теперь, очевидно, возникает вопрос, а для чего монады пригодны? Для того чтобы проиллюстрировать пригодность монад на примере, рассмотрим синтаксический анализ методом рекурсивного спуска.

Синтаксический анализ (парсинг) представляет собой извлечение в соответствии с заданной грамматикой из последовательности символов смысла. Метод рекурсивного спуска пытается разобрать последовательность символов, следуя некоторому правилу. Если правило не выполняется, то метод пытается разобрать ту же последовательность с помощью другого правила. Построение синтаксического анализатора (парсера) возможно с помощью монад, и сделать это будет на удивление легко. Монады и другие функции, приведенные ниже, представлены в статье «Monadic Parsing in Haskell».

Идея состоит в том, чтобы определить парсер как функцию, принимающую строку в качестве параметра и определяющую соответствует ли она грамматике или нет. Если строка (или некоторая начальная часть строки) удовлетворяет парсеру, то он (в нашем случае это функция) возвращает некоторое значение, которое является смыслом разобранной части, и оставшуюся после разбора часть строки. Если строка не удовлетворяет парсеру, возвращается *nil*. Как и все элегантные идеи, эта идея очевидна, но для ее разработки потребовались большие усилия. Статья, описывающая ее, стоит того, чтобы ее прочитать.

Несколько замечаний. Парсер представляет собой функцию, которая принимает строку и возвращает список из некоторого значения и новой строки. Но это есть описание *state-m* монады, где состоянием является строка. Исключением является то, что парсер может вернуть *nil* в том случае, если строка не удовлетворяет парсеру. Поэтому для реализации такой монады потребуется что-то похожее на *state-m*, но позволяющее использовать *nil* в качестве монадического значения.

Для определения монады в библиотеке *clojure.contrib.monads* предусмотрена конструкция *'defmonad'*. Таким образом, реализация монады парсера может выглядеть следующим образом:

```
(defmonad parser-m
          [m-result (fn [x]
                      (fn [strn]
                          (list x strn)))

           m-bind (fn [parser func]
                    (fn [strn]
                        (let [result (parser strn)]
                             (when (not= nil result)
                             ((func (first result)) (second result))))))

           m-zero (fn [strn]
                    nil)

           m-plus (fn [& parsers]
                    (fn [strn]
                        (first
                              (drop-while nil?
                                          (map #(% strn) parsers)))))])
```

Функция *m-result* точно такая же как и функция *m-result* в монаде *state-m*.

Функция *m-bind* немного модифицирована. Возвращаемая функция принимает строку и применяет парсер к ней также как и в монаде *state-m*. Однако, затем идет

проверка: является ли результат *nil*-ом или нет. Если да, то возвращается *nil*. Если нет, то он разбивается на части и обрабатывается также как и в монаде *state-m*.

Монада *state-m* не имеет функции *m-zero*. Значением «отсутствие», которое сигнализирует об ошибке, является *nil*, поэтому монадическим значением *m-zero* является функция, которая лишь принимает строку и возвращает *nil*.

Функция *m-plus* потребует некоторых размышлений. В общем случае в монадах *m-plus* используется для объединения монадических значений. В монаде *parser-m* монадическими значениями являются функции. Чтобы объединить функции, можно вызвать их одну за другой, передавая результат одной функции как параметр другой, возможно, вставляя некоторый дополнительный код между ними. Также можно вызвать их всех с одинаковым набором параметров и затем либо отобрать один из результатов, либо собрать все результаты в единственное значение.

Реализация разбора методом рекурсивного спуска потребует от нас разобрать строку с помощью парсера, и в том случае если этот процесс завершится неудачей, необходимо будет попробовать следующий парсер, и так до тех пор пока один из парсеров не завершится удачно. Эта задача как раз подходит для *m-plus*. *m-plus* принимает в качестве параметра список парсеров и возвращает функцию, которая принимает строку. Эта функция применяет один за другим парсер к строке, возвращая список результатов, полученных от работы парсеров. Все что остается, это найти первый не-*nil* результат и вернуть его в качестве результата работы функции. Дополнительным преимуществом является то, что *first*, *drop-while* и *map* являются ленивыми, поэтому *m-plus* применяет только то количество парсеров, которое требуется для того чтобы получить результат. Если ни один из парсеров не завершится удачно, то возвращается *nil*.

Теперь, когда монада реализована, давайте вернемся к самим парсерам, которых у нас уже два: *m-result* и *m-zero*, но они слишком просты и неинтересны.

Рассмотрим самый простой парсер, который будет распознавать первый символ строки, а если строка пуста, то возвращать *nil*:

```
(defn any-char [strn]
      (if (= "" strn)
          nil
          (list (first strn) (. strn (substring 1)))))
```

Вспомним, что когда парсер возвращает *nil*, то весь процесс разбора останавливается, поэтому этот парсер может использоваться для прекращения разбора в том случае, если строка пуста. Когда символ распознан, парсер должен вернуть значение, представленное распознанным символом, и новую строку, полученную из старой путем удаления первого символа.

Рассмотрим немного более сложный парсер, который принимает строку, проверяет первый символ этой строки и, если проверка завершилась неудачей, возвращает *nil*, в противном случае — распознанный символ и новую строку.

```
(defn char-test [pred]
      (domonad parser-m
               [c any-char
                :when (pred c)]
               (str c)))
```

Первое, что стоит отметить: функция *'char-test'* не является парсером. Она создает парсер, используя для этого конструкцию *domonad*. Параметр *'pred'*, который передается в *char-test*, представляет собой функцию, проверяющую первый символ и возвращающую *true* или *false* в зависимости от того, прошла проверка успешно или нет. Принимая во внимание эти замечания, давайте внимательнее взглянем на конструкцию *domonad* в этой функции.

В первом выражении *any-char* должна обработать строку и связать результат с *'c'*. Если *any-char* вернет *nil*, что означает пустую строку, то парсер также вернет *nil*.

Теперь разберемся, что делает здесь *:when*? Это специальное выражение, называемое охраной, или охранным выражением, разрешающее дальнейшее вычисление в том случае, если результатом его выполнения является *true*. Поэтому когда встречается *:when*, то вызывается *'pred'* со значением *'c'*. Если *'pred'* вернет *false*, то парсер завершает работу и возвращает *nil*. В противном случае парсер возвращает то, что определено в заключительном выражении конструкции *domonad*. В нашем случае *char-test* возвращает разобранный символ как строку.

Важный вывод, который стоит здесь сделать, заключается в том, что работа функций, обрабатывающих строку, скрыта и не касается нас. Код был написан, отлажен и забыт.

Итак, как определить парсер, который проверяет конкретный символ?

```
(defn is-char [c]
      (char-test (partial = c)))
```

Нет ничего проще. Функция *is-char* принимает на вход символ, а затем используется *partial* для создания другой функции, которая проверяет этот символ на равенство с первым символом входящей строки. *is-char* передается в качестве предиката функции *char-test*, которая строит парсер. Этот парсер принимает на вход строку и, если первый символ этой строки сопадает с переданным в *is-char* символом, то возвращает список из двух строк: строка из первого символа и оставшаяся часть входной строки. Если первый символ не совпадает, то возвращается *nil*.

```
(def is-n (is-char \n))

(assert (= '("n" "bc")
           (is-n "nbc")))

(assert (empty?
           (is-n "xbc")))
```

Очевидно, что следующим парсером, который необходимо написать, должен быть парсер, проверяющий один за другим последовательность символов:

```
(defn match-string [target-strn]
      (if (= "" target-strn)
          (m-result "")
          (domonad parser-m
                   [c (is-char (first target-strn))
                    cs (match-string (. target-strn (substring 1)))]
                   (str c cs))))
```

Давайте вспомним: во-первых, *match-string* является функцией, которая возвращает парсер, а парсеры представляют собой монадические значения для монады *parser-m*. Во-вторых, *m-result* и *is-char* являются функциями, которые также возвращают парсеры. В-третьих, в конструкции *domonad* в парах «имя выражения/выражение» выражения должны возвращать монадические значения (парсеры), а переменные связываться с результатом выполнения этих парсеров.

Поэтому *match-string* проверяет является ли входящая строка пустой. Если это так, то просто возвращается парсер, который всегда возвращает пустую строку. Этот случай является базисом рекурсии.

Если входящая строка не пуста, то вызовом *domonad* создается новый парсер. Отсюда начинается весьма интересная часть кода. Первое выражение обычно и является вызовом *is-char*, который возвращает парсер, разбирающий первый символ входящей строки. После успешной работы *is-char* парсера *'c'* связывается с разобранным значением этого символа. Второе выражение *domonad* выполняет рекурсивный вызов *match-string*, создавая парсер, который разбирает остаток строки. Этот парсер выполнится только тогда, когда первый символ строки разобран успешно: в этом случае первый символ удаляется, а оставшаяся строка передается для дальнейшего разбора парсеру *match-string*. Если его работа завершится успешно, то возвращенное им значение будет ассоциировано с *'cs'*. Если какая-то часть строки остается не разобранной, то она становится частью результата вместе с заключительным выражением *domonad*, которое создает единую строку из *'c'* и *'cs'*.

11.1 Парсер-комбинаторы

Парсеры, которые мы до сих пор рассматривали, являются базисными блоками для построение синтаксического анализатора методом рекурсивного спуска. Сейчас нам необходимо из них построить более сложные парсеры, способные разбирать произвольные грамматики. Для этого нам нужен некоторый набор парсер-комбинаторов, основная идея которых заключается в том, чтобы получить парсер или несколько парсеров, затем модифицировать или объединить их для создания нового парсера, который затем может также участвовать в композиции простым блоком и т.д., т.д.

Рассмотрим парсер-комбинатор, полученный путем преобразования из парсера, который ОБЯЗАН обнаружить совпадение строки, в парсер, который МОЖЕТ и обнаружить заданную строку, а может и нет.

```
(defn optional [parser]
      (m-plus parser (m-result nil)))
```

Данный парсер полезен для случая, когда мы хотим сделать что-то необязательным. Для этого нам необходимо найти способ создания нового парсера, который завершается успешно, когда работа первого парсера заканчивается неудачей. Новый парсер должен вернуть значение, позволяющее дальнейший разбор с того же места, не влияя на первоначальную строку. Эта задача легко решается с помощью *m-result*. Он объединяется с первым парсером, используя *m-plus*, так, что вначале первый парсер пытается разобрать строку, и если эта попытка заканчивается неудачей, то выполняется парсер *m-result*, который возвращает список, состоящий из *nil* и первоначальной строки.

Следующий комбинатор принимает список парсеров и возвращает результат первого парсера, который завершился успешно. Данное описание соответствует деятельности *m-plus*, поэтому для создания этого парсера достаточно переименовать *m-plus*:

```
(def match-one m-plus)
```

Также нам необходим комбинатор, который принимает набор парсеров, выполняет их один за другим и возвращает конечный результат. Конструкция *domonad* делает что-то похожее, но в случае с *domonad* мы должны заранее знать количество парсеров. А нам необходим комбинатор, работающий с неизвестным набором парсеров.

```
(defn match-all [& parsers]
      (let [combined-parsers (m-seq parsers)]
           (fn [strn]
               (let [result (combined-parsers strn)]
                    (when result
                          (list (apply str (first result))
                                (second result)))))))
```

Функция *m-seq* является стандартной монадной функцией, доступной всем монадам. Она принимает список монадических значений и последовательно выполняет их композицию, возвращая монадическое значение (в нашем случае парсер). Проблема с *m-seq* заключается в том, что этот парсер возвращает не список, состоящий из разобранной строки и оставшейся части строки, а список, состоящий из списка строк и оставшейся строки. Поэтому нам необходимо собрать список из строк в одну строку, а затем вернуть список из нашей собранной строки и оставшейся строки.

Эту реализацию *match-all* прислал мне Konrad Hinsen. Первое, что хочется отметить, *combined-parsers* представляет собой монадическое значение. Второе — если вынести вызов *'apply str'*, то функция, возвращаемая *match-all*, выглядит идентично функции, возвращаемой *m-bind* для монады *parser-m*.

Вспомним, что монадическим значением для монады *parser-m* является функция, принимающая строку для разбора и возвращающая список из некоторого значения и оставшейся части строки. А монадическая функция монады *parser-m* принимает некоторое значение и возвращает монадическое значение. В нашем случае нам необходима функция, которая принимает список строк и собирает их в одну, а затем возвращает функцию, которая принимает состояние и возвращает список из объединенной строки и неизменного состояния. Последняя часть предыдущего предложения выполняется следующим кодом:

```
(m-result (apply str x))
```

, где *'x'* является списком строк, который объединяется и затем возвращается вместе с неизмененным состоянием. Поэтому, функция, написанная следующим образом

```
(fn [x]
    (m-result (apply str x)))
```

, может быть монадической функцией для *parser-m*, а также являться монадическим значением, т.е. парсером, возвращающим список строк, которые необходимо затем объединить, что является задачей, решаемой с помощью *combined-parsers*. Поэтому, *match-all* может быть переписана следующим образом:

```
(defn match-all [& parsers]
               (m-bind (m-seq parsers)
                       (fn [x]
                           (m-result
                               (apply str x)))))
```

Это даже более лаконично, чем вариант Konrad Hinsen. Можно сравнить этот вариант кода с реализацией функции *m-fmap*.

Для полноты нашего набора парсер-комбинаторов, давайте рассмотрим способ распознавания с повторением. Существует два варианта этого комбинатора: один из них требует по-крайней мере одного успешного завершения работы парсера, а другой нет. Они взаимно рекурсивные: каждый может быть определен через другой. Определить взаимно рекурсивные функции в Clojure достаточно легко:

```
(def one-or-more)

(defn none-or-more [parser]
      (optional (one-or-more parser)))

(defn one-or-more [parser]
      (domonad
               [a parser
                as (none-or-more parser)]
               (str a as)))
```

Кроме взаимной рекурсии эти комбинаторы не требуют дополнительных пояснений.

Теперь с нашим набором примитивных парсеров и парсер-комбинаторов возможно построить синтаксический анализатор методом рекурсивного спуска для любой грамматики. Последнее, что мы сделаем, это взглянем на несколько удобных функций:

```
(defn one-of [target-strn]
      (let [str-chars (into #{} target-strn)]
           (char-test #(contains? str-chars %))))

(def alpha (one-of "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"))
(def whitespace (one-of " \t\n\r"))
(def digit (one-of "0123456789"))
(def hexdigit (one-of "0123456789abcdefghABCDEFGH"))
```

Функция *one-of* может быть реализована с помощью *is-char* для каждого символа в *target-strn*. Однако, более эффективно создать *hash-set* из символов *target-strn* и затем, используя *char-test*, проверять находится ли данный символ в этом *hash-set*.

Парсеры *alpha*, *whitespace*, *digit* и *hexdigit*, как и любые другие парсеры, могут участвовать в композиции.

12 Заключение

В данной статье речь шла об основах монад. В следующий раз я расскажу как написать монаду *parser-m* в одну строку:

```
(def parser-m (state-t maybe-m))
```