Что такое Clojure?

Совсем недавно для того, чтобы запустить программу на Java Virtual Machine (JVM), необходимо было написать эту программу на языке Java. Эти дни безвозвратно ушли в прошлое, потому что теперь есть широкий выбор языков программирования для JVM. Наиболее популярные, такие как Groovy, Ruby (с интерпретатором JRuby), и Python (с интерпретатором Jython), предназначены в основном для процедурного программирования или же относятся к группе объекто-ориентированных языков. Парадигмы процедурного и объекто-ориентированного программирования хорошо знакомы Java-программистам, так что сторонники Java имеют веский аргумент в свою пользу – используя вышеназванные языки, вы напишете примерно такой же код, какой вы написали бы на Java, вам просто придется использовать другой синтаксис.

Clojure – тоже язык программирования для JVM, однако он в корне

отличается от Java-технологии и других упомянутых выше языков программирования для JVM. Clojure – диалект Lisp. Семейство языков программирования Lisp существует уже довольно длительное время – фактически с 50-х годов прошлого века. Lisp использует S-выражения, или *польскую префиксную* запись. Такая запись выглядит как (`function arguments...`). Вы всегда начинаете с задания имени функции, за которым следует список (пустой или непустой) аргументов, передаваемых функции. Чтобы понять, где начинается и где заканчивается определение функции, ее имя и список аргументов заключаются в скобки. Именно это требование синтаксиса языка привело к появлению огромного количества скобок в коде, что стало своего рода торговым знаком Lisp.

Как можно догадаться, Clojure – функциональный язык программирования. С чисто академической точки зрения его «чистоту» можно оспаривать, однако Clojure, безусловно, опирается на основные принципы функционального программирования: отказ от изменяемых переменных (mutable state), использование рекурсии и

функций высокого порядка и т.п. В Clojure используется динамическая типизация, однако при необходимости вы можете задать определенный тип данных, чтобы обеспечить максимальную производительность критически важных фрагментов кода. Clojure не просто работает на JVM, он разрабатывался с учетом операционной совместимости с Java. Кроме того, создатели Clojure ориентировались на поддержку многопоточности, так что Clojure предоставляет разработчикам уникальные возможности для параллельного программирования.

Clojure в примерах

Для большинства программистов лучший способ освоить новый язык– это начать писать на нем программы. Поэтому мы рассмотрим несколько простых проблем, с которыми часто сталкиваются разработчики программ, и попробуем их решить с помощью Clojure. Детальный разбор решений поможет вам лучше понять, как работает Clojure, как его использовать, и для каких задач он наиболее подходит. Для работы с Clojure, как и для других языков программирования, сначала необходимо установить соответствующую среду разработки. К счастью, в случае Clojure сделать это довольно просто.

Минимальные требования

Все, что вам нужно для работы с Clojure, - это JDK и библиотека Clojure, которая содержится в одном JAR-файле. Существует два стандартных способа разработки и запуска программ на Clojure. Наиболее распространенный способ – это использование простой интерактивной среды разработки REPL, цикла «чтение-вычисление-печать».

Листинг 1. Запуск Clojure REPL

```
$ java -cp clojure-1.0.0.jar clojure.lang.Repl
Clojure 1.0.0-
user=>
```

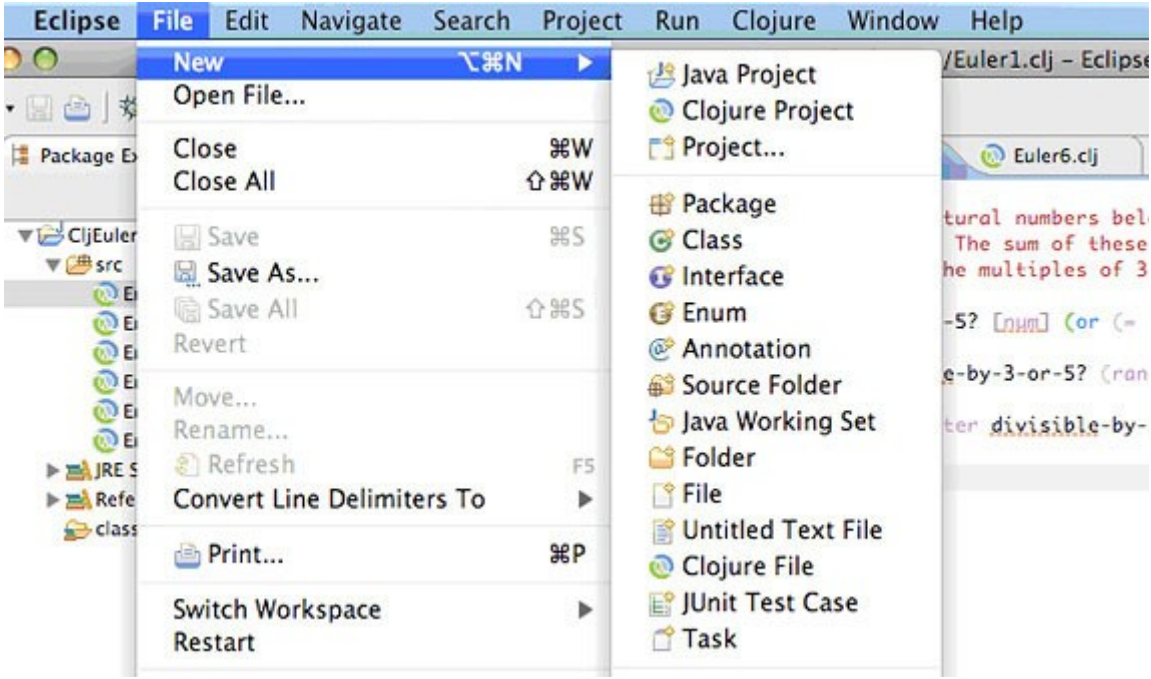
Команда должна выполняться в директории, где расположен JAR-файл Clojure. Если вам это неудобно, добавьте путь до соответствующего JAR-файла к своим стандартным путям поиска. Вместо «ручного» запуска команды REPL вы можете написать скрипт, который будет выполнять эту команду, и просто запускать свой скрипт. Для этого вам надо будет вызвать Java-класс clojure.main.

Листинг 2. Вызов clojure.main.

```
$ java -cp clojure-1.0.0.jar clojure.main /some/path/to/Euler1.clj
233168
```

Не забудьте определить путь к JAR-файлу Clojure и к вашему скрипту. Наконец, существует интегрированная среда разработки для Clojure. Пользователи Eclipse могут скачать и установить плагин clojure-dev. После установки плагина переключитесь в Java-перспективу. Теперь вы можете создать новый Clojure-проект и добавить в него файлы с исходным кодом Clojure, как показано на рисунке:

Рисунок 1. Использование плагина clojure-dev в Eclipse



Установив плагин clojure-dev, вы сможете использовать синтаксические подсказки и встроенные проверки Eclipse, в том числе для обеспечения соответствия количества открывающих и закрывающих скобок (функциональность, абсолютно необходимая при работе с любой разновидностью Lisp). Кроме того, Eclipse включает в себя интерактивную среду REPL, так что вы сможете выполнить любой скрипт в командном окне REPL в Eclipse. Плагин Clojure был выпущен незадолго до написания этой статьи, однако его функциональность очень быстро развивается. Теперь, когда у нас есть необходимая среда разработки, можно приступить к созданию программ на Clojure.

Пример 1. Работа с последовательностями

Название языка Lisp – это аббревиатура «list processing» – обработка списков. Очень часто говорится, что все объекты Lisp – это списки. В диалекте Clojure списки обобщены в последовательности. В качестве первого примера рассмотрим следующую задачу программирования.

Рассмотрим все натуральные числа меньше 10, которые делятся на 3 или на 5 – это числа 3, 5, 6 и 9. В сумме они дают 23. Теперь найдем сумму всех чисел меньше 1000, которые кратны 3 или 5.

Эта задача опубликована на сайте Project Euler. Проект Project Euler содержит коллекцию математических проблем, которые могут быть решены с использованием хорошо продуманных (или не очень хорошо продуманных) компьютерных программ. Рассматриваемая здесь задача – первая (и сама простая) в списке Project Euler. Листинг 3 предлагает ее решение на Clojure.

Листинг 3. Задача 1 из списка Project Euler

```
Первая строка листинга определяет функцию.
Запомните: функции — основные строительные блоки программы на Clojure.
Большинство Java-программистов привыкли к тому,
что базовыми элементами программы являются объекты,
так что, возможно, вам понадобится некоторое время,
чтобы приспособиться к разработке программ,
основными элементами которых являются функции.
Вы можете подумать, что defn — это служебное слово языка Clojure,
но на самом деле это макрос.
Макросы позволяют расширить функции компилятора Clojure
 - в основном путем добавления новых ключевых слов.
  Таким образом, defn не определяется спецификацией языка,
  а добавляется при подключении основной библиотеки Clojure.
```

В нашем случае макрос defn определяет функцию `divisible-by-3-or-5?`. Имя функции выбрано в соответствии с конвенцией выбора имен в Clojure. Слова разделяются дефисами, а в конце имени функции стоит вопросительный знак, который означает, что функция возвращает логическое значение «ложь» или «истина». Функции передается единственный аргумент `num`. Если бы нам потребовалось передать функции больше одного параметра, все параметры надо было бы перечислить в квадратных скобках, разделяя их пробелами.

За объявлением следует тело функции. Сначала мы вызываем функцию `or`. Это обычное логическое выражение `or` с той лишь разницей, что это не оператор, а функция. Мы передаем функции or два параметра. Каждый параметр, в свою очередь, тоже является выражением. Первое выражение начинается с функции `==`. Это функция, которая сравнивает значения своих аргументов. В нашем примере функции сравнения передаются 2 параметра. Первый – опять-таки выражение, которое вызывает функцию mod. Это обычное математическое деление по модулю, соответствующее оператору `%` в Java. Функция `mod` возвращает остаток от деления, т.е. в нашем случае, остаток от деления числа `num` на 3. Затем этот остаток сравнивается с 0. Если остаток равен 0, то число `num` делится на 3. Аналогично мы проверяем, делится ли число `num` на 5. Если остаток от деления на 3 или на 5 равен 0, то функция divisible-by-3-or-5? возвращает значение «истина».

В следующей строчке мы вычисляем выражение и выводим его на печать. Начнем разбор выражения с самых внутренних скобок. Здесь мы вызываем функцию range и передаем ей число 1000. Функция создает последовательность целых чисел от 0 до 999 включительно. Это именно те числа, чью делимость на 3 и 5 мы должны проверить. Далее переходим к следующему уровню вложенности. Здесь мы вызываем функцию `filter` с двумя аргументами. Первый аргумент должен быть логическим выражением, принимающим значение «истина» или «ложь», а второй аргумент — последовательностью. В нашем примере это последовательность `(0, 1, 2, ... 999)`. Функция `filter` вычисляет логическое выражение, и если оно истинно, соответствующий элемент последовательности добавляется к результату. Логическое выражение – это функция `divisible-by-3-or-5?`, которую мы подробно разобрали выше.

Таким образом, результатом работы функции filter будет последовательность целых чисел меньше 1000, кратных 3 или 5. Это именно те числа, которые нам нужно найти. Теперь для решения задачи остается только получить сумму этих чисел. Для этого мы используем функцию reduce. Эта функция использует 2 параметра: функцию-аргумент и последовательность. reduce применяет функцию-аргумент к первым двум элементам последовательности, а затем проходит по последовательности, каждый раз применяя функцию-аргумент к предыдущему полученному результату и следующему элементу последовательности. В нашем примере в качестве функции-аргумента выступает функция +, то есть обычное сложение. Таким образам, на выходе мы получаем сумму всех элементов последовательности.

Взгляните на листинг 3 - просто поразительно, сколько действий выполняет такой крошечный фрагмент. Однако наше пространное описание работы кода не должно вас пугать. Как только вы освоитесь с префиксной записью, код станет для вас вполне очевидным. Безусловно, если использовать Java для решения этой же задачи, программа будет гораздо длиннее. Перейдем к следующему примеру.

Пример 2. Лень - это добродетель

Следующий пример поможет нам освоить использование рекурсии и «ленивых» (lazy) вычислений в Clojure. Концепция рекурсии и «ленивых» вычислений, возможно, также окажется новой для большинства Java-программистов. Clojure позволяет определять «ленивые» последовательности, элементы которых вычисляются только тогда, когда это необходимо. Это дает возможность использовать бесконечные последовательности, что невозможно в Java. Чтобы разобраться, где и как именно применяются «ленивые» последовательности, рассмотрим задачу, решение которой основано на использовании еще одного важного аспекта функционального программирования: рекурсии. Вновь обратимся к коллекции Project Euler; на этот раз возьмем задачу №2.

Каждый новый член последовательности Фибоначчи является суммой двух предыдущих членов последовательности. Если мы начнем ряд с чисел 1 и 2, то получим следующие десять первых элементов ряда: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...

Найдите сумму всех четных чисел последовательности Фибоначчи, которые не превышают 4 миллионов. Для решения этой проблемы Java-программист, скорее всего, определил бы функцию, которая вычисляет *n*-й элемент последовательности Фибоначчи. Несколько наивная реализация такой функции приведена в листинге 4.

Листинг 4. Наивная функция Фибоначчи

```
(defn fib [n]
    (if (= n 0) 0
        (if (= n 1) 1
            (+ (fib (- n 1)) (fib (- n 2))))))
```

проверяет, равно ли *n* 1. Если это так, то функция возвращает 1. Для всех остальных значений n функция вычисляет (*n-1*)-й и (*n-2*)-й элементы последовательности Фибоначчи и складывает их. Все это выглядит абсолютно правильным, однако если у вас есть достаточно большой опыт программирования на Java, то вы уже нашли проблему. Рекурсивное определение в приведенном выше примере очень быстро заполнит весь стек и приведет к переполнению.

Числа Фибоначчи образуют бесконечную последовательность, и, будучи таковыми, должны определяться как бесконечная «ленивая» последовательность Clojure. Такое определение показано в листинге 5. Здесь необходимо заметить, что, стандартная библиотека clojure-contrib включает более эффективное определение последовательности Фибоначчи, однако библиотечная функция является довольно сложной для понимания, так что мы в данной статье воспользуемся примером из книги Стюарта Хэлловея (Stuart Halloway) - подробную информацию можно найти в разделе Ресурсы.

Листинг 5. «Ленивая» последовательность для чисел Фибоначчи

```
(defn lazy-seq-fibo
    ([]
        (concat [0 1] (lazy-seq-fibo 0 1)))
    ([a b]
        (let [n (+ a b)]
            (lazy-seq
                (cons n (lazy-seq-fibo b n))))))
```

В листинге 5 функция `lazy-seq-fibo` определяется дважды. Первое определение не имеет аргументов, отсюда пустые квадратные скобки. Второе определение использует два аргумента `[a b]`. Для безаргументного определения мы используем последовательность `[0 1]` и преобразовываем ее в выражение. Выражением является рекурсивный вызов функции, но при этом вызывается уже функция с аргументами 0 и 1.

Определение функции с двумя аргументами начинается с выражения `let`. Это способ задания переменных в Clojure. Выражение `[n (+ a b)]` определяет переменную `n` и задает ее значение, равное `a+b`. Затем используется макрос `lazy-seq`. Как следует из имени макроса, он создает «ленивую» последовательность. Телом макроса является выражение. В нашем примере, выражение обращается к функции `cons`. Это классическая функция Lisp. Функция использует элемент последовательности и саму последовательность и возвращает новую последовательность, добавляя элемент к началу последовательности. В нашем случае, последовательностью является результат вызова функции `lazy-seq-fibo`. Если бы последовательность не была бы ленивой, функция lazy-seq-fibo вызывалась бы снова и снова. Однако благодаря макросу `lazy-seq` функция `lazy-seq-fibo` будет вызываться только тогда, когда элементы последовательности используются. Для того чтобы увидеть, как работает такая последовательность, воспользуйтесь интерактивным режимом REPL, как показано в листинге 6.

Листинг 6. Генерация чисел Фибоначчи

```
1:1 user=> (defn lazy-seq-fibo
    ([]
        (concat [0 1] (lazy-seq-fibo 0 1)))
    ([a b]
        (let [n (+ a b)]
            (lazy-seq
                (cons n (lazy-seq-fibo b n))))))
#'user/lazy-seq-fibo
1:8 user=> (take 10 (lazy-seq-fibo))
(0 1 1 2 3 5 8 13 21 34)
```

Функция `take` используется для выбора определенного числа элементов последовательности (в нашем примере это 10). Теперь, когда у нас есть отличный метод вычисления последовательности Фибоначчи, мы можем решить исходную задачу.

Листинг 7. Задача 2

```
(defn less-than-four-million? [n] (< n 4000000))

(println (reduce +
    (filter even?
        (take-while less-than-four-million? (lazy-seq-fibo)))))
```

В листинге 7 мы определяем функцию `less-than-four-million?`. Это простая проверка того, что параметр функции меньше 4 миллионов. Разбор следующего выражения лучше начать с самых внутренних скобок. Сначала мы получаем бесконечную последовательность чисел Фибоначчи. Затем мы используем функцию `take-while`. Это аналог функции `take`, который в качестве аргумента использует не конкретное число, а условное выражение. Функция `take-while` берет элементы из последовательности до тех пор, пока условие не становится ложным. Таким образом, как только число из последовательности Фибоначчи становится больше 4 миллионов, мы перестаем вычислять элементы последовательности. Далее полученную последовательность мы передаем функции `filter`. В качестве фильтра используется встроенная функция `even?` , которая делает именно то, что вы подумали: проверяет, является ли число четным. Результирующая последовательность содержит четные числа Фибоначчи, меньшие 4 миллионов. Затем мы суммируем все эти числа, используя функцию `reduce`, так же, как мы это делали в предыдущем примере.

Код, приведенный в листинге 7, решает поставленную задачу, но он еще далек от совершенства. Для использования функции `take-while` нам пришлось определить совсем простую функцию`less-than-four-million?`. Однако делать это было совсем не обязательно. Из самого названия языка Clojure понятно, что он поддерживает замыкания (closure). Упрощенный код показан в листинге 8.

Замыкания в Clojure

Замыкания часто используются во многих языках программирования, особенно в функциональных языках, таких как Clojure. Помимо того, что функции являются основными элементами программы и могут передаваться в качестве аргументов другим функциям, они еще могут быть встроенными или анонимными. Листинг 8 демонстрирует упрощенный код листинга 7 с использованием замыкания.

Листинг 8. Упрощенный код

```
(println (reduce +
    (filter even?
        (take-while (fn [n] (< n 4000000)) (lazy-seq-fibo)))))
```

В листинге 8 мы используем макрос `fn`. Он создает анонимную функцию и возвращает ее значение. Условные функции, как правило, очень просты, и поэтому их лучше определять с использованием замыкания. Как мы увидим далее, Clojure предоставляет очень удобный способ записи замыканий.

Листинг 9. Краткая запись замыкания

```
(println (reduce +
    (filter even?
        (take-while #(< % 4000000) (lazy-seq-fibo)))))
```

В листинге 9 для создания замыкания вместо макроса `fn` мы воспользовались `#`. Кроме того, мы использовали символ `%` для обозначения первого параметра функции. Вы также можете использовать `%1` для первого параметра и `%2`, `%3` и т.д. в случае, если функция принимает несколько параметров.

На примере двух несложных задач мы рассмотрели множество свойств Clojure. Еще одним из важных качеств Clojure является его тесная интеграция с Java. Рассмотрим пример, наглядно демонстрирующий, насколько полезна может быть подобная интеграция при использовании Clojure.

Пример 3: Использование Java-технологий

Платформа Java обладает целым рядом неоспоримых преимуществ. Основными ее достоинствами являются высокая производительность JVM и богатейший набор функций, поставляемых как в основных API, так и в виде библиотек, созданных независимыми разработчиками на Java. Таким образом, Java предоставляет вам возможность использовать готовые функции вместо того, чтобы многократно заново изобретать велосипед. Clojure был создан с учетом возможности использования Java. Вы можете вызвать Java-методы, создавать Java-объекты, использовать Java-интерфейсы и расширения Java-классов. Для наглядной демонстрации возможностей использования Java решим еще одну задачу из коллекции Project Euler.

Листинг 10. Задача №8 из коллекции Project Euler

```
Найти наибольшее произведение пяти последовательных цифр в записи
1000-знакового числа.
```

731671765313306249192251196744265747423553491949349698352031277450632623957831801698480186947885184385861560789112949495459501737958331952853208805511125406987471585238630507156932909632952274430435576689664895044524452316173185640309871112172238311362229893423380308135336276614282806444486645238749303589072962904915604407723907138105158593079608667017242712188399879790879227492190169972088809377665727333001053367881220235421809751254540594752243525849077116705560136048395864467063244157221553975369781797784617406495514929086256932197846862248283972241375657056057490261407972968652414535100474821663704844031998900088952434506585412275886668811642717147992444292823086346567481391912316282458617866458359124566529476545682848912883142607690042242190226710556263211111093705442175069416589604080719840385096245544436298123098787992724428490918884580156166097919133875499200524063689912560717606058861164671094050775410022569831552000559357297257163626956188267042825248360082325753042075296345

В этом примере у нас есть число, в записи которого использовано 1000 знаков. Такое число может быть представлено в Java как `BigInteger`. Однако нам нужно не значение этого числа, а всего лишь каждые пять последовательных цифр в его записи. Таким образом, нам удобнее рассматривать это число не как числовое значение, а как строку. Тем не менее для дальнейших вычислений мы должны рассматривать цифры в записи числа как целые числа. К счастью, Java API позволяет преобразовывать строки в числа и обратно. Начнем с того, что разберемся с большим куском неупорядоченного текста, который представляет собой данная выше запись числа.

Листинг 11. Разбор текста

```
(def big-num-str
    (str "731671765313306249192251196744265747423553491949349698352031277450632623957831801698480186947885184385861560789112949495459501737958331952853208805511254069874715852386305071569329096329522744304355766896648950445244523161731856403098711121722383113622298934233803081353362766142828064444866452387493035890729629049156044077239071381051585930796086670172427121883998797908792274921901699720888093776657273330010533678812202354218097512545405947522435258490771167055601360483958644670632441572215539753697817977846174064955149290862569321978468622482839722413756570560574902614079729686524145351004748216637048440319989000889524345065854122758866688116427171479924442928230863465674813919123162824586178664583591245665294765456828489128831426076900422421902267105562632111110937054421750694165896040807198403850962455444362981230987879927244284909188845801561660979191338754992005240636899125607176060588611646710940507754100225698315520005593572972571636269561882670428252483600823257530420752963450"))
```

Здесь мы воспользовались реализацией многострочных текстовых переменных в Clojure. Функция `str` выполняет разбор многострочной текстовой константы. Затем мы используем макрос `def` для определения текстовой константы `big-num-str`. Однако, для решения нашей задачи самое удобное – это преобразовать эту константу в последовательность целых чисел. Это преобразование показано в листинге 12.

Листинг 12. Создание последовательности целых чисел

```
(def the-digits
    (map #(Integer. (str %))
        (filter #(Character/isDigit %) (seq big-num-str))))
```

Вновь начнем разбор кода с самых внутренних скобок. Функция `seq` используется для преобразования `big-num-str` в последовательность. Однако оказывается, что эта последовательность – не совсем то, что нам нужно. С помощью REPL вы и сами легко можете в этом убедиться:

Листинг 13. Проверка последовательности `big-num-str`

```
user=> (seq big-num-str)
(\7 \3 \1 \6 \7 \1 \7 \6 \5 \3 \1 \3 \3 \0 \6 \2 \4 \9 \1 \9 \2 \2 \
5 \1 \1 \9
 \6 \7 \4 \4 \2 \6 \5 \7 \4 \7 \4 \2 \3 \5 \5 \3 \4 \9 \1 \9 \4 \9 \
3 \4
 \newline...
```

REPL показывает цифры в записи числа (Java char) как `\c`. Таким образом, цифре 7 в записи числа соответствует элемент `\7`, а сочетанию \n (переход на новую строку) соответствует элемент `\newline`. Именно такую последовательность мы и получим при непосредственном преобразовании текста. Таким образом, прежде чем приступать к каким-либо вычислениям, нам надо избавиться от элементов, соответствующих переходу на новую строку, и преобразовать цифры в целые числа. Эти дополнительные действия приведены в листинге 12. Чтобы избавиться от символов новой строки, мы накладываем фильтр. Обратите внимание, что в коде опять используется краткая запись условной функции, передаваемой `filter` в качестве аргумента. Функция-замыкание использует `Character/isDigit`. Это static-метод `isDigit` из класса `java.lang.Character`. Таким образом, наш фильтр пропустит цифры и отфильтрует символы новой строки.

Теперь, когда мы избавились от символов новой строки, можно приступать к преобразованию цифр в целые числа. Рассмотрим следующий уровень вложенности фрагмента, приведенного в листинге 12. Здесь используется функция map с двумя параметрами: функцией-аргументом и последовательностью `map` возвращает новую последовательность, *n*-ный элемент которой есть результат применения функции-аргумента к *n*-ному члену исходной последовательности. В качестве функции-аргумента мы вновь используем краткую запись замыкания. Сначала мы вызываем функцию Clojure `str` для преобразования символа в строку. Зачем нам это? - спросите вы. Затем, что потом мы преобразуем эту строку в целое число, используя конструктор для `java.lang.Integer`. В листинге этому соответствует `Integer`. Вы можете рассматривать это выражение как `java.lang.Integer(str(%))` . Объединив такое преобразование с функцией `map`, мы получаем необходимую последовательность целых чисел. Теперь мы можем решить исходную задачу.

Листинг 14. Задача 3

```
(println (apply max
    (map #(reduce * %)
        (for [idx (range (count the-digits))]
            (take 5 (drop idx the-digits))))))
```

Чтобы разобраться в этом фрагменте кода, начнем с макроса `for`. Этот не Java-цикл `for`, это последовательное включение. Сначала мы определяем привязку, используя квадратные скобки. В нашем примере переменная `idx` привязана к последовательности 0… N-1, где *N* – количество элементов в последовательности `the-digits` (N=1000, так как в исходном числе 1000 знаков). Затем в макросе `for` определяется выражение, по которому будет создаваться новая последовательность. Для каждого элемента в последовательности `idx` макрос вычислит заданное выражение и добавит результат к новой последовательности. Как можно заметить, в определенном смысле макрос for работает аналогично циклу `for`. Выражение, используемое для вычисления элементов новой последовательности, сначала обращается к функции `drop` для того, чтобы отбросить первые *M* элементов последовательности, а затем вызывает функцию `take` для того чтобы отобрать первые пять элементов из оставшейся последовательности. Следует помнить, что *M* будет принимать значения 0, 1, 2, и т.д., так что результатом работы будет последовательность наборов по пять чисел в каждом. Первый элемент будет иметь вид (e1, e2, e3, e4, e5), второй элемент будет иметь вид (e2, e3, e4, e5, e6), и т.д., где e1, e2, и т.д. - элементы `the-digits`.

Теперь применим функцию `map` к набору последовательностей из 5 элементов. Функция-аргумент `reduce` преобразует последовательность из 5 элементов в произведение этих элементов, так что на выходе функции мы получим последовательность целых чисел, первое из которых является произведением элементов 1-5, второе – произведением элементов 2-6, и т.п. Нам нужно найти наибольшее из таких произведений. Для этого используется функция `max`. Однако она работает с несколькими аргументами, а не с одной последовательностью. Для преобразования последовательности в набор параметров, используется функция `apply`. Таким образом, мы находим требуемый максимум и выводим его на печать. Мы с вами разобрали несколько задач и написали несколько программ на Clojure для их решения, одновременно потренировавшись в применении языка.