ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

Лекция 8. Введение в Clojure.

Почему Clojure?

Clojure – язык программирования, соответствующий этому стандарту. Заимствовавший самое лучшее из нескольких языков программирования – включая различные реализации Lisp, а также Ruby, Python, Java, Haskell и других – Clojure обладает множеством особенностей, пригодных для решения самых сложных задач, с которыми приходится сталкиваться программистам в наши дни, и которые уже маячат на горизонте. А отсутствие необходимости переходить на совершенно новую, незнакомую платформу и среду выполнения (что обычно присуще многим другим языкам, появившимся в последние годы), так как Clojure выполняется под управлением виртуальной машины Java, нередко перевешивает все прагматические доводы и проблему наличия унаследованного кода в пользу выбора нового языка программирования.

Clojure выполняется под управлением JVM

В программах на языке Clojure можно использовать любые Java-библиотеки. Библиотеки на языке Clojure, в свою очередь, можно использовать в программах на Java. Приложения на языке Clojure могут упаковываться подобно любым Java-приложениям и разворачиваться везде, где могут разворачиваться другие приложения на Java: на серверах веб-приложений; на персональных компьютерах с интерфейсом Swing, SWT или командной строки; и так далее. Это также означает, что среда выполнения Clojure также является средой выполнения Java, одной из самых эффективных и надежных в мире.

Clojure – язык функционального программирования

Язык Clojure поощряет использование функций, как объектов первого рода, а также функции высшего порядка, и предлагает собственный набор эффективных неизменяемых типов данных. Clojure фокусируется на функциональной парадигме, что устраняет типичные ошибки и недостатки, связанные с беспрепятственным изменением значений переменных. Кроме того, функциональный стиль значительно упрощает разработку параллельных программ.

Переменные в языке Clojure не являются переменными в классическом понимании

Прежде чем продолжить знакомство с языком программирования Clojure, необходимо внести дополнительную ясность, касающуюся термина «переменные».

В языке Clojure отсутствуют переменные в том понимании, в каком они существуют в императивных языках программирования, таких как C/C++, Java или Python. Их место занимают «переменные», своим поведением больше напоминающие константы.

Переменная в Clojure – это единовременная привязка значения к его имени (если точнее – к «символу» Clojure). Однажды привязав, скажем, значение 1 к символу `a`, изменить эту привязку, назначив значение 2 символу `a`, уже нельзя.

Тем не менее, в Clojure существуют специальные типы данных, позволяющие изменять значения внутри себя. Например, привязав объект типа `ref` к символу `b`, мы не сможем в дальнейшем изменить эту привязку. Но сам объект типа `ref` сможет изменять свое содержимое. Изменение (мутация) данных внутри таких объектов может происходить только в транзакциях. Как правило, начинающие Clojure-программисты часто используют этот механизм и применяют много изменяемых данных. Но настоящий путь Clojure состоит в максимально полном отказе от приемов, основанных на изменении переменных, настолько, насколько это вообще возможно. Для обозначения описанных выше неизменяемых переменных создателями языка Clojure был даже придуман специальный термин: «var» (вероятно сокращение от «variable» – переменная), не имеющий аналогов в русском языке. По этой причине далее в книге под термином «переменная» будут подразумеваться переменные Clojure (vars), а для обозначения классических переменных будет использоваться термин «изменяемая переменная».

Clojure предлагает современные решения проблем, свойственных выполнению в многопоточной среде

Для эффективного использования многоядерных процессоров, многопроцессорных систем и систем распределенных вычислений нужно использовать языки и библиотеки, разрабатывавшиеся с учетом новых возможностей. Ссылочные типы в языке Clojure обеспечивают четкое разделение *состояния* (state) и *идентичности* (identity). Данная семантика упрощает реализацию параллельных вычислений, избавляя от необходимости использовать ручное управление блокировками, подобно тому как автоматическая сборка мусора облегчает управление памятью.

Clojure – динамический язык программирования

Clojure – динамический и строго типизированный язык программирования (и этим он напоминает языки Python и Ruby), однако вызовы функций компилируются в (быстрые!) вызовы Java-методов. Clojure является динамичным еще и в том смысле, что поддерживает возможность обновления и загрузки нового программного кода, локального или удаленного, прямо во время выполнения. Это особенно удобно для интерактивной разработки и отладки или даже для расширения и исправления удаленных приложений без их остановки.

Реализации многих языков включают интерактивную командную оболочку REPL, которую часто называют интерпретатором: в Ruby имеется `irb`; в Python имеется свой интерпретатор командной строки; в Groovy имеется консоль; даже в Java имеется нечто похожее на REPL в BeanShell. Аббревиатура «REPL» получена из простого описания принципа функционирования оболочки:

1. Read (прочитать): выполняется чтение программного кода из некоторого источника (обычно `stdin`, но может быть иной, например при использовании REPL в IDE).
2. Eval (вычислить): программный код выполняется, возвращая некоторое значение.
3. Print (вывести): значение передается в некоторый поток вывода (обычно `stdout`, но перед результатом, самим программным кодом может быть выведена дополнительная информация).
4. Loop (повторить): управление возвращается к шагу чтения (Read).

Пример 1.1. Запуск Clojure REPL из командной строки

```
% java -cp clojure-1.4.0.jar clojure.main
Clojure 1.4.0
user=>
```

Когда на экране появится приглашение к вводу `user=>`, оболочка REPL будет готова к вводу программного кода на Clojure. Часть строки приглашения в Clojure REPL, предшествующая `=>`, является именем *текущего пространства имен*. Пространства имен похожи на модули или пакеты; мы подробно будем обсуждать их далее в этой

Пример 1.2. Вычисление среднего арифметического значения в Java, Ruby и Python

```
public static double average (double[] numbers) {
     double sum = 0;
     for (int i = 0; i < numbers.length; i++) {
          sum += numbers[i];
     }
     return sum / numbers.length;
}

def average (numbers)
     numbers.inject(:+) / numbers.length
end

def average (numbers):
     return sum(numbers) / len(numbers)
```

Ниже приводится эквивалентная функция на языке Clojure:

```
(defn average                                    ❶
    [numbers]                                    ❷
    (/ (apply + numbers) (count numbers)))       ❸
```

❶ Ключевое слово `defn` определяет в текущем пространстве имен новую функцию с именем `average`.

❷ Функция `average` принимает один аргумент, на который в теле функции можно ссылаться по имени `numbers`. Обратите внимание на отсутствие объявления типа – эта функция одинаково хорошо может обрабатывать любые коллекции или массивы чисел любых типов.

❸ В теле функции `average` вычисляется сумма переданных ей чисел `(apply + numbers)`[1], затем сумма делится на количество чисел `(count numbers)` и результат деления возвращается.

Выражение `defn` можно ввести в интерактивной оболочке REPL, затем вызвать функцию, передав ей вектор чисел, и получить ожидаемый результат:

```
user=> (defn average
           [numbers]
           (/ (apply + numbers) (count numbers)))
#'user/average
user=> (average [60 80 100 400])
160
```

Выражения, операторы, синтаксис и очередность

Весь программный код на Clojure состоит из выражений, каждое из которых возвращает единственное значение. Этим он отличается от многих других языков, имеющих инструкции, не возвращающие значений, такие как `if`, `for` и `continue`, и используемые для управления потоком выполнения программы. В Clojure все эти языковые конструкции являются *выражениями*, возвращающими некоторое значение.

Выше уже было показано несколько примеров выражений на языке Clojure:

- `60`;
- `[60 80 100 400]`;
- `(average [60 80 100 400])`;
- `(+ 1 2)`.

Все эти выражения возвращают единственное значение. Правила их вычисления чрезвычайно просты, в сравнении с другими языками программирования:

1. Списки (ограничиваются круглыми скобками) являются вызовами. Первое значение в списке интерпретируется как оператор, а остальные – как параметры. Первый элемент списка часто называют *позицией функции* (function position), потому что в нем передается функция или символ, указывающий на функцию для вызова. Выражение вызова получает значение, возвращаемое функцией.
2. Символы (такие как `average` или `+`) преобразуются в именованное значение, находящееся в текущей области видимости. Это значение может быть функцией, локальными данными (как вектор `numbers` в функции `average`), Java-классом, макросом или специальной формой. О макросах и специальных формах мы поговорим чуть ниже, а пока просто считайте их функциями.
3. Все остальные выражения возвращают буквальные значения, которые они описывают.

Таблица 1.1. Сравнение синтаксиса вызовов функций в Clojure, Java, Python и Ruby

Выражение на языке Clojure	Эквивалент на языке Java	Эквивалент на языке Python	Эквивалент на языке Ruby
`(not k)`	`!k`	`not k`	`not k` или `! k`
`(inc a)`	`a++`, `++a`, `a += 1`, `a + 1`[a]	`a += 1`, `a + 1`	`a += 1`
`(/ (+ x y) 2)`	`(x + y) / 2`	`(x + y) / 2`	`(x + y) / 2`
`(instance? java.util.List al)`	`al instanceof java.util.List`	`isinstance(al, list)`	`al.is_a? Array`
`(if (not a) (inc b) (dec b))`[b]	`!a ? b + 1 : b - 1`	`b + 1 if not a else b - 1`	`!a ? b + 1 : b - 1`
`(Math/pow 2 10)`[c]	`Math.pow(2, 10)`	`pow(2, 10)`	`2 ** 10`
`(.someMethod someObj "foo" (.otherMethod otherObj 0))`	`someObj.someMethod ("foo", otherObj. otherMethod(0))`	`someObj. someMethod ("foo", otherObj. otherMethod(0))`	`someObj.someMethod ("foo", otherObj. otherMethod(0))`

Гомоиконность

Формат записи выполняемого кода совпадает с форматом записи данных. Эта особенность формально называется *гомоиконностью* (homoiconicity), или неформально – *код-это-данные*[2]. Такое существенное упрощение в сравнении с другими языками обеспечивает более широкие возможности метапрограммирования, чем в негомоиконных языках. Чтобы понять причину, необходимо немного поговорить о языках вообще и о том, как их программный код соотносится с внутренним его представлением.

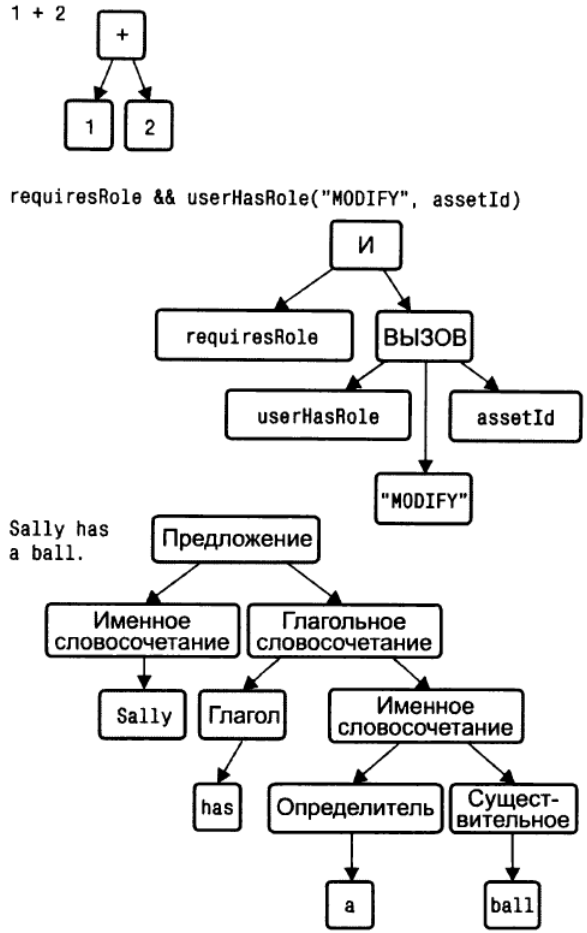

Рис. 1.1. Примеры преобразования исходных текстов в формальные модели

```
(and requiresRole (userHasRole "MODIFY" assetId))
```

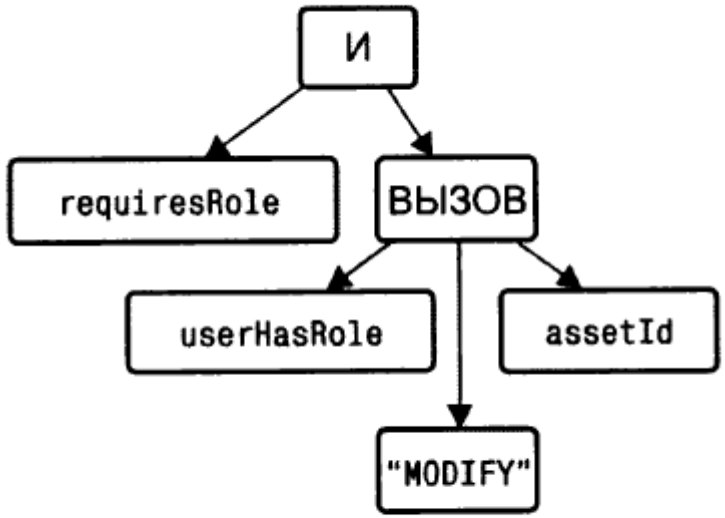

Рис. 1.2. Результат преобразования выражения в дерево АСТ

С практической точки зрения прямое соответствие между кодом и данными означает, что программный код на языке Clojure, который вводится в интерактивной оболочке REPL или присутствующий в исходном файле, строго говоря не является текстом: программирование выполняется с применением литералов структур данных языка Clojure. Вспомним простую функцию вычисления среднего арифметического из примера 1.2:

```
(defn average
    [numbers]
    (/ (apply + numbers) (count numbers)))
```

Это не просто фрагмент текста, который волшебным образом, преобразуется в определение функции. Это – структура данных списка, содержащая четыре значения: символ `defn`, символ `average`, вектор, содержащий символ `numbers`, и еще один список, описывающий тело функции. После обработки этого списка будет определена функция `average`.

Механизм чтения

Хотя компиляция и выполнение программ происходит с использованием структур данных языка Clojure, исходный программный код все еще принято хранить в виде текстовых файлов. Поэтому, необходим некоторый способ получения структур данных из исходного текста. Эта задача решается *механизмом чтения* языка Clojure.

Действие механизма чтения полностью определяется единственной функцией `read`, которая читает текстовое содержимое из потока

символов[1] и возвращает структуру данных, закодированную в нем. Именно этот механизм используется интерактивной оболочкой Clojure REPL для чтения ввода. Каждая законченная структура данных затем передается для обработки среде выполнения Clojure.

Для целей исследования удобнее использовать функцию `read-string`, которая действует так же как `read`, но данными для нее служит строковый аргумент:

```
(read-string "42")
;= 42
(read-string "(+ 1 2)")
;= (+ 1 2)
```

Механизм чтения по сути выполняет *десериализацию* (deserialization). Структуры данных и другие литералы имеют в языке Clojure определенные текстовые представления, которые механизм чтения десериализует в соответствующие значения и структуры данных.

Возможно, вы обратили внимание, что значения в REPL выводятся точно так же, как были введены: числа и другие атомарные литералы выводятся в ожидаемом формате, списки заключены в круглые скобки, векторы – в квадратные скобки, и так далее. Это объясняется наличием функций `pr` и `pr-str`, парных функциям `read` и `read-string`, первая из которых выводит значения Clojure в текстовом представлении в `*out*`[2], а вторая возвращает строку с текстовым представлением, которое в свою очередь можно передать функции `read`. Поэтому, структуры данных и значения в языке Clojure легко сериализуются и десериализуются в представления, понятные человеку и механизму чтения:

```
(pr-str [1 2 3])
;= "[1 2 3]"
(read-string "[1 2 3]")
;= [1 2 3]
```

Скалярные литералы

Скалярные литералы – это синтаксис представления значений, не являющихся коллекциями. Большинство из них являются значениями распространенных типов, имеющихся в Java или аналогичных в Ruby, Python и других языках. Другие характерны для языка Clojure и несут новую семантику.

Строки

Строки в языке Clojure являются обычными Java-строками (то есть, экземплярами класса `java.lang.String`) и в текстовом представлении точно так же заключаются в двойные кавычки:

```
"hello there"
;= "hello there"
```

Строковые литералы в языке Clojure могут занимать несколько строк, без использования особого синтаксиса (как, например, в Python):

```
"multiline strings
are very handy"
;= "multiline strings\nare very handy"
```

Логические значения

Для обозначения логических значений в языке Clojure используются лексемы `true` и `false`, так же как в Java, Ruby и Python (в последнем из них эти лексемы записываются, начиная с символа верхнего регистра).

nil

Значение `nil` в Clojure соответствует значению `null` в Java, `nil` в Ruby и `None` в Python. Кроме того, в условных инструкциях значение `nil` в языке Clojure интерпретируется как `false`, точно так же, как в языках Ruby и Python.

Знаки (characters)

Символьные литералы начинаются с обратного слеша:

```
(class \c)
;= java.lang.Character
```

Символьные литералы могут также записываться в формате Юникода и восьмеричной форме:

```
\u00ff
;= \ÿ
\o41
;= \!
```

Кроме того, существует несколько символьных литералов со специальными именами для обозначения часто используемых символов, которые выводятся как пробельные:

- `\space`;
- `\newline`;
- `\formfeed`;
- `\return`;
- `\backspace`;
- `\tab`.

Ключевые слова (keywords)

Ключевые слова обозначают сами себя и часто используются как средство доступа к значениям в коллекциях и типах, таких как ассоциативные массивы и записи:

```
(def person {:name "Sandra Cruz"
             :city "Portland, ME"})
;= #'user/person
(:city person)
;= "Portland, ME"
```

Здесь создается ассоциативный массив с двумя элементами, `:name` и `:city`, и затем выполняется поиск по ключу `:city`. Такое возможно, потому что ключевые слова – это функции, отыскивающие сами себя в коллекциях, передаваемых в аргументе.

Символы (symbols)

Если ключевые слова – это идентификаторы в пределах коллекций (таких как ассоциативные массивы и записи), то *символы* – это идентификаторы, действующие в пределах всей среды выполнения Clojure. В их число входят переменные (vars), которые являются именованными блоками памяти для хранения функций и данных, Java-классы, локальные ссылки и так далее. Вернемся к предыдущему примеру 1.2:

```
(average [60 80 100 400])
;= 160
```

`average` – это символ, ссылка на функцию, хранящаяся в переменной с именем `average`.

Символы должны начинаться с нечислового знака и помимо букв и цифр могут содержать `*`, `+`, `!`, `-`, `_` и `?`. Символы, содержащие слеш (`/`), обозначают *символы с пространством имен* (namespaced symbols) и соответствуют именованным значениям в указанном пространстве имен. Результат интерпретации символов в сущности, на которые они ссылаются, зависит от контекста вычислений и пространств имен, доступных внутри этого контекста. Подробнее о семантике пространств имен и интерпретации символов рассказывается в разделе «Пространства имен» ниже.

Числа

В языке Clojure поддерживается множество числовых литералов (табл. 1.2). Многие из них весьма обычны, но есть и такие, которые редко встречаются в языках программирования общего назначения и применение которых может упростить реализацию отдельных алгоритмов, особенно когда алгоритмы определяются в терминах определенных способов представления чисел (восьмеричные и двоичные числа, рациональные числа и числа в экспоненциальном представлении).

Таблица 1.2. Числовые литералы в языке Clojure

Литерал	Числовой тип
`42, 0xff, 2r111, 040`	длинное целое (64-разрядное целое со знаком)
`3.14, 6.0221415e23`	вещественное двойной точности (64-разрядное вещественное с плавающей точкой в формате IEEE)
`42N`	`clojure.lang.BigInt` (целое число произвольной точности[a])
`0.01M`	`java.math.BigDecimal` (вещественное число произвольной точности)
`22/7`	`clojure.lang.Ratio`

Регулярные выражения

Строки, начинающиеся со знака «диез» (#), механизм чтения языка Clojure интерпретирует как литералы регулярных выражений:

```
(class #"(p|h)ail")
;= java.util.regex.Pattern
```

Синтаксис литералов регулярных выражений в Clojure в точности повторяет синтаксис /.../ регулярных выражений в Ruby, за исключением несущественных различий в символах-ограничителях. Фактически, реализации поддержки регулярных выражений в Ruby и Clojure очень близки:

```
# Ruby
>> "foo bar".match(/(...) (...)/).to_a
["foo bar", "foo", "bar"]

;; Clojure
(re-seq #"(...) (...)" "foo bar")
;= (["foo bar" "foo" "bar"])
```

Синтаксис регулярных выражений в Clojure не требует экранирования обратных слешей как в Java:

```
(re-seq #"(\d+)-(\d+)" "1-3") ;; превратится в "(\\d+)-(\\d+)" в Java
;= (["1-3" "1" "3"])
```

Комментарии

Механизм чтения различает два типа комментариев.

- Однострочные комментарии начинаются с точки с запятой (;). Все, что следует за точкой с запятой до конца строки, игнорируется механизмом чтения. Комментарии этого типа эквивалентны комментариям в Java и JavaScript, начинающимся с пары символов //, и комментариям в Ruby и Python, начинающимся с символа #.
- Комментарии *уровня формы* (form-level) определяются с помощью макроса #_. Он вынуждает механизм чтения игнорировать форму, следующую за макросом:

```
(read-string "(+ 1 2 #_(* 2 2) 8)")
;= (+ 1 2 8)
```

Список из четырех чисел – (+ 1 2 4 8) – превращается в список из трех чисел, потому что целая форма умножения игнорируется из-за префикса #_.

Литералы коллекций

Механизм чтения предусматривает синтаксис определения наиболее используемых структур данных в программах на языке Clojure:

```
'(a b :name 12.5)        ;; список
['a 'b :name 12.5]       ;; вектор
{:name "Chas" :age 31}   ;; ассоциативный массив
#{1 2 3}                 ;; множество
```

Поскольку списки в Clojure используются также для вызовов функций, чтобы предотвратить интерпретацию литералов списков как вызовов функций, их необходимо предварять апострофом (').

Одним из *ссылочных типов*[2] являются переменные (vars), которые в Clojure представляют хранилища значений любых *ссылочных* типов (`ref`, `atom`, `agent`). Они связаны с символами внутри пространства имен, которые другой код может использовать для их поиска и, соответственно, получения их значений.

Переменные определяются в языке Clojure с помощью специальной формы `def`, действие которой распространяется только на текущее пространство имен[3]. Теперь определим переменную `x` в пространстве имен `user`. Именем этого значения является символ, служащий ключом в текущем пространстве имен:

```
(def x 1)
;= #'user/x
```

Обратиться к значению можно по этому символу:

```
x
;= 1
```

Здесь используется *неквалифицированный* символ x, поэтому его поиск выполняется в текущем пространстве имен. Переменные можно переопределять, что очень важно для поддержки разработки в интерактивном режиме в REPL:

```
(def x "hello")
;= #'user/x
x
;= "hello"
```

Подавление вычислений: quote

Форма `quote` подавляет вычисление выражения. Самый доступный пример – символы, если они ссылаются на переменные, то интерпретируются в соответствующие значения. Применение `quote` подавляет этот процесс, поэтому символы преобразуются в самих себя (так же как строки, числа, и так далее):

```
(quote x)
;= x
(symbol? (quote x))
;= true
```

Механизм чтения предоставляет отдельный синтаксис для `quote` – символ апострофа, предшествующий любой форме, действует подобно `quote`:

```
'x
;= x
```

Подавлено может быть вычисление любой формы в языке Clojure, включая структуры данных. В этом случае возвращается структура данных, в которой рекурсивно подавляется вычисление всех ее элементов:

```
'(+ x x)
;= (+ x x)
(list? '(+ x x))
;= true
```

Списки обычно интерпретируются как вызовы функций, однако добавление `quote` перед списком подавляет его интерпретацию, и возвращает сам список, в данном случае – список из трех символов: `'+`, `'x` и `'x`. Обратите внимание, что это в точности то, что получается при конструировании списка «вручную», без использования литерала списка:

```
(list '+ 'x 'x)
;= (+ x x)
```

Блоки кода: do

Конструкция `do` вычисляет все выражения, переданные ей, и в качестве собственного значения возвращает значение последнего выражения. Например:

```
(do
     (println "hi")
     (apply * [4 5 6]))
; hi
;= 120
```

Значения всех выражений, кроме последнего, просто отбрасываются, однако побочные эффекты этих выражений (такие как вывод в стандартный поток вывода, как в данном примере, или изменение состояния объекта, доступного в текущей области видимости) продолжают действовать.

Обратите внимание, что многие другие формы (включая `fn`, `let`, `loop` и `try` – и их производные, такие как `defn`) неявно обертывают свое содержимое конструкцией `do`, чтобы обеспечить возможность вычисления нескольких внутренних выражений. Как, например, в следующей форме `let`, где определяются два локальных значения. Как, например, в следующей форме `let`, где определяются два локальных значения:

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
  (println (format "You rolled a %s and a %s" a b))
  (+ a b))
```

Эта особенность позволяет вычислять любое количество выражений в контексте формы `let`, последнее из которых определяет значение всей конструкции. Если бы конструкция `let` не обертывала свое тело формой `do`, ее необходимо было бы использовать явно[1]:

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
  (do
    (println (format "You rolled a %s and a %s" a b))
    (+ a b)))
```

Определение переменных: def

Мы уже видели форму `def` в действии[2]. Она определяет (или переопределяет) переменную в текущем пространстве имен:

```
(def p "foo")
;= #'user/p
p
;= "foo"
```

Многие другие формы неявно создают или переопределяют переменные и, как следствие, используют форму `def`. Это характерно для форм, начинающихся с префикса «def», таких как `defn`, `defn-`, `defprotocol`, `defonce`, `defmacro`, и так далее.

Связывание локальных значений: let

Форма `let` позволяет определять именованные ссылки, доступные только в этой форме. Например, следующий простейший статический метод на Java:

```
public static double hypot (double x, double y) {
     final double x2 = x * x;
     final double y2 = y * y;
     return Math.sqrt(x2 + y2);
}
```

эквивалентен следующей функции на Clojure:

```
(defn hypot
  [x y]
  (let [x2 (* x x)
        y2 (* y y)]
    (Math/sqrt (+ x2 y2))))
```

Здесь `x2` и `y2` являются локальными символами, доступными только в теле функции/метода и служат цели создания именованных ссылок с ограниченной областью видимости на промежуточные значения.

Обратите внимание, что форма `let` неявно используется везде, где требуются локальные символы. В частности `fn` (и, соответственно, все другие формы создания и определения функций, такие как `defn`) использует `let` для связывания параметров функции с локальными символами в области видимости определяемой функции. Например, символы `x` и `y` в функции `hypot` выше, являются let-связками (let-bound), создаваемыми формой `defn`. То есть, вектор, определяющий множество привязок для области видимости `let`, следует той же семантике, как для определения параметров функции, так и для определения вспомогательных локальных символов.

Деструктуризация (let, часть 2)

Программирование на языке Clojure в значительной степени связано с обработкой различных реализаций абстрактных структур данных, основными примерами которых могут служить *упорядоченные коллекции* (sequential collections) и *ассоциативные массивы* (maps). Многие функции в языке Clojure принимают и возвращают абстрактные коллекции, а не какие-то конкретные их реализации, и большинство библиотек Clojure и приложений созданы с опорой на эти абстракции, а не на конкретные структуры, классы, и так далее. Это помогает конструировать функции и библиотеки с минимумом служебного, «связующего» кода для работы с данными.

Одна из сложностей при работе с абстрактными коллекциями заключается в том, чтобы обеспечить лаконичность обращения к множественным значениям в этих коллекциях. Например, следующая коллекция является вектором Clojure:

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
```

Рассмотрим несколько способов доступа к значениям в этом векторе:

```
(first v)          ❶
;= 42
(second v)
;= "foo"
(last v)
;= [5 12]
(nth v 2)          ❷
;= 99.2
(v 2)              ❸
;= 99.2
(.get v 2)         ❹
;= 99.2
```

❶ Clojure предоставляет вспомогательные функции для доступа к первому (`first`), второму (`second`) и последнему (`last`) значениям в упорядоченной коллекции.
❷ Функция `nth` позволяет получить любое значение из упорядоченной коллекции, используя индекс в этой коллекции.
❸ Векторы являются функциями, принимающими индекс в виде аргумента.
❹ Все упорядоченные коллекции в языке Clojure реализуют интерфейс `java.util.List`, поэтому для доступа к их содержимому можно использовать метод `.get` интерфейса.

Все эти механизмы прекрасно подходят для получения доступа к единственному «верхнеуровнему» значению в векторе, но дело осложняется, когда при выполнении некоторых операций требуется получить доступ сразу к нескольким значениям:

```
(+ (first v) (v 2))
;= 141.2
```

или к значениям во вложенных коллекциях:

```
(+ (first v) (first (last v)))
;= 47
```

Деструктуризация упорядоченных коллекций

Деструктуризация упорядоченных коллекций (sequential destructuring) применяется к любым упорядоченным коллекциям, включая:

- списки, векторы и последовательности Clojure;
- любые коллекции, реализующие интерфейс `java.util.List` (подобно `ArrayLists` и `LinkedLists`);
- массивы Java;
- строковые значения, которые разлагаются на знаки.

Ниже приводится простой пример, где выполняется деструктуризация значения `v`, приведенного выше:

Пример 1.3. Простейшая деструктуризация упорядоченной коллекции

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
(let [[x y z] v]
     (+ x z))
;= 141.2
```

В простейшем случае вектор, передаваемый форме `let`, содержит пары имен и значений, но здесь вместо скалярного имени символа мы передаем вектор символов (`[x y z]`). Это вызывает деструктуризацию значения `v` в теле формы `let` и связывание первого значения с символом `x`, второго значения – с `y`, и третьего – с `z`. Затем эти локальные значения, полученные в результате деструктуризации, можно использовать как любые другие локальные значения. Следующая форма связывания эквивалентна предыдущей:

```
(let [x (nth v 0)
      y (nth v 1)
      z (nth v 2)]
  (+ x z))
;= 141.2
```

Формы деструктуризации отражают структуру связываемых коллекций[1]. То есть, применяя форму деструктуризации к коллекции можно получить точное представление о том, как значения будут связаны с именами[2]:

```
[x y z]
[42 "foo" 99.2 [5 12]]
```

Формы деструктуризации могут быть составными, благодаря чему легко можно извлечь значения из вектора, вложенного в v[3]:

```
(let [[x _ _ [y z]] v]
(+ x y z))
;= 59
```

Если визуально расположить форму деструктуризации в одну линию над вектором, результат ее действия станет более очевидным:

```
[x  _      _     [y z ]]
[42 "foo" 99.2 [5 12]]
```

Извлечение оставшихся значений из упорядоченных последовательностей

Чтобы извлечь все оставшиеся значения из упорядоченной коллекции, находящиеся за последней именованной позицией в форме деструктуризации, можно воспользоваться оператором `&`. По своему действию он напоминает механизм, лежащий в основе списков аргументов переменной длины (varargs) в методах Java, и является основой для извлечения оставшихся аргументов в функциях на языке Clojure:

```
(let [[x & rest] v]
rest)
;= ("foo" 99.2 [5 12])
```

Это особенно удобно, в частности, для обработки элементов последовательности с использованием рекурсивных вызовов функций или в сочетании с формой `loop`. Обратите внимание, что значением символа `rest` здесь является последовательность, а *не* вектор, даже при том, что форме деструктуризации передается вектор.

Сохранение деструктурируемого значения

Оригинал деструктурируемой коллекции можно связать с локальным символом, определив имя через опцию `:as` внутри формы деструктуризации:

```
(let [[x _ z :as original-vector] v]
     (conj original-vector (+ x z)))
;= [42 "foo" 99.2 [5 12] 141.2]
```

Здесь с именем `original-vector` будет связано исходное значение `v`. Данная возможность может пригодиться для деструктуризации коллекции, являющейся результатом вызова функции, когда вдобавок к деструктурированным значениям необходимо сохранить ссылку на исходный результат. Если бы это не было возможно, сохранить исходный результат можно было бы так:

```
(let [some-collection (some-function ...)
      [x y z [a b]] some-collection]
  ...выполнить операции с some-collection и ее значениями...)
```

Деструктуризация ассоциативных массивов

Деструктуризация ассоциативных массивов и упорядоченных коллекций концептуально идентичны – мы должны отразить структуру используемой коллекции. Эта разновидность деструктуризации работает с:

- ассоциативными массивами и записями[1];
- любыми коллекциями, реализующими интерфейс `java.util.Map`;
- любыми значениями, поддерживаемыми функцией `get` и позволяющими использовать индексы в качестве ключей:
  - векторы;
  - строки;
  - массивы.

Начнем рассмотрение с деструктуризации ассоциативных массивов:

```
(def m {:a 5 :b 6
        :c [7 8 9]
        :d {:e 10 :f 11}
        "foo" 88
        42 false})
;= #'user/m
(let [{a :a b :b} m]
  (+ a b))
;= 11
```

Здесь значение ключа `:a` в ассоциативном массиве связывается с символом `a`, а значение ключа `:b` – с символом `b`. Возвращаясь к приему визуального упорядочения формы деструктуризации коллекции (в данном случае, ее части), можно снова заметить структурное соответствие:

```
{a  :a b  :b}
{:a 5  :b 6}
```

Но самой «убойной» является возможность объединения обеих разновидностей деструктуризации, упорядоченных коллекций и ассоциативных массивов, позволяющая эффективно извлекать необходимые значения из имеющихся коллекций:

```
(let [{[x _ y] :c} m]
  (+ x y))
;= 16
(def map-in-vector ["James" {:birthday (java.util.Date. 73 1 6)}])
;= #'user/map-in-vector
(let [[name {bd :birthday}] map-in-vector]
  (str name " was born on " bd))
;= "James was born on Thu Feb 06 00:00:00 EST 1973"
```

Механизм деструктуризации ассоциативных массивов обладает также рядом дополнительных возможностей.

Сохранение деструктурируемого значения. Как и при деструктуризации упорядоченных коллекций, добавление пары с ключом `:as` позволяет сохранить ссылку на исходную коллекцию, которую затем можно использовать как любую другую `let`-связку:

```
(let [{r1 :x r2 :y :as randoms}
      (zipmap [:x :y :z] (repeatedly (partial rand-int 10)))]
  (assoc randoms :sum (+ r1 r2)))
;= {:sum 17, :z 3, :y 8, :x 9}
```

Значения по умолчанию. С помощью конструкции `:or` можно определить ассоциативный массив с парами ключ/значение по умолчанию. Если ключ, указанный в форме деструктуризации, отсутствует в исходной коллекции, с символом будет связано значение по умолчанию:

```
(let [{k :unknown x :a
       :or {k 50}} m]
  (+ k x))
;= 55
```

Создание функций: fn

Функции в языке Clojure являются *сущностями первого порядка* (first-class values)[1]. А создание их возлагается на специальную форму `fn`, которая также заключает в себе семантику форм `let` и `do`.

Ниже представлена простая функция, прибавляющая 10 к значению аргумента:

```
(fn [x]          ❶
    (+ 10 x))    ❷
```

❶ Форма `fn` принимает вектор привязок в стиле `let`, определяющий имена и количество аргументов, принимаемых функцией; к каждому аргументу применяются те же формы деструктуризации, что обсуждались в разделе «Деструктуризация (let, часть 2)».

❷ Формы, следующие за вектором привязок, образуют *тело* функции. Тело неявно заключается в форму `do`, благодаря чему тело любой функции может содержать любое количество форм. Как и в форме `do`, последняя форма в теле функции определяет возвращаемое значение.

Аргументы функции соотносятся с именами или формами деструктуризации согласно их порядку в форме вызова. То есть, в следующем вызове:

```
((fn [x] (+ 10 x)) 8)
;= 18
```

8 является единственным аргументом функции, который соответствует имени `x` в теле функции. Это делает вызов функции эквивалентным следующей форме `let`:

```
(let [x 8]
  (+ 10 x))
```

Можно определить функцию, принимающую несколько аргументов:

```
((fn [x y z] (+ x y z))
  3 4 12)
;= 19
```

В этом случае вызов функции эквивалентен следующей форме `let`:

```
(let [x 3
      y 4
      z 12]
  (+ x y z))
```

Имеется также возможность создавать функции, *работающие по-разному, в зависимости от количества аргументов*; в следующем примере мы сохранили функцию в переменную, чтобы ее можно было вызывать многократно, обращаясь к имени переменной:

```
(def strange-adder (fn adder-self-reference
                     ([x] (adder-self-reference x 1))
                     ([x y] (+ x y))))
;= #'user/strange-adder


(strange-adder 10)
;= 11
(strange-adder 10 50)
;= 60
```

При определении функции с несколькими арностями (arities), вектор привязок и тело функции для каждой арности должны быть заключены в круглые скобки. При вызове функции соответствующая арность выбирается исходя из количества переданных аргументов.

Обратите внимание на необязательное имя `adder-self-reference`, передаваемое функции в последнем примере. Этот необязательный первый аргумент в форме `fn` может использоваться в функции для ссылки на саму функцию. В данном случае арность, принимающая единственный аргумент, может вызывать арность, принимающую два аргумента, добавляя второй аргумент по умолчанию, не требуя передачи вмещающей переменной.

Примечание. Реализация взаимно-рекурсивных функций с применением формы letfn. Именованные функции (как `adder-self-reference` в примере выше) позволяют легко создавать рекурсивные функции. Однако иногда бывает необходимо определить взаимно-рекурсивные функции.

В таких редких случаях применяется специальная форма `letfn`, позволяющая одновременно определить несколько именованных функций, каждая из которых будет знать о существовании друг друга. Взгляните на примитивную реализацию функций `odd?` и `even?`:

```
(letfn [(odd? [n]
          (even? (dec n)))
        (even? [n]
          (or (zero? n)
            (odd? (dec n))))]          ❶
  (odd? 11))
;= true
```

❶ Вектор привязки включает несколько определений `fn`, но без символа `fn`.

defn основывается на fn. Мы уже видели примеры использования `defn` выше. В действительности `defn` является макросом, инкапсулирующим функциональность форм `def` и `fn`, и обеспечивающим короткий способ определения именованных функций и их регистрацию в указанном пространстве имен. Например, следующие два определения эквивалентны:

```
(def strange-adder (fn strange-adder
                     ([x] (strange-adder x 1))
                     ([x y] (+ x y))))

(defn strange-adder
  ([x] (strange-adder x 1))
  ([x y] (+ x y))))
```

С помощью этой формы можно определять и функции с одной арностью, при этом ликвидируется лишняя пара скобок. Например, следующие два определения эквивалентны:

```
(def redundant-adder (fn redundant-adder
                       [x y z]
                       (+ x y z)))

(defn redundant-adder
  [x y z]
  (+ x y z))
```

Литература

1. **Эмерик Ч., Карпер Б., Гранд К. "Программирование на Clojure" (2013)**

Эта книга продемонстрирует вам гибкость Clojure в решении типичных задач, таких как разработка веб-приложений и взаимодействие с базами данных. Вы быстро поймете, что этот язык помогает устранить ненужные сложности в своей практике и открывает новые пути решения сложных проблем, включая многопоточное программирование.
Издание предназначено для программистов, желающих освоить всю мощь и гибкость функционального программирования.