Мультипарадигмене програмування

Лекції №5-6
Функціональна мова програмування
Clojure

Що з себе уявляє Clojure?

Clojure — Lisp'овидна мова загального призначення, розроблена для Java Virtual Machine (JVM)[1]. Автором мови є Rich Hickey , який декілька років проектував мову самотужки до випуску першої публічної версії в 2007-му році.

Rich Hickey розробив Clojure, оскільки йому був потрібен сучасний Lisp для функціонального програмування, розрахований на інтеграцію з розповсюдженою платформою Java й розроблений для паралельного програмування.

Починав з розробки dotLisp для платформи .NET.

На сьогодні, стабільною версією є версия 1.8, яка була випущена в 2016-му році.

В отличие от других реализаций Lisp'a и Scheme для виртуальной машины Java, таких как ABCL, Kawa и т.д., Clojure не совместим на 100 процентов ни с Common Lisp, ни с Scheme, но позаимствовал многие идеи из этих языков, добавив новые вещи, такие как неизменяемость данных, конкурентное выполнение кода и т.п.

Более подробно о том, зачем был создан новый язык, можно прочитать на сайте проекта - http://clojure.org/rationale .

Несмотря на то, что Clojure — молодой язык программирования, достаточно много людей используют его в своих проектах, в том числе и коммерческих, например, FlightCaster, который использует Clojure при обработке большого количества данных, решая задачи Machine Learning в распределенной среде. Существуют и другие фирмы (например, Sonian, Runa, Emendio), использующие этот язык в своей работе — ссылки на них вы сможете найти на сайте языка.

Основные возможности языка

Clojure является функциональным языком программирования с поддержкой функций в качестве объектов первого класса (first class objects) и неизменяемыми (за исключением специальных случаев) данными, включая поддержку "ленивых" коллекций данных. От Lisp'а Clojure "унаследовал" макросы, мультиметоды и интерактивный стиль разработки, а JVM дает переносимость и доступ к большому набору библиотек, созданных для этой платформы.

Неизменность структур данных позволяет использовать их в разных потоках выполнения программы, что упрощает многопоточное программирование. Однако не все структуры являются неизменяемыми — в нужных случаях программист может явно использовать изменяемые структуры данных, используя Software Transactional Memory (STM), что обеспечивает надежную работу в многопоточной среде. (В качестве примера многопоточной программы, работающей с разделяемыми данными, можно привести программу "муравьи" (ants), которую достаточно сложно написать на Java из-за большого количества моделируемых сущностей, но которая достаточно просто выглядит на Clojure).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Ant
sim ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Copyright (c) Rich Hickey. All rights reserved.
; The use and distribution terms for this software are covered by the
; Common Public License 1.0 (http://opensource.org/licenses/cpl.php)
; which can be found in the file CPL.TXT at the root of this
distribution.
; By using this software in any fashion, you are agreeing to be bound
by
; the terms of this license.
; You must not remove this notice, or any other, from this software.

;dimensions of square world
(def dim 80)
;number of ants = nants-sqrt^2
(def nants-sqrt 7)
;number of places with food
(def food-places 35)
;range of amount of food at a place
```

```
(def food-range 100)
;scale factor for pheromone drawing
(def pher-scale 20.0)
;scale factor for food drawing
(def food-scale 30.0)
;evaporation rate
(def evap-rate 0.99)

(def animation-sleep-ms 100)
(def ant-sleep-ms 40)
(def evap-sleep-ms 1000)

(def running true)

(defstruct cell :food :pher) ;may also have :ant and :home

;world is a 2d vector of refs to cells
(def world
(apply vector
(map (fn [_]
```

```
(apply vector (map (fn [_] (ref (struct cell 0 0)))
(range dim))))
(range dim))))

(defn place [[x y]]
(-> world (nth x) (nth y)))

(defstruct ant :dir) ;may also have :food

(defn create-ant
"create an ant at the location,returning an ant agent on the location"
[loc dir]
(sync nil
  (let [p (place loc)
  a (struct ant dir)]
  (alter p assoc :ant a)
(agent loc))))

(def home-off (/ dim 4))
(def home-range (range home-off (+ nants-sqrt home-off)))
```

```
(defn setup
"places initial food and ants, returns seq of ant agents"
[]
(sync nil
(dotimes [i food-places]
(let [p (place [(rand-int dim) (rand-int dim)])]
(alter p assoc :food (rand-int food-range))))
(doall
(for [x home-range y home-range]
(do
(alter (place [x y])
assoc :home true)
(create-ant [x y] (rand-int 8)))))))

(defn bound
"returns n wrapped into range 0-b"
[b n]
(let [n (rem n b)]
(if (neg? n)
```

```
(+ n b)
n)))

(defn wrand
"given a vector of slice sizes, returns the index of a slice given a
random spin of a roulette wheel with compartments proportional to
slices."
[slices]
(let [total (reduce + slices)
r (rand total)]
(loop [i 0 sum 0]
(if (< r (+ (slices i) sum))
i
(recur (inc i) (+ (slices i)
sum))))))

;dirs are 0-7, starting at north and going clockwise
;these are the deltas in order to move one step in given dir
(def dir-delta {0 [0 -1]
1 [1 -1]
```

```
2 [1 0]
3 [1 1]
4 [0 1]
5 [-1 1]
6 [-1 0]
7 [-1 -1]})

(defn delta-loc
"returns the location one step in the given dir. Note the world is a
torus"
[[x y] dir]
(let [[dx dy] (dir-delta (bound 8 dir))]
[(bound dim (+ x dx)) (bound dim (+ y dy))]))

;(defmacro dosync [& body]
; `(sync nil ~@body))

;ant agent functions
;an ant agent tracks the location of an ant, and controls the behavior
of
```

```
;the ant at that location

(defn
turn
"turns the ant at the location by the
given amount"
[loc
amt]
(dosync
(let [p (place
loc)
ant (:ant
@p)]
(alter p assoc :ant (assoc ant :dir (bound 8 (+
(:dir ant) amt))))))
loc
)

(defn move
"moves the ant in the direction it is heading. Must
be called in a
```

```
transaction that has verified the
way is clear"
[loc
]
(let [oldp (place
loc)
ant (:ant
@oldp)
newloc (delta-loc loc
(:dir ant))
p (place
newloc)]
;move the
ant
(alter p
assoc :ant ant)
(alter oldp
dissoc :ant)
;leave pheromone
trail
(when-not (:home
@oldp)
```

```
(alter oldp assoc :pher (inc (:pher
@oldp))))
newloc)
)

(defn take-food
[loc]
"Takes one food from current location. Must be
called in a
transaction that has verified there is
food available"
(let [p (place
loc)
ant (:ant
@p)]
(alter p
assoc
:food (dec (:food
@p))
:ant (assoc ant :food
true))
```

```
loc)
)

(defn drop-food
[loc]
"Drops food at current location. Must be
called in a
transaction that has verified the
ant has food"
(let [p (place
loc)
ant (:ant
@p)]
(alter p
assoc
:food (inc (:food
@p))
:ant (dissoc
ant :food))
loc)
)
```

```
(defn rank-
by
"returns a map of xs to their 1-based rank when
sorted by keyfn"
[keyfn
xs]
(let [sorted (sort-by (comp float
keyfn) xs)]
(reduce (fn [ret i] (assoc ret (nth
sorted i) (inc i)))
{} (range (count
sorted)))))


(defn
behave
"the main function for the ant
agent"
[loc
]
(let [p (place
loc)
```

```
ant (:ant
@p)
ahead (place (delta-loc loc
(:dir ant)))
ahead-left (place (delta-loc loc (dec
(:dir ant))))
ahead-right (place (delta-loc loc (inc
(:dir ant))))
places [ahead ahead-left ahead-
right]]
(. Thread (sleep ant-
sleep-ms))
(dosync
(when
running
(send-off *agent*
#'behave))
(if (:food
ant)
;going
home
(cond
```

```
(:home @p)
(-> loc drop-food
(turn 4))
(and (:home @ahead) (not (:ant
@ahead)))
(move
loc)
:els
e
(let [ranks (merge-
with +
(rank-by (comp #(if (:home %) 1 0)
deref) places)
(rank-by (comp :pher deref)
places))]
(([move #(turn % -1)
#(turn % 1)]
(wrand [(if (:ant @ahead) 0 (ranks
ahead))
(ranks ahead-left) (ranks ahead-
right)]))
```

```
loc))
)
;foragin
g
(cond
(and (pos? (:food @p)) (not (:home
@p)))
(-> loc take-food
(turn 4))
(and (pos? (:food @ahead)) (not (:home @ahead)) (not
(:ant @ahead)))
(move
loc)
:els
e
(let [ranks (merge-
with +
(rank-by (comp :food deref)
places)
(rank-by (comp :pher deref)
places))]
```

```
(([move #(turn % -1)
#(turn % 1)]
(wrand [(if (:ant @ahead) 0 (ranks
ahead))
(ranks ahead-left) (ranks ahead-
right)]))
loc))))
)))

(defn
evaporate
"causes all the pheromones to
evaporate a bit"
[]
(dorun
(for [x (range dim) y
(range dim)]
(dosync
(let [p (place
[x y])]
```

```
(alter p assoc :pher (* evap-rate
(:pher @p))))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
UI ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(import
'(java.awt Color Graphics
Dimension)
'(java.awt.image
BufferedImage)
'(javax.swing JPanel
JFrame))

;pixels per world
cell
(def scale
5)

(defn fill-cell [#^Graphics
g x y c]
```

```
(doto
g
(.setColor
c)
(.fillRect (* x scale) (* y scale)
scale scale)))

(defn render-ant [ant #^Graphics
g x y]
(let [black (. (new Color 0 0 0 255)
(getRGB))
gray (. (new Color 100 100 100 255)
(getRGB))
red (. (new Color 255 0 0 255)
(getRGB))
[hx hy tx ty] ({0 [2
0 2 4]
1 [4 0 0 4]
2 [4 2 0 2]
3 [4 4 0 0]
4 [2 4 2 0]
```

```
5 [0 4 4 0]
6 [0 2 4 2]
7 [0 0 4 4]}
(:dir ant))]
(doto g
(.setColor (if (:food ant)
(new Color 255 0 0 255)
(new Color 0 0 0 255)))
(.drawLine (+ hx (* x scale)) (+ hy (* y scale))
(+ tx (* x scale)) (+ ty (* y scale))))))

(defn render-place [g p x y]
(when (pos? (:pher p))
(fill-cell g x y (new Color 0 255 0
(int (min 255 (* 255 (/ (:pher p) pher-scale)))))))
(when (pos? (:food p))
(fill-cell g x y (new Color 255 0 0
(int (min 255 (* 255 (/ (:food p) food-scale)))))))
(when (:ant p)
(render-ant (:ant p) g x y)))
```

```
(defn render [g]
(let [v (dosync (apply vector (for [x (range dim) y (range dim)]
@(place [x
y])))))
img (new BufferedImage (* scale dim) (*
scale dim)
(. BufferedImage TYPE_INT_ARGB))
bg (. img
(getGraphics))]
(doto
bg
(.setColor (. Color
white))
(.fillRect 0 0 (. img (getWidth)) (. img
(getHeight))))
(dorun
(for [x (range dim) y
(range dim)]
(render-place bg (v (+ (* x dim)
y)) x y)))
```

```
(doto
bg
(.setColor (. Color
blue))
(.drawRect (* scale home-off) (* scale
home-off)
(* scale nants-sqrt) (* scale
nants-sqrt)))
(. g (drawImage img 0 0
nil))
(. bg
(dispose))))

(def panel (doto (proxy
[JPanel] []
(paint [g] (render
g)))
(.setPreferredSize (new
Dimension
(* scale
dim)
```

```
(* scale
dim)))))

(def frame (doto (new JFrame) (.add
panel) .pack .show))

(def animator (agent
nil))

(defn animation
[x]
(when
running
(send-off *agent*
#'animation))
(. panel
(repaint))
(. Thread (sleep animation-
sleep-ms))
nil
)
```

```
(def evaporator (agent
nil))

(defn evaporation
[x]
(when
running
(send-off *agent*
#'evaporation))
(evaporat
e)
(. Thread (sleep evap-
sleep-ms))
nil
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
use ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(comment
```

```
;demo
;; (load-file
"/Users/rich/dev/clojure/ants.clj")
(def ants
(setup))
(send-off animator
animation)
(dorun (map #(send-off % behave)
ants))
(send-off evaporator
evaporation)

;; )
```

За счет того, что Clojure был спроектирован для работы на базе JVM, обеспечивается доступ к большому набору библиотек, существующих для данной платформы. Взаимодействие с Java реализуется в обе стороны — как вызов кода, написанного на Java, так и реализация классов, которые доступны как для вызова из Java, так и из других языков, существующих для JVM, например, Scala. Подробнее о взаимодействии с JVM написано далее.

Отличия от Lisp

Несмотря на схожесть синтаксиса, Clojure отличается и от Common Lisp, и от Scheme. Некоторые отличия обусловлены тем, что язык разработан для платформы JVM, что накладывает некоторые ограничения на реализацию. Например, JVM не поддерживает оптимизацию хвостовых вызовов (tail call optimization, TCO), поэтому в язык были введены явные операторы `loop` и `recur`. Также важными определяющими факторами JVM-платформы являются:

- boxed integers — нет поддержки полного набора типов чисел (numeric tower), которые есть в Scheme и Common Lisp;
- система исключений как в Java (в Common Lisp используется сигнальный протокол);
- используется соглашение о вызовах как в Java.

Полный список отличий можно найти на отдельной странице на сайте языка. Из явных отличий от Common Lisp можно отметить следующие:

- идентификаторы в Clojure регистрозависимы (case-sensitive);
- большая часть данных — неизменяемая;
- пользователь не может изменять синтаксис языка путем ввода собственных макросов в процедуре чтения кода (read macros);
- введен специальный синтаксис для литералов, векторов, отображений (maps), регулярных выражений, анонимных функций и т.д.;
- существует возможность связывания метаданных с переменными и функциями;
- можно реализовать функции с одним именем и разным набором аргументов;
- многие привычные вещи, такие как `let`, по синтаксису отличаются от их аналогов в Common Lisp и Scheme (при этом используется меньше скобок), например, `let` связывает данные последовательно, аналогично `let*` в Scheme;

- вместо функций `car` и `cdr` используются функции `first` и `rest`;
- `nil` не равен пустому списку или другому набору данных (коллекции) — он всего лишь означает отсутствующее значение (аналог `null` в Java);
- используется общее пространство имен, как в Scheme;
- сравнение на равенство производится одной функцией в отличие от Common Lisp и Scheme;
- поддержка "ленивых" коллекций.

Источники информации о языке

Основной источник информации по данному языку — сайт проекта и список рассылки. Помимо сайта проекта, хорошим источником информации является набор видеолекций на Blip.TV, а также видеолекции, в которых автор языка рассказывает о Clojure и об особенностях его использования. Кроме того, следует отметить набор скринкастов, созданных Sean Devlin, в которых он рассказывает о разных возможностях языка, включая новые, появившиеся в версии 1.1.

Из книг в настоящее время доступна книга Programming Clojure, выпущенная в серии Pragmatic Programmers, которая в принципе содержит всю необходимую информацию о языке, включая описание основных возможностей языка, вопросы взаимодействия с Java, основные функции, отличие языка от Common Lisp, и т.п. В мае 2010 года издательство Apress выпустило еще одну книгу по Clojure — Practical Clojure. The Definitive Guide, которая является кратким описанием современной версии языка, включая новшества, которые введены в версии 1.2. А на начало 2011 года в издательстве Manning запланирован выход книг Clojure in Action (введение в язык и примеры практического использования) и The Joy of Clojure. Thinking the Clojure Way (более "глубокое" описание языка, с разъяснением сложных понятий).

В свободном доступе можно найти книгу Clojure Programming, работа над которой ведется в рамках проекта WikiBooks. Также существует достаточно подробный практический учебник — Clojure Scripting. Кроме того, недавно был опубликован учебник Clojure Notes, который использовался в рамках курса обучения Clojure.

Хорошее описание того, как можно использовать макросы для построения абстракций, можно найти в известной книге On Lisp Пола Грэма (Paul Graham). Несмотря на то, что в ней используется Common Lisp, многие вещи будут применимы и для Clojure.

Очень большое количество информации о языке, разрабатываемых библиотеках и проектах, использующих Clojure, публикуется в блогах. Для того, чтобы свести всю эту информацию воедино, существует проект Planet Clojure, на который вы можете подписаться, чтобы быть в курсе новостей о языке.

Установка и запуск

Установка Clojure достаточно проста — скачайте последнюю версию с сайта языка и распакуйте в нужный каталог. После этого вы можете запустить ее с помощью команды:

```
java -cp clojure.jar clojure.main
```

Эта команда приведет к запуску JVM и вы получите доступ к REPL ("read-eval-print loop" — цикл ввода выражений и выдачи результатов). Стандартный REPL имеет не очень хорошие возможности по редактированию кода, так что при работе с REPL лучше использовать библиотеку `jline`, как описано в разделе Getting Started официальной документации Clojure, или воспользоваться одной из сред разработки, описанных в разделе Среды разработки. Более подробные инструкции по развертыванию для разных сред разработки вы можете найти в описании проекта labrepl, целью которого является упрощение начала работы с Clojure.

В составе данного проекта имеется набор учебных материалов, которые будут полезны начинающим работать с языком.

Работая в REPL вы можете получать информацию о функциях, макросах и других объектах языка. Для получения информации о каком-либо символе или специальной форме вы можете использовать макрос `doc`. Например, `(doc map)` напечатает справку по функции `map`, которая была задана при объявлении этой функции. А если вы не помните точное название символа, можно провести поиск по документации с помощью функции `find-doc`, которая принимает один аргумент — строку с регулярным выражением по которому будет проводиться поиск.

Из чего состоит язык Clojure

Синтаксис языка Clojure следует стандартному для Lisp'образных языков подходу "код как данные", когда данные и код имеют общий синтаксис. Как и в других диалектах Lisp'а, код записывается в виде списков, используя префиксную нотацию и представляя собой синтаксическое дерево. Однако по сравнению с другими языками, в Clojure введены дополнительные сущности: кроме стандартных для Lisp'а символов, базовых литералов (строки, числа и т.п.) и списков, в язык введен дополнительный синтаксис для векторов, отображений (maps) и множеств (sets), являющихся объектами первого класса (first class objects).

Кроме этого, процедура чтения кода (reader) распознает специфические для Clojure конструкции: @ — для доступа к изменяемым данным и различные конструкции, начинающиеся с символа # — анонимные функции, метаданные (включая информацию о типах данных), регулярные выражения и т.д. Процедура чтения также рассматривает пробелы и запятые между элементами языка как один символ, разделяющий эти элементы.

Основные типы данных

Данные в Clojure можно разделить на две большие группы: базовые типы данных — числа, строки и т.д., и последовательности (коллекции), к которым относятся списки, векторы, отображения и множества. Пользователь может определять свои структуры данных с помощью `defstruct`, но они являются частным случаем отображений и введены для обеспечения более эффективной работы со сложными данными.

Все типы данных имеют общий набор характеристик: данные неизменяемы и реализуют операцию "равенство" (equality).

Базовые типы данных

К базовым типам данных Clojure относятся следующие:

логические значения
: в языке определено два объекта для представления логических значений: `true` — для истинного значения и `false` — для ложного. (Все остальные значения, кроме `false` и `nil`, рассматриваются как истинные);

числа

в языке могут использоваться числа разных типов. По умолчанию для представления целых чисел используются классы, унаследованные от `java.lang.Number` — `Integer`, `BigInteger`, `BigDecimal`, но в Clojure реализуется специальный подход, который позволяет представлять число наиболее эффективным способом, автоматически преобразуя числа в случае необходимости — например, при переполнении числа. Если вы хотите для целого числа явно указать тип `BigDecimal`, то вы можете добавить букву `M` после значения.

Для чисел с плавающей точкой используется стандартный класс `Double`.

Кроме этих видов чисел, в Clojure определен специальный тип `Ratio`, представляющий числа в виде рациональных дробей, что позволяет избегать ошибок округления — например, при делении.

строки

строки в Clojure являются экземплярами класса `java.lang.String` и к ним можно применять различные функции определенные в этом классе. Форма записи строк Clojure совпадает со стандартной записью строк в Java;

знаки (characters)

являются экземплярами класса `java.lang.Character` и записываются либо в форме `\N`, где `N` — соответствующая буква, либо как названия для неотображаемых букв — например, как `\tab` и `\space` для символа табуляции и пробела и т.д.;

символы (symbols)

используются для ссылки на что-то — параметры функций, имена классов, глобальные переменные и т.д. Для представления символа как отдельного объекта, а не как значения, для которого он используется в качестве имени, используется стандартная запись `'symbol` (или специальная форма `quote`);

keywords (ключевые символы)

это специальные символы, имеющие значение самих себя[3], аналогично символам (symbols) в Lisp и Ruby. Одним из важных их свойств является очень быстрая операция проверки на равенство, поскольку происходит проверка на равенство указателей. Это свойство делает их очень удобными для использования в качестве ключей в отображениях (maps) и тому подобных вещах. Для именованных аргументов существует специальная форма записи `:keyword`.

Стоит также отметить, что символы и keywords имеют некоторую общность — в рамках интерфейса `IFn` для них создается функция `invoke()` с одним аргументом, что позволяет использовать символы и keywords в качестве функции. Например, конструкция `(:mykey my-hash-map)` или `('mysym my-hash-map)` аналогичны вызову `(get my-hash-map :mykey)` или `(get my-hash-map 'mysym)`, который приведет к извлечению значения с нужным ключом из соответствующего отображения.

В языке Clojure имеется специальное значение `nil`, которое может использоваться как значение любого типа данных, и совпадающее с `null` в Java. `nil` может использоваться в условных конструкциях наравне со значением `false`. Однако стоит отметить, что, в отличие от Lisp, `nil` и пустой список — `()` не являются взаимозаменяемыми и использование пустого списка в условной конструкции будет рассматриваться как значение `true`;

Коллекции, последовательности и массивы

Кроме общих характеристик базовых типов перечисленных выше, все коллекции в Clojure имеют следующие характеристики:

- вся работа с коллекциями проводится через общий интерфейс;
- существует возможность связывания метаданных с коллекцией;
- для коллекций реализуются интерфейсы `java.lang.Iterable` и `java.util.Collection`, что позволяет работать с ними из Java;
- все коллекции рассматриваются как "последовательности" данных, вне зависимости от конкретного представления данных внутри них.

Неизменяемость коллекций означает, что результатом работы всех операций по модификации коллекций является другая, новая коллекция, в то время как исходная коллекция остается неизменной. В Clojure существует эффективный механизм, помогающий реализовывать неизменяемые коллекции. С его помощью операции, изменяющие коллекцию, могут эффективно создавать "измененную" версию данных, которая использует большую часть исходных данных, не создавая полной копии.

В текущей версии Clojure реализованы следующие основные виды коллекций:

списки (lists)
: записываются точно также как и в других реализациях Lisp. В Clojure списки напрямую реализуют интерфейс `ISeq`, что позволяет функциям работы с последовательностями эффективно работать с ними. (При использовании функции `conj` новые элементы списков добавляются в начало);

векторы (vectors)
: представляют собой последовательности, элементы которых индексируются целым числом (с последовательными значениями индекса в диапазоне `0..N`, где `N` — размер вектора). Для определения вектора необходимо заключить его элементы в квадратные скобки, например, `[1 2 3]`. Для преобразования других коллекций в вектор можно использовать функции `vector` или `vec`. Поскольку вектор индексируется целым числом, то операция доступа к произвольному элементу реализуется достаточно эффективно, что удобно при работе с некоторыми видами данных. (При использовании функции `conj` новые элементы векторов добавляются в конец.)

Кроме того, для вектора в Clojure создается функция одного аргумента (целого числа — индекса значения) с именем, совпадающим с именем символа, связанным с вектором. Это позволяет использовать имя вектора в качестве функции для доступа к нужному значению. Например, вызов `(v 3)` в данном коде:

```
user> (def v [1 2 3 4 5 "string"])
user> (v 3)
4
```

вернет значение четвертого элемента вектора.

отображения (maps)

это специальный вид последовательности, который отображает одни значения данных (ключ) в другие (значения). В Clojure существуют два вида отображений: `hash-map` и `sorted-map`, которые создаются с помощью соответствующих функций. `hash-map` обеспечивает более быстрый доступ к данным, а `sorted-map` хранит данные в отсортированном по ключу виде. Отображения записываются в виде набора значений (с четным количеством элементов), заключенных в фигурные скобки. Значения, стоящие на нечетных позициях рассматриваются как ключи, а на четных — как значения, связанные с данным ключом. В качестве ключа могут использоваться любые поддерживаемые Clojure типы данных, но очень часто в качестве ключей используют keywords, поскольку для них реализована очень быстрая проверка на равенство.

Также как и для векторов, для отображений создается функция одного аргумента (ключа), которая позволяет использовать имя символа, связанного с отображением, для доступа к элементам. Например,

```
user> (def m {:1 1 :abc 33 :2 "2" })
#'user/m
user> (m :abc)
33
```

множества (sets)

представляет собой набор уникальных значений. Также как и для отображений, существует два вида множеств — `hash-set` и `sorted-set`. Определение множества имеет следующий вид `#{elements...}`, а для создания множества из других коллекций может использоваться функция `set`, например, для получения множества уникальных значений вектора, можно использовать следующий код:

```
user> (set [1 2 3 2 1 2 3])
#{1 2 3}
```

В Clojure также определены дополнительные виды отображений, позволяющие в специальных случаях добиться большей производительности:

отображения-структуры (struct maps)

могут использоваться для эмуляции записей (records), имеющихся в других языках программирования. В этом случае отображения имеют набор одинаковых ключей и Clojure реализует эффективное хранение информации о ключах, а также предоставляет быстрый доступ к элементам по ключу. В случае необходимости, имеется возможность генерации специализированной функции доступа с помощью функции `accessor`.

Определение отображения-структуры производится с помощью макроса `defstruct` или функции `create-struct`. Новые экземпляры отображений создаются с помощью функции `struct-map` или `struct`, которые получают список элементов для заполнения данного отображения. При этом стоит отметить, что отображение-структура может иметь большее количество ключей, чем было определено в `defstruct` — в этом отношении, отображения-структуры ведут себя точно также, как и обычные отображения.

отображения-массивы (array maps)

это специальный вид отображений, в котором сохраняется порядок ключей. Такие отображения реализованы в виде обычного массива, содержащего ключи и значения. Поиск в отображении является линейной функцией от количества элементов, и поэтому, такие отображения должны использоваться только для хранения небольшого количества элементов. Новые отображения-массивы могут создаваться с помощью функции `array-map`.

Работа с коллекциями выполняется единообразно — для всех коллекций поддерживаются операции `count` для получения размера коллекции, `conj` для добавления элементов в коллекцию (реализуется по-разному, в зависимости от конкретного типа) и `seq` для представления коллекции в виде последовательности — это позволяет применять к ним функции работы с последовательностями: `cons`, `first`, `map` и т.д. Функцию `seq` также можно использовать для преобразования в последовательности и коллекций Java.

Большая часть функций для работы с последовательностями является "ленивой", обрабатывая данные по мере их надобности, что позволяет эффективно работать с данными большого размера, в том числе и с бесконечными последовательностями. Пользователь может создавать свои функции, которые возвращают "ленивые" последовательности, с помощью макроса `lazy-seq`. Также в версии 1.1 было введено понятие блоковых последовательностей (chunked sequence), которые позволяют создавать элементы блоками по N элементов, что в некоторых случаях позволяет улучшить производительность.

Из общего ряда выпадает работа с массивами Java, поскольку они не являются коллекциями в терминах Clojure. Для работы с массивами определен набор функций, которые позволяют определять массивы разных типов (`make-array`, `XXX-array`, где `XXX` — название типа), получения (`aget`) и установки (`aset`) значений в массиве, преобразования коллекций в массив (`into-array`) и т.д.

"Ленивые" структуры данных

Как уже упоминалось выше, большая часть структур данных в Clojure (и функций для работы с этими структурами данных) являются "ленивыми" (lazy). В языке имеется явная поддержка "ленивых" структур данных, позволяя программисту эффективно работать с ними. Одним из достоинств поддержки "ленивых" структур данных является то, что можно реализовывать очень большие или бесконечные последовательности используя конечное количество памяти. "Ленивые" последовательности и функции также могут использоваться для обхода ограничений, связанных с отсутствием оптимизации хвостовых вызовов в Clojure.

В Clojure программист может определить "ленивую" структуру данных воспользовавшись макросом `lazy-seq`. Данный макрос в качестве аргумента принимает набор выражений, которые возвращают структуру данных, реализующую интерфейс `ISeq`. Из этих выражений затем создается объект типа `Seqable`, который вызывается по мере необходимости, кешируя полученные результаты.

В качестве примера использования "ленивых" структур данных давайте рассмотрим создание бесконечной последовательности чисел Фибоначи:

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

В данном случае мы определяем функцию, которая при запуске без аргументов создает начальную последовательность из чисел 1 и 1 и затем вызывает сама себя, передавая эти числа в качестве аргументов. А функция двух аргументов обернута в вызов `lazy-seq`, который производит вычисление следующих чисел Фибоначи.

При этом мы можем определить переменную, которая, например, будет содержать первые сто миллионов чисел Фибоначи:

```
user> (def many-fibs (take 100000000 (fibo)))
#'user/many-fibs
```

но поскольку мы работаем с "ленивыми" последовательностями, то значение будет создано мгновенно, без вычисления всех чисел Фибоначи. А само вычисление чисел будет происходить по мере надобности. Например, мы можем получить 55-е число Фибоначи с помощью следующего кода:

```
user> (nth many-fibs 55)
225851433717
```

Переходные структуры данных (transients)

При интенсивной работе с неизменяемыми коллекциями иногда возникает слишком много промежуточных объектов, что достаточно неэффективно. В версии 1.1 появилась возможность временно использовать изменяемые коллекции данных используя переходные (transient) структуры данных. Эта функциональность была специально введена в язык для оптимизации производительности.

Основная идея заключается в том, чтобы избежать ненужного копирования данных, что происходит когда вы работаете с неизменяемыми данными. Стоит отметить, что не все структуры данных поддерживают эту возможность — в версии 1.1.0 поддерживаются векторы, отображения и множества, а списки — нет, поскольку для них нет существенного выигрыша в производительности. В общем виде работа с переходными структурами данных происходит следующим образом:

- вы преобразуете стандартную структуру данных в переходную структуру (вектор, отображение и т.д.) с помощью функции `transient`, получающей один параметр — соответствующую структуру данных;
- выполняете изменение структуры по месту (inplace) с помощью специальных функций `assoc!`, `conj!` и т.п., которые аналогичны по действию соответствующим функциям, но без символа `!`, но применяются только к переходным структурам данных;
- после окончания обработки, превращаете переходную структуру данных в стандартную, неизменяемую структуру данных с помощью функции `persistent!`.

Рассмотрим простой пример использования стандартных и переходных структур данных:

```
(defn vrange [n]
  (loop [i 0
         v []]
    (if (< i n)
      (recur (inc i) (conj v i))
      v)))

(defn vrange2 [n]
  (loop [i 0
         v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))
```

Обе функции создают вектор заданного размера, состоящий из чисел в диапазоне `0..n`. В первой функции используются стандартные, неизменяемые структуры данных, а во второй — переходные структуры данных. Как видно из примера, во второй функции выполняются все требования к использованию переходных структур — сначала с помощью вызова `(transient [])` создается ссылка на вектор, который затем заполняется с помощью функции `conj!`, и в конце происходит возвращение неизменяемой структуры данных, созданной из переходной структуры с помощью вызова `(persistent! v)`. Если мы запустим обе функции с одинаковыми параметрами, то мы получим следующие результаты:

```
user> (time (def v (vrange 1000000)))
"Elapsed time: 439.037 msecs"
user> (time (def v2 (vrange2 1000000)))
"Elapsed time: 110.861 msecs"
```

Как видно из этого примера, использование переходных структур дает достаточно большой выигрыш в производительности — примерно в четыре раза. А на некоторых структурах данных выигрыш в производительности может быть больше. Копирование исходных данных и создание неизменяемой структуры — это операции со сложностью O(1), при этом происходит эффективное использование оригинальных данных. Также стоит отметить, что использование переходных структур данных приводит к принудительной изоляции потока выполнения — изменяемые данные становятся недоступными из других потоков выполнения.

Более подробно о переходных структурах данных вы можете прочитать на сайте языка.

Базовые конструкции языка

Синтаксис языка в общем виде (за исключением специальных форм для отображений, векторов и других элементов) совпадает с синтаксисом Common Lisp и Scheme — все описывается S-выражениями. Код записывается в виде списков, используя префиксную нотацию, когда первым элементом списка является функция, макрос или специальная форма, а остальные элементы — аргументы, которые будут переданы в это выражение. Кроме списков, S-выражения могут состоять из атомов: чисел, строк, логических констант и т.д.

Объявление и связывание символов

В Lisp'е существуют объекты типа символ, которые в отличие от других языков представляют собой не просто имена переменных, а отделенные сущности. То, что в других языках понимается как присваивание значения переменной (с определенным именем, которое по сути является просто меткой области памяти), в Lisp'е формулируется по другому — как связывание значения с символом, т.е. связывание двух объектов. Через эти же понятия формулируется и подход Lisp'а к типизации — значения имеют типы, а переменные — нет, поскольку переменная — это и есть пара символ-значение. Если вы используете имя символа без маскирования (специальная форма `quote`, или `'`), то вместо символа будет подставлено значение, связанное с этим символом. Если же вам нужно передать сам символ, то вы должны использовать запись `'имя-символа`.

Имеется два вида связывания символа со значением:

- *лексическое*, когда значение привязано к символу только внутри тела одной формы, например `let` или `fn`. Как раз тут и проявляется особенность Clojure — функциональные переменные: внутри тела `let` нельзя установить значение используя `set!`. Можно считать, что в этом случае `x` — это псевдоним для значения `2`.

- *динамическое*, когда значение привязывается к символу на время выполнения программы. Как правило, в этом случае символ делается доступным глобально (хотя в Common Lisp он может быть и локальным, но объявленным с помощью *declare special*). Суть слова *динамичность* в том, что в любой момент и в любом месте символ может быть заново связан с другим значением, и значение будет "видно" при всех обращениях к этому символу в любых формах. Такие динамические (в данном случае и глобальные в рамках пространства имен) символы определяются и связываются в Clojure формой `def`, на основе которой затем строятся макросы `defn`, `declare`, `defonce` и т.д. Таким образом, например, имя функции привязывается к самому объекту функции, что будет видно на примере раскрытия макроса `defn` ниже.

Таким образом, говорится о двух областях видимости: лексической и динамической. Нужно иметь в виду, что это не то же самое, что и локальные и глобальные переменные. Хотя динамически связанные символы, как правило, глобальны, а лексически связанные — локальны. А как быть в случае конфликта лексической и динамической привязки?

- В Common Lisp существует дуализм связывания, выраженный в специальной форме `let`, которая может связывать как лексические, так и специальные (динамические) символы. Преимущество этой формы в том, что при такой привязке динамический символ как бы приобретает свойство лексического: его новое значение видно только во время выполнения тела `let`, в то время как в других формах доступно его "глобальное" значение. Это очень мощная возможность, благодаря которой использование глобальных переменных в Lisp не приводит к тем разрушительным последствиям, которые присутствуют в других языках;
- В Clojure реализован тот же принцип за тем исключением, что разделены формы, выполняющие обычное лексическое связывание (для лексических

символов, которые теперь смело можно назвать локальными) — `let`, и лексическое связывание для динамических символов — это форма `binding`. Подробней об этом можно прочитать в обсуждении "Let vs. Binding in Clojure" на StackOverflow.com.

Общая форма объявления "динамического" символа выглядит как `(def имя значение?)`, при этом значение может не указываться и тогда символ будет несвязанным со значением. Существуют разные макросы для объявления символов, которые выполняют определенные задачи, например, `defonce` позволяет объявить символ только один раз, и если он уже имеет значение, то новое значение будет проигнорировано. А с помощью макроса `declare` можно выполнить опережающее определение (forward declaration) символа — это часто удобно при объявлении взаимно рекурсивных функций. При объявлении символа можно указать метаданные, связанные с данным символом (см. раздел Метаданные).

Для объявления "лексических" символов используется форма `let`, которая выглядит следующим образом: `(let [имя1 знач1 имя2 знач2 ....] код)`. Например, выполнение следующего кода:

```
(let [x 1
      y 2]
   (+ x y))
```

выдаст значение `3`. В этом случае переменные `x` и `y` видны только внутри формы `(+ x y)`, и возможно маскируют значения переменных, объявленных на глобальном уровне, или выше по коду.

В некоторых случаях, программист может переопределить значение "динамического" символа для конкретного потока выполнения. Это делается с помощью макроса `binding`, который переопределяет значение указанного символа для данного потока выполнения и всех вызываемых из него функций.

Например:

```
user> (def x 10)
user> (def y 20)
user> (defn test-fn []
            (+ x y))
user> (test-fn)
30
user> (binding [x 11
                y 22]
         (test-fn))
33
user> (let [x 11
            y 22]
         (test-fn))
30
```

В данном коде, если мы выполним `test-fn` на верхнем уровне кода, то получим значение `30`, равное сумме значений переменных `x` и `y`. А если эта функция будет вызвана из `binding`, то мы получим значение `33`, равное сумме переменных объявленных в `binding`. Данные значения изменяются только для текущего потока выполнения, и только для кода, который будет вызван из `binding`. После завершения выполнения кода в этой форме все предыдущие значения восстанавливаются. А при использовании `let` значения `x` и `y` не воздействуют на функцию `test-fn`, и в ней используются "глобальные" значения, давая в результате `30`. *Стоит быть осторожным при использовании функций работы с последовательностями внутри* `binding`, *поскольку они возвращают "ленивые" последовательности, данные в которых будут вычисляться уже вне* `binding`.

Деструктуризация параметров

Если вы передаете сложные структуры данных в функции (`defn` или `fn`), или связываете значения с символами в `let`, то вы можете использовать встроенную в Clojure деструктуризацию параметров, что позволяет связывать части структур данных с символами. Это значительно упрощает (и сокращает) код, поскольку вам не нужно писать явный код для получения значения значений из массивов или отображений.

Существует две формы деструктуризации параметров — деструктуризация отображений, и деструктуризация векторов, строк и массивов.

Деструктуризация векторов, строк и массивов

Деструктуризация векторов/массивов/строк имеет простую форму — вы помещаете список символов внутри вектора, и значения, стоящие на соответствующих местах в деструктурируемом объекте, будут связаны с этими символами. В том случае, если вы укажете меньше аргументов, чем указано символов, то отсутствующие значения будут рассматриваться как имеющие значение `nil`, если укажете больше, то они будут отброшены. Вы можете собрать "лишние" значения в отдельный список, используя запись `& символ`, и все "лишние" значения будут помещены в список, связанный с указанным символом.

Вот примеры использования деструктуризации векторов и строк:

```
user> (let [ [a b] [1 2]]
  (list a b))
(1 2)
user> (let [ [a b] [1]]
  (list a b))
(1 nil)
user> (let [ [a b] [1 2 3 4]]
  (list a b))
(1 2)
user> (let [ [a b & c] [1 2 3 4]]
  (list a b c))
(1 2 (3 4))
user> (let [ [a b & c] "abcde"]
  (list a b c))
(\a \b (\c \d \e))
user> ((fn [ [a b] ] (list a b)) [1 2])
(1 2)
user> (defn dfunc1 [ [a b] ]
  (list a b))
user> (dfunc1 [1 2])
(1 2)
```

Деструктуризация отображений

Деструктуризация отображений выглядит следующим образом — вы записываете отображение, в котором ключом является символ, с которым будет связано значение, а значением — является ключ в деструктурируемом отображении. Например,

```
user> (let [{a :k1 b :k2 c :k3} {:k1 1 :k2 2 :k3 3}]
  (list a b c))
(1 2 3)
user> (let [{a :k1 b :k2 c :k3} {:k1 1 :k3 3}]
  (list a b c))
(1 nil 3)
```

Также как и в случае с деструктуризацией векторов, если конкретный ключ не был указан, то символ получает значение `nil`, как это видно во втором примере.

Деструктуризация отображений может иметь более сложную форму, поскольку можно указывать еще и значения по умолчанию, которые будут присвоены соответствующим символам вместо `nil`, если нужный ключ не был указан в отображении. Это осуществляется путем указания ключевого символа `:or` и отображения, в котором перечисляются значения по умолчанию:

```
user> (let [{a :a b :b c :c :or {a 1 b 2 c 3}} {:a 4}]
  (list a b c))
(4 2 3)
```

Для того, чтобы не писать пары символ/ключ, имеется упрощенная форма записи, когда вы указываете ключевой символ `:keys` и вектор, содержащий список символов, которые будут превращены в ключевые слова с теми же самыми именами, и которые будут использоваться для поиска ключей в деструктурируемом отображении. Например, предыдущие примеры можно переписать в более компактной форме:

```
user> (let [{:keys [a b c]} {:a 1 :b 2 :c 3}]
  (list a b c))
(1 2 3)
user> (let [{:keys [a b c]} {:a 1 :c 3}]
  (list a b c))
(1 nil 3)
```

Кроме ключевых символов, в качестве ключей отображений вы можете использовать строки и символы. Для этого нужно использовать ключевое слово `:strs` — для строк и `:syms` — для символов, как это показано в следующих примерах:

```
user> (let [{:strs [a b c] :as m :or {a 2 b 3}} {"a" 5 "c" 6}]
  (list a b c m))
(5 3 6 {"a" 5, "c" 6})

user> (let [{:syms [a b c] :as m :or {a 2 b 3}} {'a 5 'b 6}]
  (list a b c m))
(5 3 6 {a 5, c 6})
```

Особенно полезна деструктуризация отображений для тех случаев, когда вы хотите иметь необязательные (и именованные) параметры в функциях. В этом случае, объявление функции будет иметь вид

```
(defn имя-функции [обязательные параметры
         & {:keys [необязательные параметры]
            :or {значения по умолчанию}}]
тело функции)
```

например,

```
user> (defn dfunc2 [a b & {:keys [c d] :or {c "c" d "d"}}]
  (list a b c d))
user> (dfunc2 1 2 :c 3)
(1 2 3 "d")
```

Здесь мы определяем функцию, принимающую два обязательных параметра, и два необязательных, значения которых можно указывать после соответствующих ключевых символов.

Получение оригинальных значений

Деструктуризация — это полезный инструмент, но иногда необходимо получить все переданные параметры в неизменном виде. Для этого можно использовать ключевой символ `:as символ`, которая связывает указанный символ с оригинальными параметрами. Например:

```
user> (let [[a b :as c] [1 2 3]]
  (list a b c))
(1 2 [1 2 3])
user> (let [{a :a b :b :as c} {:a 1 :b 2 :c 3}]
  (list a b c))
(1 2 {:a 1, :b 2, :c 3})
```

Деструктуризация вложенных структур

Также хочется отметить, что деструктуризация может применяться и к вложенным структурам, например, если у вас есть вектор векторов, или вложенные отображения. Например,

```
user> (let [ [a [b c]] [1 [2 3]]]
  (list a b c))
(1 2 3)
user> (let [ [a [b c]] [1 2]]
  (list a b c))
Error....
user> (let [{a :a {c :c d :d} :b} {:a 1 :b {:c 3 :d 4}}]
  (list a c d))
(1 3 4)
```

Но тут стоит быть осторожным, поскольку если вместо вектора вы передадите объект несовместимого типа, то вы получите ошибку, как во втором примере.

Дополнительные примеры вы можете найти на данной странице. Также хорошее описание деструктуризации параметров можно найти в книге "Clojure in Action".

Управляющие конструкции

Для управления потоком выполнения программы в Clojure имеется некоторое количество специальных форм, на основе которых затем строятся остальные примитивы языка, управляющие выполнением программы.

Для организации последовательного выполнения нескольких выражений существует специальная форма `do`, которой передаются выражения, которые вычисляются последовательно, и результат вычисления последнего выражения возвращается как результат `do`. Использование `do` похоже на использование `let` без объявления переменных. `do` часто используется в ветках условных выражений, когда необходимо выполнить несколько выражений вместо одного.

Условные операторы

Для обработки простых условий используется конструкция `if`, которая является специальной формой. На основе `if` затем строятся остальные конструкции — `when`, `when-not`, `cond` и т.д. `if` выглядит стандартно для Lisp'образных языков: `(if условие t-выражение f-выражение?)`, т.е. если `условие` вычисляется в истинное значение, то выполняется выражение `t-выражение`, иначе — выражение `f-выражение` (оно может не указываться, что используется в макросе `when`). Результаты вычисления одного из этих выражений возвращаются в качестве результата `if`.

Макрос `cond` позволяет проверить сразу несколько условий. По своему синтаксису он отличается от `cond` в Common Lisp и Scheme, и в общем виде записывается как `(cond условие1 выражение1 условие2 выражение2 ... :else выражение-по-умолчанию)`. Заметьте, что дополнительных скобок вокруг пары `условиеN выражениеN` не требуется.

Например:

```
(defn f1 [n]
  (cond
    (number? n) "number"
    (string? n) "string"
    :else "unknown"))
```

выражение `:else` не является ключевым словом языка и введено исключительно для удобства использования — вместо него можно использовать значение `true`.

Циклы и рекурсивные функции

Для организации циклов Clojure имеет несколько специальных форм, функций и макросов. Поскольку JVM имеет некоторые ограничения, не позволяющие реализовать оптимизацию хвостовых вызовов (Tail Call Optimization, TCO), то это накладывает ограничения на способ реализации некоторых алгоритмов, которые обычно реализуются через TCO в Scheme и других языках, поддерживающих эту оптимизацию.

Явные циклы организуются с помощью специальных форм `loop` и `recur`. Объявление `loop` похоже на `let`, но при этом имеется возможность повторного выполнения выражений, путем вызова `recur`[6] с тем же числом аргументов, которые были объявлены в списке переменных `loop` — обычно это новые значения цикла. Вот простой пример — реализация функции вычисления факториала с помощью `loop/recur` (здесь нет проверки на отрицательный аргумент):

```
(defn factorial[n]
  (loop [cnt n
         acc 1]
    (if (zero? cnt)
      acc
      (recur (dec cnt) (* acc cnt)))))
```

В данном случае объявляется цикл с двумя переменными `cnt` и `acc`, которые получают начальные значения `n` и `1`. Цикл прекращается, когда `cnt` будет равен нулю — в этом случае возвращается накопленное значение, хранящееся в переменной `acc`. Если `cnt` больше нуля, то цикл начинается снова, уменьшая значение `cnt`, и увеличивая значение `acc`.

В большинстве случаев явный цикл по элементам последовательности можно заменить на вызов функций `reduce`, `map`, `filter` или макроса раскрытия списков (list comprehension) `for`. Функция `reduce` реализует операцию "свертка" (fold), и используется при реализации многих функций, таких как функции `+`, `-`, `import` и т.д. Для примера с факториалом, код становится значительно проще, чем в предыдущем примере:

```
(defn fact-reduce [n]
  (reduce * (range 1 (inc n))))
```

Существует еще один метод оптимизации потребления стека при использовании взаимно рекурсивных функций — функция `trampoline`. Она получает в качестве аргумента функцию и аргументы для нее, и если переданная функция возвращает в качестве результата функцию, то возвращенная функция вызывается уже без аргументов. Вот пример определения четности числа, написанная для использования с `trampoline`:

```
(declare t-odd? t-even?)
(defn t-odd? [n]
  (if (= n 0)
    false
    #(t-even? (dec n))))
(defn t-even? [n]
  (if (= n 0)
    true
    #(t-odd? (dec n))))
```

Единственными отличиями от "стандартных" версий является то, что функции возвращают анонимные функции (строки 5 и 9). Если мы вызовем одну из этих функций с большим аргументом, например, вот так: `(trampoline t-even? 1000000)`, то вычисление произойдет без ошибки переполнения стека, в отличие от версии, которая не использует `trampoline`.

Стоит также отметить, что достаточно часто рекурсивные функции можно преобразовать в функции, производящие "ленивые" последовательности, как это было показано в разделе "Ленивые" структуры данных. Это положительно сказывается на производительности кода и потреблении памяти.

Исключения

Clojure поддерживает работу с исключениями (exceptions), которые часто используются в коде на Java. Специальная форма `throw` в качестве аргумента получает выражение, результат вычисления которого будет использован в качестве объекта-исключения.

Для выполнения выражений и перехвата исключений, которые могут возникнуть во время выполнения кода, используется специальная форма `try`. Форма `try` записывается следующим образом: `(try выражения* (catch имя-класса аргумент выражения*)* (finally выражения*)?)`. Блоки `catch` позволяют обрабатывать разные исключения в одном выражении `try`. А форма `finally` может использоваться для выражений, которые должны выполняться, и для случаев нормального завершения кода, и если произошел выброс исключения — например, для закрытия файлов и подобных этому задач.

Если мы введем следующий код:

```
(try
  (/ 1 0)
  (println "not executed")
  (catch ArithmeticException ex
    (println (str "exception caught... " ex)))
  (finally (println "finally is called")))
```

то на экран будет выведено следующее:

```
exception caught... java.lang.ArithmeticException: Divide by zero
finally is called
```

т.е. мы перехватили исключение и вывели его на экран, а затем выполнили выражение, указанное в блоке `finally`. При этом выражения, стоящие после строки приводящей к ошибке — `(println "not executed")`, не выполняются.

Функции

Функции в общем случае создаются с помощью макроса `fn`. Для объявления функций на верхнем ("глобальном") уровне пространства имен используются макросы `defn` или `defn-`, которые раскрываются в запись вида `(def имя-функции (fn ...))`. Второй макрос отличается от первого только тем, что функция будет видна только в текущем пространстве имен. Например, следующие объявления являются одинаковыми:

```
(def  func1 (fn [x y] (+ x y)))
(defn func2 [x y] (+ x y))
```

В общем виде объявление функции с помощью `fn` выглядит как `(fn имя? [аргументы*] условия? выражения+)`. Функция также может иметь разные наборы аргументов (разное число параметров) — тогда объявление будет выглядеть как `(fn имя? ([аргументы*] условия? выражения+)+)`.

Например, объявление функции:

```
(defn func3
  ([x] "one argument")
  ([x y] "two arguments")
  ([x y z] "three arguments"))
```

позволяет вызывать ее с одним, двумя или тремя аргументами[8]. Программист также может определить функцию, имеющую переменное число параметров, если укажет знак амперсанд (&) перед аргументом, в который будут помещены оставшиеся параметры. Например, следующий код:

```
user> (defn func4 [x y & z] z)
user> (func4 1 2)
nil
user> (func4 1 2 3 4)
(3 4)
```

объявит функцию `func4`, имеющую два обязательных параметра — `x` и `y`, а остальные параметры будут помещены в список, который будет передан как аргумент `z`. См. раздел Деструктуризация параметров.

Для объявления небольших анонимных функций используется специальная запись `#(выражения+)`, а доступ к аргументам производится с помощью специальных переменных `%1`, `%2` и т.д., или просто `%`, если функция принимает один параметр. Например, следующие выражения эквивалентны:

```
(#(+ %1 %2) 10 20)
((fn [x y] (+ x y)) 10 20)
```

оба этих выражения возвращают одно и то же число. Специальная запись удобна, когда вам надо передать функцию в качестве аргумента, например, для функции `map` или `filter`.

Начиная с версии 1.1, при объявлении функции можно указывать пред- и постусловия, которые будут применяться к аргументам и результату. Эта функциональность реализует концепцию "контрактного программирования". Пред- и постусловия задаются как метаданные `:pre` и `:post`, которые указываются после списка аргументов. Каждое из условий состоит из вектора анонимных функций, которые должны вернуть `false` в случае ошибочных данных. Например, рассмотрим следующую функцию:

```
(defn constrained-sqr [x]
  {:pre  [(pos? x)]
   :post [(> % 16), (< % 225)]}
  (* x x))
```

Данная функция принимает в качестве аргументов только положительные числа — условие `(pos? x)`, в диапазоне `5..14` — условие `(> % 16), (< % 225)`, иначе будет выдана ошибка проверки аргументов или результата.

В Clojure имеется набор функций, которые позволяют создавать новые функции на основе существующих. Функция `partial` используется для создания функций с меньшим количеством аргументов путем подстановки части параметров (каррирование), а функция `comp` создает новую функцию из нескольких функций (композиция функций):

```
user> (defn sum [x y] (+ x y))
user> (def add-5 (partial sum 5))
user> (add-5 10)
15
user> (def my-second (comp first rest))
user> (my-second [1 2 3 4 5])
2
```

В первом примере мы создаем функцию, которая будет прибавлять число `5` к переданному ей аргументу. А во втором примере, мы создаем функцию, эмулирующую функцию `second`, которая сначала применяет функцию `rest` к переданным ей аргументам, а затем применяет к результатам функцию `first`.

Макросы

Макросы — это мощное средство уменьшения сложности кода, позволяющие строить проблемно-ориентированную среду на основе базового языка. Макросы активно используются в Clojure, и множество конструкций, составляющих язык, определены как макросы на основе ограниченного количества специальных форм и функций, реализованных в ядре языка.

Макросы в Clojure смоделированы по образцу макросов в Common Lisp, и являются функциями, которые выполняются во время компиляции кода. В результате выполнения этих функций должен получиться код, который будет подставлен на место вызова макроса. Основная разница заключается в синтаксисе.

В общем виде определение макроса выглядит следующим образом:

```
(defmacro name doc-string? attr-map? ([params*] body)+)
```

описание макроса (документация) — `doc-string?` и список атрибутов — `attr-map?` являются не обязательными, а список параметров и тело макроса могут указываться несколько раз, что позволяет определять макросы с переменным числом аргументов также, как и при объявлении функций (см. пример ниже).

Тело макроса должно представлять собой список выражений языка, результат выполнения которых будет подставлен на место использования макроса в виде списка Clojure, содержащего набор операций. Этот список может быть сформирован с помощью операций над списками — этот подход используется в простых случаях, так что вы можете сформировать тело макроса, используя операции `list`, `cons` и т.д. Хорошим примером этого подхода является макрос `when`, показанный ниже.

Другим подходом является маскирование (quote) всего выражения, с раскрытием только нужных частей кода. Для этого используется специальный синтаксис записи `` ` `` (обратная кавычка), внутри которого можно использовать `~` (тильда) для подстановки значений (аналогично операции `,` (запятая) в Common Lisp). Для подстановки списка не целиком, а поэлементно, используется синтаксис: `~@`. Хорошим примером второго подхода является макрос `and`, приведенный далее.

При работе с макросами очень полезными являются функции `macroexpand-1` и `macroexpand`, которые производят раскрытие заданного макроса, что позволяет программисту проверять корректность кода, используемого в макросах. Отличие между этими функциями заключается в том, что первая функция раскрывает макрос один раз, выполняя подстановки, но возможно возвращая код, который использует другие макросы. В то время как функция `macroexpand` — раскрывает макрос рекурсивно, раскрывая все использованные макросы.

Рассмотрим подстановки имен и значений более детально. Допустим, у нас есть две переменные — `x` со значением `2`, и `y`, представляющая собой список из трех элементов — `(4 5 6)`. Если мы попытаемся раскрыть разные выражения, то мы будем получать разные результаты:

```
user> (def x 2)
user> (def y '(4 5 6))
user> `(list 1 x 3)
(list 1 user/x 3)
user> `(list 1 ~x 3)
(list 1 2 3)
user> `(list 1 ~y 3)
(list 1 (4 5 6) 3)
user> `(list 1 ~@y 3)
(list 1 4 5 6 3)
```

В первом случае мы не выполняем никакой подстановки, поэтому `x` подставляется как символ. Во втором случае мы раскрываем выражение, подставляя значение символа и получая выражение `(list 1 2 3)`. В третьем случае у нас подставляется значение символа `y` в виде списка, в отличие от четвертого выражения, когда значение списка поэлементно подставляется (spliced) в выражение в раскрытом виде.

Примеры макросов

В качестве простого примера рассмотрим макрос `when`, определенный в базовой библиотеке:

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  [test & body]
  (list 'if test (cons 'do body)))
```

Данный макрос принимает один обязательный аргумент — условие `test`, а остальные аргументы рассматриваются как набор выражений, которые будут выполнены, если условие вернет истинное значение. Для того, чтобы можно было указать несколько выражений в качестве тела макроса, они обертываются в конструкцию `do`. Если мы воспользуемся `macroexpand` для раскрытия макроса, то для конструкции вида:

```
(when (pos? a)
   (println "positive") (/ b a))
```

мы получим следующий код:

```
(if (pos? a)
  (do
    (println "positive")
    (/ b a)))
```

`when` — это достаточно простой макрос.

Более сложные макросы могут создавать переменные, иметь разное количество аргументов, и т.д. Например, макрос `and`, определенный следующим образом:

```
(defmacro and
  "Evaluates exprs one at a time, from left to right.
  If a form returns logical false (nil or false), and
  returns that value and doesn't evaluate any of the
  other expressions, otherwise it returns the value
  of the last expr. (and) returns true."
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

имеет три разных раскрытия для разного количества аргументов, при этом, если макросу передано больше одного аргумента, то он рекурсивно вызывает сам себя.

Мы можем увидеть это, раскрыв макрос для разных наборов аргументов:

```
user> (macroexpand '(and ))
true
user> (macroexpand '(and (= 1 2)))
(= 1 2)
user> (macroexpand '(and (= 1 2) (= 3 3)))
(let* [and__4457__auto__ (= 1 2)] (if and__4457__auto__
(clojure.core/and (= 3 3)) and__4457__auto__))
```

В этом примере вызов макроса без параметров приводит к подстановке значения `true`. При использовании одного параметра-выражения подставляется само выражение. А если параметров больше одного, то формируется форма `let`, в которой вычисляется первое выражение и связывается с переменной с уникальным именем (сгенерированным автоматически), а затем проверяется значение этой переменной. Если это значение истинное, то макрос вызывается еще раз, получая в качестве аргументов список параметров без первого элемента. А в том случае, если выражение не истинное, возвращается результат вычисления.

В макросе `and` для избежания конфликтов с кодом пользователя используется генерация уникальных имен переменных. Для этого используется специальный синтаксис `prefix#`, который создает уникальное имя, начинающееся с заданного префикса (в нашем случае имя начинается с `and`).