АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 13

ПЛАН

- Clojure

ОБЩЕЕ ОПИСАНИЕ

- Нишевый язык
- Аналог Lisp/Scheme/Racket
- Программа - S-выражение
- Можно работать как с данными

КАК НАЧАТЬ РАБОТАТЬ

- Вариант 1: скачать с официального сайта REPL-среду
- Вариант 2: IDEA
- Плагин Cursive
- Для начала - Bare Project / Clojure REPL / Local

HELLO, WORLD

```
1 (println "Hello")
2
3 (def x 23)
4
5 (println x)
```

СИНТАКСИС И СЕМАНТИКА

- Синтаксис предельно прост - все выражается списками
- Список - это перечень элементов в скобках
- Шаг вычисления тоже представлен списком
- Ключевой элемент семантики - вызов функции

СТРУКТУРА ПРОГРАММЫ

- В REPL удобно экспериментировать
- Но полезный код надо оформить в программу
- Это тоже делается в стиле S-выражений
- Единица декомпозиции кода - namespace (аналог модуля)

HELLO, WORLD

```
1 (ns examples.Hello
2   (:gen-class))
3
4 (println "Hello World")
```

ОПЕРАТОРЫ И ВЫЗОВЫ ФУНКЦИЙ

- Операторы мало отличимы от вызова функций
- Сложение:

```
(+ 12 33)
```

- На места операнда может быть другое выражение
- Вопрос приоритетов снимается сам собой

ОПЕРАТОРЫ И ВЫЗОВЫ ФУНКЦИЙ

- Многократное применение оператора можно сокращать
- Например, так:

```
(- (+ a 6 b 10)  d e)
```

- Для некоторых (моноидных) операций все красиво обобщается на 1 или 0 аргументов
- Для других все не так регулярно

ПРИМЕР

```
 1 (println (+ 5 3 10))
 2 (println (+ 5 3))
 3 (println (+ 5))
 4 (println (+))
 5
 6 (println (* 5 3 10))
 7 (println (* 5 3))
 8 (println (* 5))
 9 (println (*))
10
11 ; ........................
```

ПРИМЕР

```
 1 ; ..........................
 2 (println (- 5 3 10))
 3 (println (- 5 3))
 4 (println (- 5))
 5
 6 (println (/ 5 3 10))
 7 (println (/ 5 3))
 8 (println (/ 5))
 9
10 (println (/ 5.0 3 10))
11 (println (/ 5.0 3))
12 (println (/ 5.0 ))
```

ЧИСЛОВЫЕ ТИПЫ

- Аналоги JVM-целых
- Местами с плавным переходом в `BigInt`
- Числовой литерал трактуется как `Long`, если влезает
- И `BigInt`, если не влезает

ЧИСЛОВЫЕ ТИПЫ

- Явное приведение - `(short 123)`
- Операции типа `(*'` переходят в `BigInt`
- `abs` для `Long/MAX_VALUE` работает "как в Java"
- `abs'` из коробки отсутствует

СИМВОЛЫ

- Символ - универсальное понятие
- `def` привязывает символ к объекту
- Объектом может быть число, строка, функция
- Символ может включать в себя знаки операций

СИМВОЛЫ

- Операции - тоже символы, привязанные к реализации
- И их можно переприсваивать
- Если точнее - стандартные операции определены как `clojure.core.+` и т.п.
- А просто + - локальный алиас

ПРИМЕР

```clojure
(def :+: +)

(def + -)

(println (+ 5 3))

(println (:+: 5 3))

(println (clojure.core+ 5 3))

(def + :+:)

(println (+ 5 3))
```

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

- Тоже в S-выражение
- Первый элемент - `defn`
- Второй элемент - перечень символов в квадратных скобках
- Это имена параметров
- Почему в квадратных - пока примем как данность

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

- Потом идет тело функции
- Как S-выражение
- Его результат - результат функции
- Никакой типизации параметров нет

ПРИМЕР

```clojure
(def *5 [n] (* 5 n))

(println (*5 44))

(println (+ 5 3))

(println (:+: 5 3))

(println (clojure.core+ 5 3))

(def + :+:)

(println (+ 5 3))
```

РЕКУРСИЯ

- Можно рекурсивно вызывать функцию
- Нужно какое-то условие для выхода
- Есть интуитивно понятный `if`
- Формально выглядит как вызов функции

МАКРОСЫ

- Но это не функция
- Потому что у функции аргументы должны быть вычислены до вызова
- Что для if нежелательно
- Для такого есть отдельный иеханизм - макрос

ПРИМЕР

```clojure
(defn ! [n]
    (if (= n 0)
       1
       (* n (! (- n 1))))
    )
)

(! 10)
```

ХВОСТОВАЯ РЕКУРСИЯ

- В буквальном смысле - отсутствует
- Фактически реализована в более честном виде
- Специальная форма цикла
- Через объявление переменных цикла, их начальных значений и логики изменения

ПРИМЕР

```clojure
(defn ! [n]
    (loop [cnt n
           acc 1]
      (if (< cnt 2) acc
        (recur (dec cnt) (* cnt acc)))))
```

СИМВОЛ КАК ЗНАЧЕНИЕ

- Символ можно получить как значение
- У него свой отдельный тип данных
- Получить - функция `quote` или одинарная кавычка
- Разыменовать - функция `eval`
- Помогает разобраться функция `type`

ПРИМЕР

```clojure
(type 10)
(type 10.0)
(type nil)

(type +)
(type str)


(defn f [] ())
(type f)
(defn f [] ())
(type f2)

(= (type f) (type f2))
```

ПРИМЕР

```
 1 (type type)
 2 (type (type 10))
 3 (type (nil))
 4
 5 (def v1 123)
 6
 7 (type 123)
 8 (type v1)
 9 (type 'v1)
10 (type (quote v1))
11 (type 'v2345)
12 (type (quote v2345))
13
14 (def pv1 'v1)
15 (def pv2345 'v2345)
```

ПРИМЕР

```
1 (println pv1)
2 (println pv2345)
3 (println (eval pv1))
4
5 (def v2345 222)
6 (println (eval pv2345))
```

Смысл `quote`

- `quote` отменяет интерпретацию аргумента как выражения
- Можно применить к списку в круглых скобках
- И получить именно список
- Вместо попытки вызвать функцию
- А `eval` может исполнить список, хранящийся в переменной

ПРИМЕР

```clojure
(defn iffunc [cond then] (and
     cond (eval then)
    )
)

(iffunc (= 1 0) '(println "T") )
```

ПЕРЕМЕННОЕ ЧИСЛО АРГУМЕНТОВ

- В списке аргументов можно указать & и следом ровно одно имя
- Первые параметры попадут в указанные до &
- Все остальные - в указанный после

ПРИМЕР

```
1 (defn f [a b & c] (println (list a b c)))
2
3 (f 1 2 3 4 5 6)
```

ПРИЛОЖЕНИЕ

- Сделаем утилиту типа `cat/grep`
- Можем получить аргументы коиандной строки
- В специальной переменной `*command-line-args*`
- В форме списка строк

ПРИМЕР

```
1 (println *command-line-args*)
2 (println (type *command-line-args*))
3 (println (count *command-line-args*))
```

СПИСОК

- Функциональный спискок, голова и хвост
- `list` - литеральный конструктор
- `list*` - наращивающий конструктор
- `first`/`rest` - голова/хвост
- `empty?` - понятно

ПРИМЕР

```clojure
(def data '( 1  2 3))

(empty? data)
(first data)
(rest data)
(first (rest data))
(first (rest (rest data)))
(first (rest (rest (rest data))))

(first '(nil))
(first '())
(first '(()))

(identical? '() nil)
```

ПРИМЕР

```
1 (if (empty? *command-line-args*)
2     (println "no args")
3
4     (doseq [name *command-line-args*]
5        (println name)
6     )
7 )
```

РАБОТА С ФАЙЛАМИ

- Своя обертка для `JDK`
- В `clojure/io`
- Можно читать/писать целиком
- Можно по строчкам

ПРИМЕР

```
 1 (if (empty? *command-line-args*)
 2     (println "no args")
 3
 4     (doseq [name *command-line-args*]
 5         (with-open [rdr (clojure.java.io/reader name)]
 6             (doseq [line (line-seq rdr)]
 7                 (println line)
 8             )
 9         )
10     )
11 )
```

СТРОКА

- В основе - JVM-строка
- Задекорирована под clojure-стиль
- Функции сильно напоминают методы `java.lang.String`
- "Списочный" интерфейс тоже есть
- Но `(rest "hello")` не идентично другому `(rest "hello")`

ПРИМЕР: WC -L

```
1 (if (empty? *command-line-args*)
2     (println "no args")
3
4     (doseq [name *command-line-args*]
5         (with-open [rdr (clojure.java.io/reader name)]
6             (println (count (line-seq rdr)))
7         )
8     )
9 )
```

ПРИМЕР: WC -C

```
 1 (if (empty? *command-line-args*)
 2     (println "no args")
 3
 4     (doseq [name *command-line-args*]
 5         (with-open [rdr (clojure.java.io/reader name)]
 6             (println (reduce + (map (comp inc count)
 7                                     (line-seq rdr))))
 8         )
 9     )
10 )
```

ЧТО ДАЛЬШЕ

- Нам еще надо бы количество слов посчитать
- И все суммировать
- Нужно к каждой строке применить список функций
- А потом сделать "списочный" `reduce`

ПРИМЕР: WC

```clojure
(defn apply-all [fs v]
    (map (fn [f] (f v))  fs)
)

(apply-all (list inc dec) 12)
```

ПРИМЕР: WC

```
1 (defn ++ [p1 p2]
2     (list (+ (first p1) (first p2))
3           (+ (first (rest p1)) (first (rest p2)))
4     )
5 )
6
7 (++ '(1 2) '(22 33))
```

ПРИМЕР: WC

```clojure
(defn apply-all [fs v]
    (map (fn [f] (f v))  fs)
)

(defn ++ [p1 p2]
    (list (+ (first p1) (first p2))
          (+ (first (rest p1)) (first (rest p2)))
    )
)

(def sfuncs (list (fn [s] 1) count))
; ........................
```

ПРИМЕР: WC

```
1  ; ........................
2  (if (empty? *command-line-args*)
3      (println "no args")
4
5      (doseq [name *command-line-args*]
6          (with-open [rdr (clojure.java.io/reader name)]
7              (println (reduce ++
8                  (map (fn [s] (apply-all sfuncs s))
9                       (line-seq rdr))))
10         )
11     )
12 )
```

ПРИМЕР: WC

```clojure
(defn zip [vals1 vals2] (map list vals1 vals2))

(defn apply-all [fs v]
    (map (fn [f] (f v))  fs)
)

(defn ++ [p1 p2]
    (map (fn [v] (reduce + v))  (zip p1 p2))
)
; ..............................
```

ПРИМЕР: WC

```
1 ; ..............................
2 (def sfuncs (list
3           (fn [s] 1)
4           (comp count (fn [s] (clojure.string/split
5                                s #" +")))
6           (comp inc count)
7 ))
8 ; ..............................
```

ПРИМЕР: WC

```
1  ; ..............................
2  (if (empty? *command-line-args*)
3      (println "no args")
4
5      (doseq [name *command-line-args*]
6          (with-open [rdr (clojure.java.io/reader name)]
7              (println (reduce ++
8        (map (fn [s] (apply-all sfuncs s)) (line-seq rdr))))
9          )
10     )
11 )
```

ДРУГИЕ КОЛЛЕКЦИИ

- `set` - классическое множество
- `sorted-set` - упорядоченное множество
- `vector` - индексированная коллекция фиксированной длины

KEYWORD

- Часто используются "символы", начинающиеся с `:`
- Они используются для маркировки, ключей и т.п.
- Это отдельный тип данных - `Keyword`