Управляемая кодогенерация

Денис С. Мигинский

Основные режимы работы среды исполнения Lisp

- Интерпретация – режим по умолчанию
- Компиляция
- Исполнение скомпилированного кода

Компиляция

```
;режим интерпретации
(defn
;режим компиляции
  test-fn [n]
  “test-fn”))

;режим интерпретации
(defn
;режим компиляции
  -main [& args]
  (println (test-fn)))

;режим интерпретации (вероятно,
;компиляция и запуск)
(println (test-fn))
```

Запуск

```
;интерпретация кода
;<код предыдущего слайда>

;переход в режим исполнения
;скомпилированного кода
;при вызове main
(apply main args)
```

Код как данные

```
(let
;построение кода в форме структуры
;данных:
  [code (list inc 1)]
;>>(inc 1)

;принудительная интерпретация кода:
  (eval code))
;>> 2
```

quote – обратная операция по отношению к eval

```
;блокировка исполнения выражения
(quote (inc 1))

;--//-- (прямая кавычка/апостроф)
'(inc 1)

;обращение операции quote
(eval '(inc 1))
>>2
```

Вызов компилятора во время исполнения

```
(defn foo []
;определение символа во время исполнения
  (eval '(defn bar [] “bar"))
;вызов определенного символа
  (eval '(println (bar))))

;ошибка компиляции
(defn foo []
  (eval '(defn bar [] “bar"))
  (println (bar)))
;ошибка исполнения
(defn foo []
  (eval '(println (bar)))
  (eval '(defn bar [] “bar")))
```

Задача

```
(def naturals
  (lazy-seq
    (cons 1 (map inc naturals))))

;определить lazy-cons так что
;следующее определение было бы
;эквивалентно предыдущему

(def naturals
  (lazy-cons 1 (map inc naturals)))
```

Попытка 1

```
;определение скомпилируется
(defn lazy-cons [val coll]
  (lazy-seq (cons val coll)))

;ошибка компиляции
(def naturals
  (lazy-cons 1 (map inc naturals)))
```

Попытка 2

```
;определение скомпилируется
(defn lazy-cons [val coll-code]
  (list 'lazy-seq
    (list 'cons val coll-code)))
;скомпилируется,
;но приходится вызывать eval
(def naturals
  (eval (lazy-cons 1
          '(map inc naturals))))
```

Попытка 3, последняя

```
;определение скомпилируется
(defmacro lazy-cons [val coll]
  (list 'lazy-seq
    (list 'cons val coll)))

;определение скомпилируется
(def naturals
  (lazy-cons 1 (map inc naturals)))
```

Модель исполнения макроса

1. Сам макрос исполняется на стадии компиляции
2. Параметры в макрос передаются без вычисления
3. Результат исполнения макроса – код, который компилируется и во время исполнения будет выполнен в контексте, откуда макрос вызывается
4. Результат вызова макроса – исполнение кода, скомпилированного макросом

Побочный эффект: макрос не видит контекста времени исполнения, т.е. не может использовать динамических данных для генерации кода

Пример: спец. форма cond

```
;исходный текст clojure/core
;стандартной библиотеки Clojure
(defmacro cond [& clauses]
  (when clauses
    (list 'if (first clauses)
      (if (next clauses)
        (second clauses)
        (throw
          (IllegalArgumentException.
            "cond requires an even
             number of forms")))
      (cons 'clojure.core/cond
        (next (next clauses)))))))
```

macroexpand

```
;отладка макроса
(macroexpand
  '(cond
     (< a 0) "negative"
     (> a 0) "positive"
     :default "zero"))

>>
(if
  (< a 0) "negative"
  (clojure.core/cond
    (> a 0) "positive"
    :default "zero"))
```

Quote/Syntax-quote

```
;прямая кавычка(quote): исполнение
;блокируется, имена не разрешаются
'(inc 1)
>>(inc 1)

;обратная кавычка (syntax-quote):
;исполнение блокируется, имена
;разрешаются на стадии компиляции
`(inc 1)
>>(clojure.core/inc 1)

`(undefined-sym 1)
>>(user/undefined-sym 1)
```

Unquote

```
;переменная n не вычислена, eval для
;внутреннего (inc …) не сработает
(let [n 1] '(inc n))
>> (inc n)

;--//--
(let [n 1] `(inc n))
>> (clojure.core/inc user/n)

;переменная вычислена, eval сработает
(let [n 1] `(inc ~n))
>> (clojure.core/inc 1)
(let [n 1] (eval `(inc ~n)))
>> 2
```

Splicing unquote

```
(defmacro my-when [test & forms]
  `(if ~test (do ~@forms)))

(println (macroexpand
    '(my-when (> n 0)
      (println "positive")
      (println "smth. else"))))
>> (if (> n 0)
      (do (println "positive")
        (println "smth. else")))

(let [n 1]
  (my-when (> n 0)
    (println “positive”)
    (println “smth. else”)))
```

Определение символов в макросах

```
;"поломанное" определение time
;из стандартной библиотеки Clojure
(defmacro my-time [expr]
  `(let [start (. System (nanoTime))
         ret ~expr]
    (prn
      (str "Elapsed time: "
        (/ (double
             (- (. System (nanoTime))
               start))
          1000000.0) "msecs"))
     ret))
>> Can't let qualified name:
       ru.nsu.fit.dt.mc/start
```

Определение my-time, попытка 2

```
(defmacro my-time [expr]
  `(let [~'start (. System (nanoTime))
         ~'ret ~expr]
     (prn
       (str "Elapsed time: "
         (/ (double
              (- (. System (nanoTime))
                 ~'start))
            1000000.0) "msecs"))
     ~'ret))

(let [n 5]
  (my-time (println n)))
>> 5
"Elapsed time: 10.132752 msecs"
```

Проблема с определением my-time

```
(let [start 5]
  (my-time (println start)))

>> 1334408170241579
"Elapsed time: 2.384257 msecs"
```

Правильное определение time

```
;правильное определение time
;исходный текст clojure/core
;стандартной библиотеки Clojure
(defmacro time [expr]
  `(let [start# (. System (nanoTime))
         ret# ~expr]
     (prn
       (str "Elapsed time: "
         (/ (double
              (- (. System (nanoTime))
                 start#))
            1000000.0) "msecs"))
     ret#))
```

Генерация символов в макросах

```
;символ с квалификацией пространства
;имен, не может использоваться в let
`start
>> user/start

;может использоваться в let
;но возможны пересечения с символами
;из контекста вызова макроса
'start
>> start

;вероятность пересечения минимальна
>> start#
start__1322__auto__
```

Задача: инфиксная арифметика

Необходимо реализовать поддержку инфиксных операций.

Набор операций должен быть расширяем.

Анализ задачи

Поддержка инфиксной арифметики должна разрешаться локально, в противном случае некоторые конструкции Clojure могут не сработать (например, `(map - arr)`

По той же причине необходимо иметь возможность локально отменять действие инфиксной арифметики.

Из требование расширяемости следует, что алгоритм должен быть универсальным. Алгоритму некоторым внешним образом можно задавать набор операций с приоритетами.

Для простоты предполагаем что все операции левоассоциативные.

Использование

```
;;префиксная форма по умолчанию
(+ 1
   (with-infix-ops
     ;;включаем инфиксную форму
     (a * b + c * 2 +
        (with-prefix-ops
          ;;отключаем инфиксную форму
          (+ x y)))
```

Определение операций

```
(def standard-ops
  [{'+ '+,
    '- '-}
   {'* '*,
    '/ '/},
   {'** 'Math/pow}])
```

Определение макроса

```
(defn- infix-form-process [all-ops form] …)

(defmacro with-infix-ops [& forms]
  `(do
     ~@(map (partial
               infix-form-process
               standard-ops)
            forms)))
```

Преобразование формы

```
(defn- infix-form-process [all-ops form]
  (cond
    (vector? form)
      (mapv (partial
              infix-form-process all-ops)
              form)
    ;;missed the same for maps and sets
    (list? form)
      (infix-list-process
        all-ops all-ops form)
    :default form))
```

Преобразование списка

```
(defn- infix-list-process
    [all-ops current-ops list-form]
  (let [[ops & rest-ops] current-ops]
    (cond
      (= (first list-form) 'with-prefix-ops)
        (cons 'do (rest list-form))

      (not (empty? ops))
        ;;most important skipped here,
        ;;see next slide

      :default
        (map
          (partial infix-form-process all-ops)
          list-form))))
```

Преобразование операций

```
;;remember about left-associative
(let [[inv-tail-form [op & inv-head-form]]
        (split-with
          (comp not (partial contains? ops))
          (reverse list-form))
       head-form (reverse inv-head-form)
       tail-form (reverse inv-tail-form)]
…
```

Преобразование операций

```
;;remember about left-associative
(let [op … head-form … tail-form …]
  (if (nil? op)
    (recur all-ops rest-ops tail-form)
    (list
      (ops op)
      (if (== 1 (count head-form))
        (infix-form-process
          all-ops (first head-form))
        (infix-list-process
          all-ops current-ops head-form))
      (if (== 1 (count tail-form))
        (infix-form-process
          all-ops (first tail-form))
        (infix-list-process
          all-ops rest-ops tail-form)))))
```

Побочный эффект

```
;;It works! Why?
(let [x 42.]
  (println
    (with-infix-ops
      (Math/sin x ** 2 +
       Math/cos x ** 2))))

>> 1.0
```