Лекция 10

Абстракция времени

Функциональное и логическое программирование

КамчатГТУ, 2013 г.

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

Функциональное программирование

Вычислительный процесс – аппликация функций к данным.

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

```
while (! m == 0) {
  int k = n;
  n = m;
  m = k mod m;
}
```

Функциональное программирование

Вычислительный процесс – аппликация функций к данным.

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

```
while (! m == 0) {
  int k = n;
  n = m;
  m = k mod m;
}
```

Функциональное программирование

Вычислительный процесс – аппликация функций к данным.

```
gcd n m = gcd m (n mod m)
gcd n 0 = n
```

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

```
while (! m == 0) {
  int k = n;
  n = m;
  m = k mod m;
}

int i = n;
int sum = 0;
while (i > 0) {
  sum = sum + i ** 2;
  i = i - 1;
}
```

Функциональное программирование

Вычислительный процесс – аппликация функций к данным.

```
gcd n m = gcd m (n mod m)
gcd n 0 = n
```

Понятие времени в программировании

Императивное программирование

Вычислительный процесс – *последовательность смены состояния* информационной среды

```
while (! m == 0) {
  int k = n;
  n = m;
  m = k mod m;
}
```

```
int i = n;
int sum = 0;
while (i > 0) {
  sum = sum + i ** 2;
  i = i - 1;
}
```

Функциональное программирование

Вычислительный процесс – аппликация функций к данным.

```
gcd n m = gcd m (n mod m)
gcd n 0 = n
```

```
sumsqr n = iter n 0
iter i sum = iter (i − 1) (sum + i^2)
iter 0 sum = sum
```

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ
- возможность существенной оптимизации

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ
- возможность существенной оптимизации
- простота параллельных вычислений

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ
- возможность существенной оптимизации
- простота параллельных вычислений

Недостатки

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ
- возможность существенной оптимизации
- простота параллельных вычислений

Недостатки

- сложность моделирования объектов, имеющих состояние

Понятие времени в программировании

В чистом функциональном программировании понятие времени отсутствует: последовательность вычислений не определена и на результат не влияет.

Строгие вычисления:

```
sqr (sqr 2)
sqr (2 * 2)
sqr 4
4 * 4
16
```

Ленивые вычисления:

```
sqr (sqr 2)
(sqr 2) * (sqr 2)
(2 * 2) * (2 * 2)
4 * 4
16
```

```
(- (read) (read)) = ?
(length (list (read) (read))) = 2
```

Достоинства

- высокая степень модульности
- простота верификации программ
- возможность существенной оптимизации
- простота параллельных вычислений

Недостатки

- сложность моделирования объектов, имеющих состояние
- сложность задания определённой последовательности действий (ввод-вывод, взаимодействие с пользователем (клиент-сервер) и т.п.)

Способы определения последовательности вычислений

- Отказ от чистоты языка: введение императивных конструкций и изменяемых состояний. (SCHEME, STANDARD ML, NEMERLE, F# и т.д.

Способы определения последовательности вычислений

- Отказ от чистоты языка: введение императивных конструкций и изменяемых состояний. (SCHEME, STANDARD ML, NEMERLE, F# и т.д.
- Использование стиля передачи продолжений (CPS). (SCHEME, HASKEL, STANDARD ML и т.д.)

Способы определения последовательности вычислений

- Отказ от чистоты языка: введение императивных конструкций и изменяемых состояний. (SCHEME, STANDARD ML, NEMERLE, F# и т.д.
- Использование стиля передачи продолжений (CPS). (SCHEME, HASKEL, STANDARD ML и т.д.)
- Использование генераторов. (SCHEME, HASKEL и т.д.)

Способы определения последовательности вычислений

- Отказ от чистоты языка: введение императивных конструкций и изменяемых состояний. (Scheme, Standard ML, Nemerle, F# и т.д.
- Использование стиля передачи продолжений (CPS). (Scheme, Haskel, Standard ML и т.д.)
- Использование генераторов. (Scheme, Haskel и т.д.)
- Использование монад. (Haskel, реализуемы в Scheme, Perl, Clojure, Scala, F#)

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
55
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
55
> (for ([i '(1 2 3 4 5 6)])
    (set! sum (+ (sqr i) sum)))
> s
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
55
> (for ([i '(1 2 3 4 5 6)])
    (set! sum (+ (sqr i) sum)))
> s
110
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
55
> (for ([i '(1 2 3 4 5 6)])
    (set! sum (+ (sqr i) sum)))
> s
110
```

Охраняющие операторы: **`when`**, **`unless`**

```
> (for ([i (in-naturals)])
    (when (= i 6) (error "stop!"))
    (display i))
```

Императивные конструкции в Scheme

- Оператор присваивания: `set!`

```
> (define x 3)
> (set! x (+ x 2))
> x
5
```

- Оператор последовательности: **`begin`**

```
> (begin
    (display "The value of x is ")
    (display x)
    (set! x (+ x 2))
    (newline)
    (display "The new value of x is ")
    (display x))
The value of x is 5
The new value of x is 7
```

- Итератор: **`for`**

```
> (define sum 0)
> (for ([i 6])
    (set! sum (+ (sqr i) sum)))
> s
55
> (for ([i '(1 2 3 4 5 6)])
    (set! sum (+ (sqr i) sum)))
> s
110
```

Охраняющие операторы: **`when`**, **`unless`**

```
> (for ([i (in-naturals)])
    (when (= i 6) (error "stop!"))
    (display i))
12345
stop!
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

```
> (define A (new-account 100))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

```
> (define A (new-account 100))
> (define B (new-account 50))
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

```
> (define A (new-account 100))
> (define B (new-account 50))
> (A 10)
110
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

```
> (define A (new-account 100))
> (define B (new-account 50))
> (A 10)
110
> (B -30)
20
```

Объекты с внутренним состоянием

Простой счётчик

```
(define (new-counter)
  (define i 0)
  (λ () (set! i (+ i 1)) i))
```

```
> (define c (new-counter))
> (c)
1
> (c)
2
> (define c2 (new-counter))
> (c2)
1
> (c)
3
```

Моделирование банковских счетов

```
(define (new-account init)
  (define (balance init))
  (λ (change)
    (cond
     [(< balance change) (error "Not enough money!")]
     [else (set! balance (+ balance change))
           balance])))
```

```
> (define A (new-account 100))
> (define B (new-account 50))
> (A 10)
110
> (B -30)
20
> (B -30)
Not enough money!
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
> (g)
1
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
> (g)
1
> (g)
2
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
> (g)
1
> (g)
2
> (g)
1
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
> (g)
1
> (g)
2
> (g)
1
> (g)
2
```

Объекты с внутренним состоянием

Простейший генератор

```
(define (in-values . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) 'stop]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-values 1 2 3))
> (g)
1
> (g)
2
> (g)
3
> (g)
stop
```

Циклический генератор

```
(define (in-cycle . lst)
  (define x lst)
  (λ ()
    (cond
     [(empty? x) (set! x lst)]
     [else (set! x (cdr x))
           (car x)])))
```

```
> (define g (in-cycle 1 2))
> (g)
1
> (g)
2
> (g)
1
> (g)
2
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
> (foldl +. 0 '(1 2 3 4 5))
15
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
> (foldl +. 0 '(1 2 3 4 5))
15
> (+. 'count)
7
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
> (foldl +. 0 '(1 2 3 4 5))
15
> (+. 'count)
7
```

Контекстный счётчик

```
(define-syntax-rule (count (f ...) expr)
  (let ([f (counting f)] ...)
    (let ([res expr])
      (printf "~a\t~a\n" 'f (f 'count)) ... res)))
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
> (foldl +. 0 '(1 2 3 4 5))
15
> (+. 'count)
7
```

Контекстный счётчик

```
(define-syntax-rule (count (f ...) expr)
  (let ([f (counting f)] ...)
    (let ([res expr])
      (printf "~a\t~a\n" 'f (f 'count)) ... res)))
```

```
> (count (+)
    (foldl + 0 '(1 2 3 4 5)))
#<procedure:+>: 5
15
```

Объекты с внутренним состоянием

Счётчик применений заданной функции

```
(define (counting f)
  (define c 0)
  (/. 'reset --> (set! c 0)
      'count --> c
      x ...  --> (let ([res (apply f x)])
                   (set! c (+ 1 c))
                   res)))
```

```
> (define +. (counting +))
> (+. 2 (* 4 (+. 5 3)))
34
> (+. 'count)
2
> (foldl +. 0 '(1 2 3 4 5))
15
> (+. 'count)
7
```

Контекстный счётчик

```
(define-syntax-rule (count (f ...) expr)
  (let ([f (counting f)] ...)
    (let ([res expr])
      (printf "~a\t~a\n" 'f (f 'count)) ... res)))
```

```
> (count (+)
    (foldl + 0 '(1 2 3 4 5)))
#<procedure:+>: 5
15

> (count (+ *)
     (+ 3 (* 5 (+ 2 3))))
#<procedure:+>: 2
#<procedure:*>: 1
28
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
> (p+ 5)
10
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
> (p+ 5)
10
> (p 7)
> (p)
7
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
> (p+ 5)
10
> (p 7)
> (p)
7
> (p+ 5)
12
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
> (p+ 5)
10
> (p 7)
> (p)
7
> (p+ 5)
12
> (with (p 100) (p+ 5))
105
```

Объекты с внутренним состоянием

Простейшая реализация глобального параметра

```
(define (parameter val)
  (case-λ
    [() val]
    [(x) (set! val x)]))

(define-syntax-rule (with (p val) expr ...)
  (begin
    (define old-p (p))
    (p val)
    (define res expr ...)
    (p old-p)
    res))
```

```
(define p (parameter 5))
(define p+ (+ (p)))
> (p)
5
> (p+ 5)
10
> (p 7)
> (p)
7
> (p+ 5)
12
> (with (p 100) (p+ 5))
105
> (p+ 5)
12
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

```
> (define (f x) (- 2 (sqr x)))
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

```
> (define (f x) (- 2 (sqr x)))

> (bisection f 1 2)
1.4142074584960938
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

```
> (define (f x) (- 2 (sqr x)))

> (bisection f 1 2)
1.4142074584960938

> (roundof (bisection f 1 2))
1.41421
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

```
> (define (f x) (- 2 (sqr x)))

> (bisection f 1 2)
1.4142074584960938

> (roundof (bisection f 1 2))
1.41421

> (roundof (sqrt 2))
1.41421
```

Объекты с внутренним состоянием

Использование глобального параметра

```
(define (bisection f a b)
  (and (<= (* (f a) (f b)) 0)
       (let ([c (* 0.5 (+ a b))])
         (if (almost-equal? a b)
             c
             (or (bisection f a c)
                 (bisection f c b))))))

(define tolerance (parameter 1e-5))

(define (almost-equal? a b)
  (< (- b a) (tolerance)))

(define (roundof x)
  (exact->inexact
   (* (inexact->exact
       (round (/ x (tolerance))))
      (tolerance))))
```

```
> (define (f x) (- 2 (sqr x)))

> (bisection f 1 2)
1.4142074584960938

> (roundof (bisection f 1 2))
1.41421

> (roundof (sqrt 2))
1.41421

> (with (tolerance 1e-14)
    (define x (bisection f 1 2))
    (displayln (roundof x))
    (almost-equal? x (sqrt 2)))
1.4142135623731
#t
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.
- Постусловия – область значений.

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.
- Постусловия – область значений.

Функции с состоянием

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.
- Постусловия – область значений.

Функции с состоянием

- Предусловия – область определения + состояние ИС.

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.
- Постусловия – область значений.

Функции с состоянием

- Предусловия – область определения + состояние ИС.
- Постусловия – область значений + состояние ИС.

Проблемы с верификацией императивных программ

Определим счётчики

```
(define (counter)
  (define i 0)
  (λ (n) (set! i (+ i n)) i))

(define a (counter))
(define b (counter))
```

Чему равны следующие выражения?

```
(a 1)
(a (a 1))
(a (a 0))
(a (- (a 1)))
(a (- (b 1)))
(map a '(1 2 3 4))
```

При наличии изменяемых состояний невозможно сказать чему равно выражение, вне контекста выполнения программы!

Чистые функции

- Предусловия – область определения.
- Постусловия – область значений.

Функции с состоянием

- Предусловия – область определения + состояние ИС.
- Постусловия – область значений + состояние ИС.

Области определения и значения могут быть не согласованы с состоянием ИС!

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,
- Web-приложения,

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,
- Web-приложения,
- генераторы и итераторы,

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,
- Web-приложения,
- генераторы и итераторы,
- поиск с возвратом, недетерминистические вычисления,

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,
- Web-приложения,
- генераторы и итераторы,
- поиск с возвратом, недетерминистические вычисления,
- сопрограммы (coroutines),

Продолжения

Определение

Продолжение (continuation) – абстрактное представление потока управления программы. Даёт возможность полного контроля над вычислительным процессом и состоянием ИС (является обобщением `goto`).

- Появились, в языке ALGOL 60, как альтернатива оператору `goto`.
- Впервые появились, как объект первого класса в языке SCHEME.
- Сейчас есть практически во всех функциональных и императивных языках.

Применение продолжений:

- реализация механизма исключений и многопоточности,
- Web-приложения,
- генераторы и итераторы,
- поиск с возвратом, недетерминистические вычисления,
- сопрограммы (coroutines),
- компиляция.

Введение в продолжения

Пусть задана последовательность вычислений:

```
norm(x, y) =
  a := x * x
  b := y * y
  c := a + b
  return sqrt(c)
```

Введение в продолжения

Пусть задана последовательность вычислений:

```
norm(x, y) =
  a := x * x
  b := y * y
  c := a + b
  return sqrt(c)
```

В функциональном языке последовательность выполения может быть задана формой **let**

```
(define (norm x y)
  (let* ([a (* x x)]
         [b (* y y)]
         [c (+ a b)]) (sqrt c)))
```

Введение в продолжения

Пусть задана последовательность вычислений:

```
norm(x, y) =
  a := x * x
  b := y * y
  c := a + b
  return sqrt(c)
```

В функциональном языке последовательность выполения может быть задана формой **let**

```
(define (norm x y)
  (let* ([a (* x x)]
         [b (* y y)]
         [c (+ a b)]) (sqrt c)))
```

Макрос **let*** раскрывается во вложенные формы **let**:

```
(define (norm x y)
  (let ([a (* x x)])
    (let ([b (* y y)])
      (let ([c (+ a b)])
       (sqrt c)))))
```

Введение в продолжения

Пусть задана последовательность вычислений:

```
norm(x, y) =
  a := x * x
  b := y * y
  c := a + b
  return sqrt(c)
```

В функциональном языке последовательность выполения может быть задана формой **let**

```
(define (norm x y)
  (let* ([a (* x x)]
         [b (* y y)]
         [c (+ a b)]) (sqrt c)))
```

Макрос **let*** раскрывается во вложенные формы **let**:

```
(define (norm x y)
  (let ([a (* x x)])
    (let ([b (* y y)])
      (let ([c (+ a b)])
        (sqrt c)))))
```

Макрос **let** раскрывается в виде аппликаций **λ**-функций:

```
(define (norm x y)
  ((λ (a)
     ((λ (b)
        ((λ (c)
           (sqrt c)))
         (+ a b)))
      (* y y)))
   (* x x))
```

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

С его помощью функция `norm` запишется так:

```
(define (norm x y)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         (sqrt c))))))))
```

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

С его помощью функция `norm` запишется так:

```
(define (norm x y)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                          (sqrt c))))))))
```

Приведём это определение к единообразной форме:

```
(define (norm& x y k)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                          ((& sqrt) c k))))))))
```

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

С его помощью функция `norm` запишется так:

```
(define (norm x y)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         (sqrt c))))))))
```

Приведём это определение к единообразной форме:

```
(define (norm& x y k)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         ((& sqrt) c k))))))))
```

Последнее определение сделано в **стиле передачи продолжений** (continuation-passig style, CPS).

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

С его помощью функция `norm` запишется так:

```
(define (norm x y)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         (sqrt c))))))))
```

Приведём это определение к единообразной форме:

```
(define (norm& x y k)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         ((& sqrt) c k))))))))
```

Последнее определение сделано в **стиле передачи продолжений** (continuation-passig style, CPS).

- Все функции имеют дополнительный аргумент – продолжение, указывающий на то куда передаётся результат вычисления функции.

Введение в продолжения

Введём оператор `&` для постфиксной нотации:

```
(define/. (& f)
  x ... k --> (k (apply f x)))
```

С его помощью функция `norm` запишется так:

```
(define (norm x y)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         (sqrt c))))))))
```

Приведём это определение к единообразной форме:

```
(define (norm& x y k)
  ((& *) x x (λ (a)
               ((& *) y y (λ (b)
                            ((& +) a b (λ (c)
                                         ((& sqrt) c k))))))))
```

Последнее определение сделано в **стиле передачи продолжений** (continuation-passig style, CPS).

- Все функции имеют дополнительный аргумент – продолжение, указывающий на то куда передаётся результат вычисления функции.
- При наличии оптимизации хвостовой рекурсии (TCO) стек не используется.

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
>(fact& 1 #<procedure:...>)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
>(fact& 1 #<procedure:...>)
>(fact& 0 #<procedure:...>)
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
>(fact& 1 #<procedure:...>)
>(fact& 0 #<procedure:...>)
<24
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
>(fact& 1 #<procedure:...>)
>(fact& 0 #<procedure:...>)
<24
24
```

Программирование в CPS

Рассмотрим определение для факториала:

```
(define/. fact
  0 --> 1
  (+ 1 n) --> (* (+ 1 n)
                 (fact n)))
```

Его выполнение использует стек:

```
>(fact 4)
> (fact 3)
> >(fact 2)
> > (fact 1)
> > >(fact 0)
< < <1
< < 1
< <2
< 6
<24
24
```

Перепишем это определение в CPS:

```
(define/. fact
  0       return --> (return 1)
  (+ 1 n) return -->
  ((& -) n 1 (λ (n-1)
               (fact& n-1 (λ (n-1!)
                            (return (* n n-1!)))))))
```

Эта функция не требует стека для своей работы:

```
> (fact& 4 identity)
>(fact& 4 #<procedure:identity>)
>(fact& 3 #<procedure:...>)
>(fact& 2 #<procedure:...>)
>(fact& 1 #<procedure:...>)
>(fact& 0 #<procedure:...>)
<24
24
```

Отложенные вычисления хранятся в памяти, а не в стеке.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Эти преимущества особенно важны при разработке Web-приложений и серверов.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Эти преимущества особенно важны при разаработке Web-приложений и серверов.

Особенности Web-программирования

- HTTP – протокол без состояния.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Эти преимущества особенно важны при разаработке Web-приложений и серверов.

Особенности Web-программирования

- HTTP – протокол без состояния.
- Сервер не поддерживает все запущенные сессии.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Эти преимущества особенно важны при разаработке Web-приложений и серверов.

Особенности Web-программирования

- HTTP – протокол без состояния.
- Сервер не поддерживает все запущенные сессии.
- Web-приложение запускается на сервере, генерирует HTML, отправляет его клиенту и полностью завершает свою работу.

Программирование в CPS

Преимущества CPS

- Память существенно больше стека.
- Продолжения можно отделять от процесса.
- Можно сохранять множество продолжений и оперировать ими (стек обычно один).

Эти преимущества особенно важны при разаработке Web-приложений и серверов.

Особенности Web-программирования

- HTTP – протокол без состояния.
- Сервер не поддерживает все запущенные сессии.
- Web-приложение запускается на сервере, генерирует HTML, отправляет его клиенту и полностью завершает свою работу.
- Клиент (браузер) может во время сессии переходить на предыдущие страницы, дублировать их, завершать сессию на любой стадии или не завершать вовсе.

Модель Web-сервера

Stateful server

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read)))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))

(define (stop) (error "Application stopped."))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))

(define (stop) (error "Application stopped."))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))

(define (stop) (error "Application stopped."))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))

(define (stop) (error "Application stopped."))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
```

Модель Web-сервера

Stateful server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p)
  (read))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Input b 5
Web output: 9
```

Stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define (web-read p)
  (display p) (read) (stop))

(define (stop) (error "Application stopped."))

(define (start)
  (web-display
   (+ (web-read "Input a ")
      (web-read "Input b "))))
```

```
> (start)
Input a 4
Application stopped.
```

Модель Web-сервера

Continuation-based stateless server

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
> (resume)
Input the first number
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
> (resume)
Input the first number 4
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
> (resume)
Input the first number 4
Application stopped.
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
> (resume)
Input the first number 4
Application stopped.
> (resume)
Input the second number
```

Модель Web-сервера

Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

```
> (start/k)
Application stopped.
> (resume)
Input the first number 4
Application stopped.
> (resume)
Input the second number 5
```

Модель Web-сервера

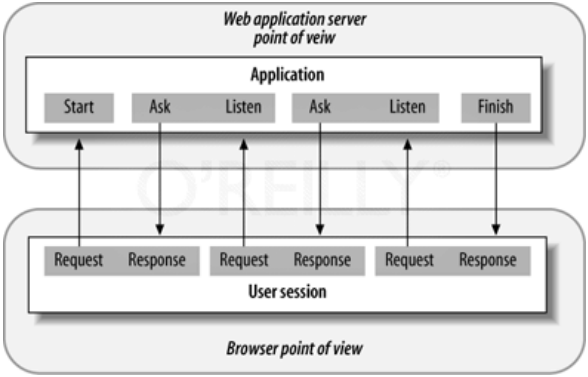
Continuation-based stateless server

```
(define (web-display n)
  (printf "Web output:  ~a ~n" n))

(define receiver (make-parameter #f))
(define prompt (make-parameter #f))

(define (web-read/k p k)
  (prompt p)
  (receiver k)
  (stop))

(define (resume)
  (display (prompt))
  ((receiver) (read)))
```

```
(define (start/k)
  (web-read/k
   "Input a "
   (λ (a) (web-read/k
           "Input b "
           (λ (b) (web-display (+ a b)))))))
```

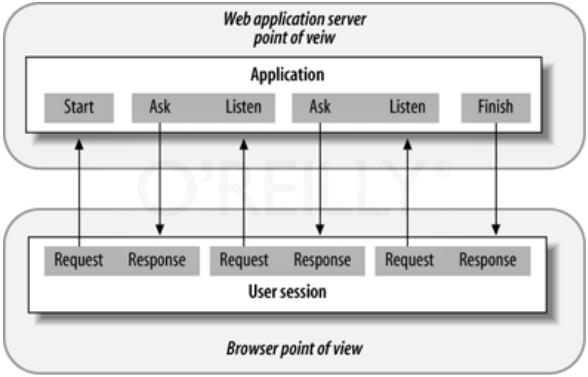
```
> (start/k)
Application stopped.
> (resume)
Input the first number 4
Application stopped.
> (resume)
Input the second number 5
Web output: 9
```

Общение с continuation-based сервером



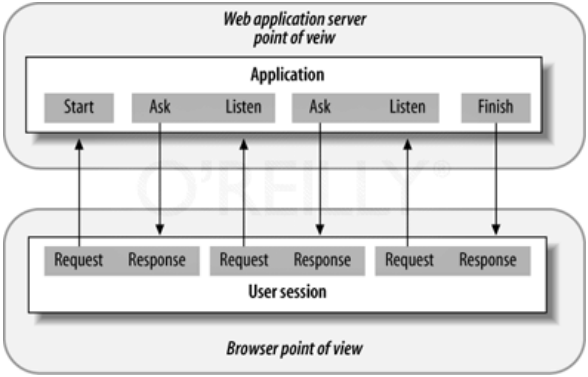
- Сервер по запросу клиента генерирует HTML-страницы и связывает поля action с индексами продолжений.

Общение с continuation-based сервером



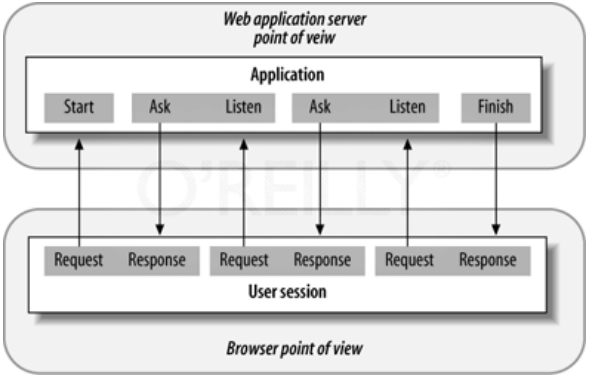
- Сервер по запросу клиента генерирует HTML-страницы и связывает поля action с индексами продолжений.
- Продолжения хранятся в хэш-таблице на сервере.

Общение с continuation-based сервером



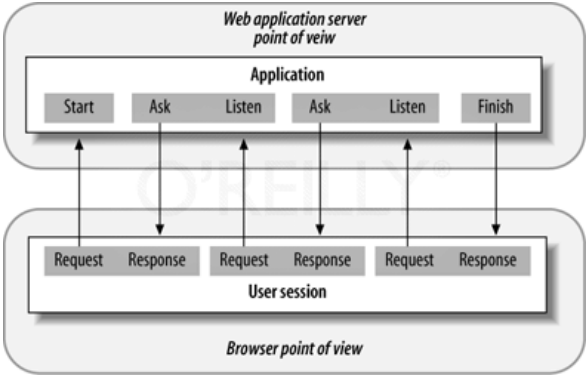
- Сервер по запросу клиента генерирует HTML-страницы и связывает поля action с индексами продолжений.
- Продолжения хранятся в хэш-таблице на сервере.
- Между запросами клиентов сервер не занят

Общение с continuation-based сервером



- Сервер по запросу клиента генерирует HTML-страницы и связывает поля action с индексами продолжений.
- Продолжения хранятся в хэш-таблице на сервере.
- Между запросами клиентов сервер не занят
- Клиент (браузер) может свободно перемещаться по истории HTML-страниц — каждая страница “знает” каким должно быть её продолжение.

Общение с continuation-based сервером



- Сервер по запросу клиента генерирует HTML-страницы и связывает поля action с индексами продолжений.
- Продолжения хранятся в хэш-таблице на сервере.
- Между запросами клиентов сервер не занят
- Клиент (браузер) может свободно перемещаться по истории HTML-страниц — каждая страница “знает” каким должно быть её продолжение.
- Не активные продолжения убираются из таблицы по мере заполнения памяти.

Модель Web-сервера

Создание независимых Web-приложений

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
Web output: 7
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
Web output: 7
> (f 'resume)
Input the second number
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
Web output: 7
> (f 'resume)
Input the second number 4
```

Модель Web-сервера

Создание независимых Web-приложений

```
(define (applet)
  (define receiver (make-parameter #f))
  (define prompt (make-parameter #f))
  (define (web-read/k p k)
    (prompt p)
    (receiver k)
    (stop))

  (/. 'resume --> (begin (display (prompt))
                         ((receiver) (read)))]
      'start --> (web-read/k
                  "Input the first number"
                  (λ (a) (web-read/k
                    "Input the second number"
                    (λ (b) (web-display (+ a b))))))))
```

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application stopped.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application stopped.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
Web output: 7
> (f 'resume)
Input the second number 4
Web output: 10
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
==> (+ 1 (* 3 6))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет

```
(λ (_) (+ 1 (* 3 _)))
```

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
==> (+ 1 (* 3 6))
```

```
(+ 1 (* 3 (call/cc (λ (k) (+ (k 2) (k 6))))))
```

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет `(λ (_) (+ 1 (* 3 _)))`

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
==> (+ 1 (* 3 6))
```

```
(+ 1 (* 3 (call/cc (λ (k) (+ (k 2) (k 6))))))
==> (+ 1 (* 3 2))
```

Основные применения:

- управление потоком (`break`, `continue`, `return`, `through/catch`, и т.д.)

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет `(λ (_) (+ 1 (* 3 _)))`

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
==> (+ 1 (* 3 6))
```

```
(+ 1 (* 3 (call/cc (λ (k) (+ (k 2) (k 6))))))
==> (+ 1 (* 3 2))
```

Основные применения:

- управление потоком (`break`, `continue`, `return`, `through/catch`, и т.д.)
- мягкая многопоточность,

Продолжение выражения

Продолжением называется контекст, в котором вычисляется выражение.

```
(+ 1 (* 3 (- 5 4))))
```

Для выражения `(- 5 4)` продолжением будет `(λ (_) (+ 1 (* 3 _)))`

Некоторые ЯП позволяют отделить продолжение от выражения.

call-with-current-continuation

Вызов в текущем продолжении (call/cc) позволяет фиксировать состояние программы и использовать его, как функцию.

```
(+ 1 (* 3 (call/cc (λ (k) (k (- 5 4))))))
==> (+ 1 (* 3 (- 5 4)))
```

В этом примере `k = (λ (x) (+ 1 (* 3 x)))`

Вызов `k` прерывает выполнение тела функции:

```
(+ 1 (* 3 (call/cc (λ (k) (+ 2 (k 6))))))
==> (+ 1 (* 3 6))
```

```
(+ 1 (* 3 (call/cc (λ (k) (+ (k 2) (k 6))))))
==> (+ 1 (* 3 2))
```

Основные применения:

- управление потоком (`break`, `continue`, `return`, `through/catch`, и т.д.)
- мягкая многопоточность,
- генераторы, поиск с возвратом, сопрограммы и т.д.

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))
```

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))

> (any odd? '(2 3 4 5))
3
```

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))
```

```
> (any odd? '(2 3 4 5))
3
```

```
(define (verbose f)
  (λ x
    (displayln (cons (object-name f) x))
    (apply f x)))
```

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))
```

```
> (any odd? '(2 3 4 5))
3
```

```
(define (verbose f)
  (λ x
    (displayln (cons (object-name f) x))
    (apply f x)))
```

```
> (any (verbose odd?) '(2 3 4 5))
(odd? 2)
(odd? 3)
(odd? 4)
(odd? 5)
3
```

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))

> (any odd? '(2 3 4 5))
3
```

```
(define (verbose f)
  (λ x
    (displayln (cons (object-name f) x))
    (apply f x)))

> (any (verbose odd?) '(2 3 4 5))
(odd? 2)
(odd? 3)
(odd? 4)
(odd? 5)
3
```

```
(define (any/c test lst)
  (call/cc
   (λ (break)
    (foldl (λ (el res)
             (and (zero? el) (break el))) #f lst))))
```

Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))

> (any odd? '(2 3 4 5))
3
```

```
(define (verbose f)
  (λ x
    (displayln (cons (object-name f) x))
    (apply f x)))

> (any (verbose odd?) '(2 3 4 5))
(odd? 2)
(odd? 3)
(odd? 4)
(odd? 5)
3
```

```
(define (any/c test lst)
  (call/cc
   (λ (break)
    (foldl (λ (el res)
             (and (zero? el) (break el))) #f lst))))

> (any/c odd? '(2 3 4 5))
3
```

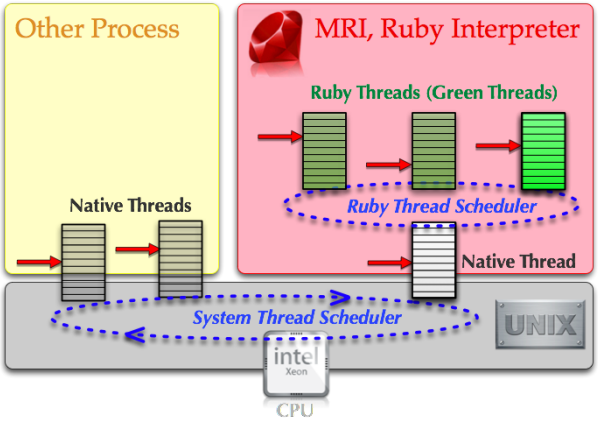
Продолжения, как объекты первого класса

```
(define (any test lst)
  (foldl (λ (el res)
           (and (zero? el) el)) #f lst))

> (any odd? '(2 3 4 5))
3
```

```
(define (verbose f)
  (λ x
    (displayln (cons (object-name f) x))
    (apply f x)))

> (any (verbose odd?) '(2 3 4 5))
(odd? 2)
(odd? 3)
(odd? 4)
(odd? 5)
3
```

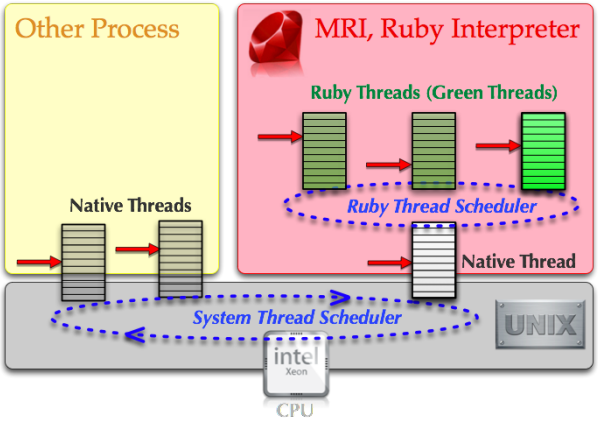
```
(define (any/c test lst)
  (call/cc
   (λ (break)
    (foldl (λ (el res)
             (and (zero? el) (break el))) #f lst))))

> (any/c odd? '(2 3 4 5))
3

> (any/c (verbose odd?) '(2 3 4 5))
(odd? 2)
(odd? 3)
3
```

Зелёные потоки



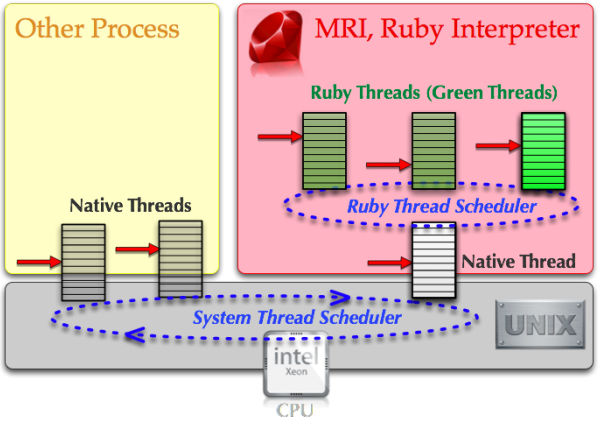
- Зелёные потоки – это потоки, управление которыми вместо операционной системы выполняет виртуальная машина.

Зелёные потоки



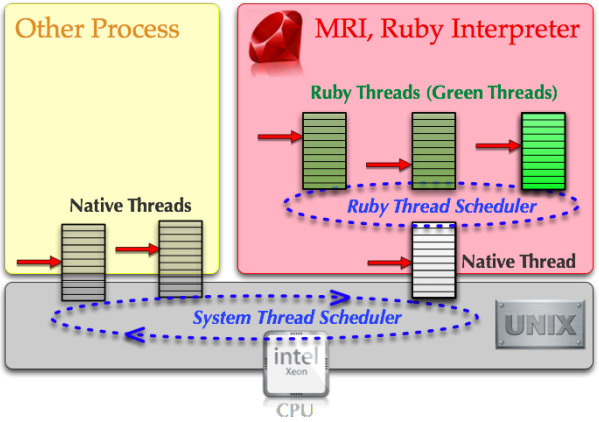
- Зелёные потоки – это потоки, управление которыми вместо операционной системы выполняет виртуальная машина.
- Позволяют создавать политику управления потоками, отличную от ОС.

Зелёные потоки



- Зелёные потоки – это потоки, управление которыми вместо операционной системы выполняет виртуальная машина.
- Позволяют создавать политику управления потоками, отличную от ОС.
- Не зависят от наличия нативной многопоточности.

Зелёные потоки



- Зелёные потоки – это потоки, управление которыми вместо операционной системы выполняет виртуальная машина.
- Позволяют создавать политику управления потоками, отличную от ОС.
- Не зависят от наличия нативной многопоточности.
- Быстро инициализируются.

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))

(define (pause)
  (call/cc (λ (k) (begin-fork (k (void))))))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции next и pause:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))

(define (pause)
  (call/cc (λ (k) (begin-fork (k (void))))))
```

Зададим последовательность простых циклических процессов:

```
> (begin
    (for ([i 5]) (display "h"))
    (for ([i 5]) (display "e"))
    (for ([i 5]) (display "y")))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))

(define (pause)
  (call/cc (λ (k) (begin-fork (k (void))))))
```

Зададим последовательность простых циклических процессов:

```
> (begin
    (for ([i 5]) (display "h"))
    (for ([i 5]) (display "e"))
    (for ([i 5]) (display "y")))
hhhhheeeeeyyyyy
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))

(define (pause)
  (call/cc (λ (k) (begin-fork (k (void))))))
```

Зададим последовательность простых циклических процессов:

```
> (begin
    (for ([i 5]) (display "h"))
    (for ([i 5]) (display "e"))
    (for ([i 5]) (display "y")))
hhhhheeeeeyyyyy
```

С помощью нашей системы многопоточности можно осуществить “параллельное” выполнение этих процессов:

```
> (begin-fork
    (for ([i 5]) (display "h") (pause))
    (for ([i 5]) (display "e") (pause))
    (for ([i 5]) (display "y") (pause)))
```

Простейшая реализация многопоточности

Потоки будем хранить в очереди:

```
(require data/queue)
(define Q (make-queue))
```

Для управления потоками определим форму **begin-fork**, а так же функции `next` и `pause`:

```
(define-syntax-rule (begin-fork proc ...)
  (begin (enqueue! Q (λ () proc)) ...
         (next)))

(define (next)
  (unless (queue-empty? Q) ((dequeue! Q))))

(define (pause)
  (call/cc (λ (k) (begin-fork (k (void))))))
```

Зададим последовательность простых циклических процессов:

```
> (begin
    (for ([i 5]) (display "h"))
    (for ([i 5]) (display "e"))
    (for ([i 5]) (display "y")))
hhhhheeeeeyyyyy
```

С помощью нашей системы многопоточности можно осуществить “параллельное” выполнение этих процессов:

```
> (begin-fork
    (for ([i 5]) (display "h") (pause))
    (for ([i 5]) (display "e") (pause))
    (for ([i 5]) (display "y") (pause)))
heyheyheyheyhey
```