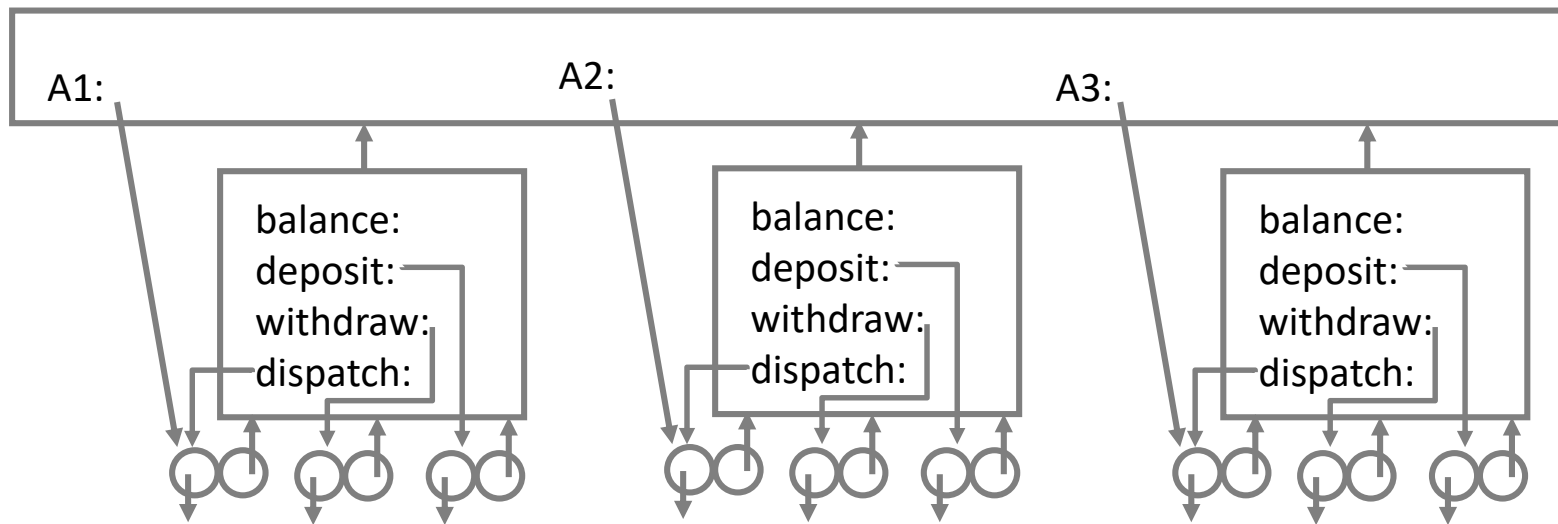


Лекция 8.

Потоки и задержанные вычисления

Потоки. Зачем?

- Предположим, что нужно промоделировать несколько счетов
- Можно использовать мутируемые счета-объекты



Потоки. Зачем? Счета-объекты

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        (error "Not enough money"))))
  (define (deposit amount) (begin
    (set! balance (+ balance amount)) balance))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Wrong message")))))
  dispatch)
```

Потоки. Зачем?

- Значения баланса счёта меняются после обработки вызовов `deposit` и `withdraw`, происходящих в какие-то моменты времени

A1.balance: 100
A1.balance: 130
A1.balance: 110
A1.balance: 70
A1.balance: 90
...

21-х A1.make-account 100

22-х A1.deposit 30

23-х A1.withdraw 20

24-х A1.withdraw 40

25-х A1.deposit 20

...

Потоки. Зачем?

- Что если вместо присваивания, хранить сами вызовы? Значение баланса счёта на любую нужно дату можно вычислить по префиксу лога вызовов нужной длины

21-x A1.make-account 100
22-x A1.deposit 30
23-x A1.withdraw 20
24-x A1.withdraw 40
25-x A1.deposit 20
...

A1.balance: 100

A1.balance: 130

A1.balance: 110

A1.balance: 70

A1.balance: 90

...

Ленивые вычисления

- *Нормальный порядок вычислений:*
 - вычисление нестрогой функции стартует до вычисления аргументов
 - подвыражение (аргумент) вычисляется только при *необходимости*
 - для вывода
 - для функции строгой по этому аргументу (элементарные функции обычно строгие)
 - для проверки (в `if`, `cond`, `case ...`)
 - для применения (1й элемент комбинации)
- *Мемоизация* – экономим вычисления за счёт запоминания вычисленных ранее результатов.
- Ещё бывает порядок, определяемый программистом: в описании функции указана её строгость по каждому аргументу.

Как узнать порядок вычислений?

Функция для визуализации порядка выполненных вычислений:

```
(define (notice x)
  (displayln x)
  x)
```

```
> (+ (notice 2) (notice 3)) ->    2
                                   3
                                   5
```

↑↑↑ подсмотрели порядок вычисления слагаемых

Поток как ленивый список

■ Ленивые аналоги cons, car, cdr (require racket/stream)

(stream-cons first rest) -- строит поток из головы и хвоста

(stream-first str) -- возвращает голову потока

(stream-rest str) -- возвращает хвост потока

■ Мы будем использовать потоки Racket, основная разница в названиях функций:

Racket

stream-cons

stream-first

stream-rest

stream

empty-stream

stream-empty? ...

Scheme (книга SICP)

cons-stream

stream-car

stream-cdr

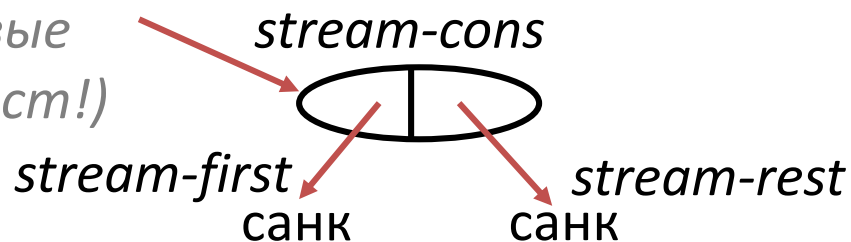
list

null

null? ...

Устройство потока

- Похож на пару, но обе части *ленивые*
(в учебнике SICP ленивый только хвост!)



- Санк (thunk) – обещание вычислить значение, когда оно будет *необходимо*.

- Санки мемоизированные (другие не рассматриваем).

■ Пример

```
> (define x (stream 99 (/ 1 0)))
```

```
> (stream-first x) -> 99
```

```
> (stream-first (stream-rest x)) -> /: division by zero
```

Визуализируем вычисление потока

■ Рассмотрим пример

```
(define s (stream (notice 1) (notice 2) (notice 3) (notice 4) (notice 5)  
  (notice 6) (notice 7) (notice 8) (notice 9) (notice 10)))
```

```
> (stream-ref s 5) -> 6
```

6

```
> (stream-ref s 7) -> 8
```

8

■ остальные элементы потока s лишь санки

```
> (stream-ref s 7) -> 8
```

```
> (stream-ref s 5) -> 6
```

■ мемоизация в действии! notice не был вычислен повторно!

Программирование с потоками

■ Простые описания в привычных терминах `foldl`, `map`, `filter`, `ormap`, `andmap` ... (т. е. `stream-fold`, `stream-map`, ...) и ленивые вычисления

■ Пример, со списком (не эффективно)

```
(list-ref (filter prime? (enumerate-interval 2 1000000000)) 100)
```

; где `(prime? x)` проверяет `x` на простоту

■ Пример с потоком (эффективнее, чем со списком)

```
(stream-ref (stream-filter prime? (in-range 2 1000000000 1)) 100)
```

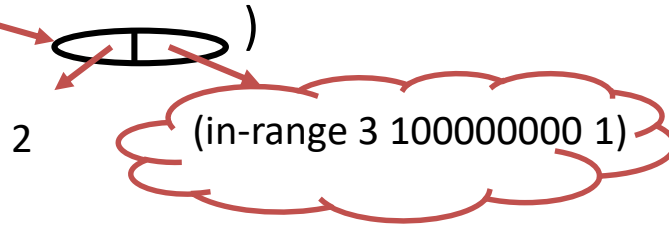
Аналог `in-range` можно реализовать самим:

```
(define (in-range a b step)
  (if (> a b) empty-stream
      (stream-cons a (in-range (+ a step) b step))))
```

Вычисления в примере

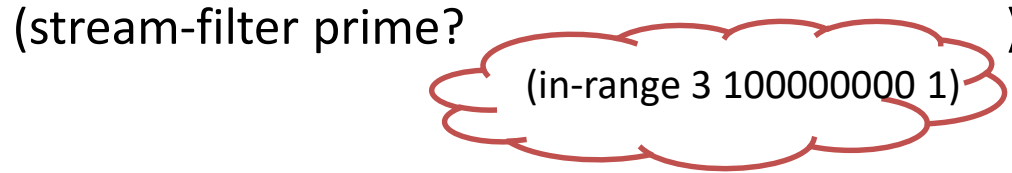
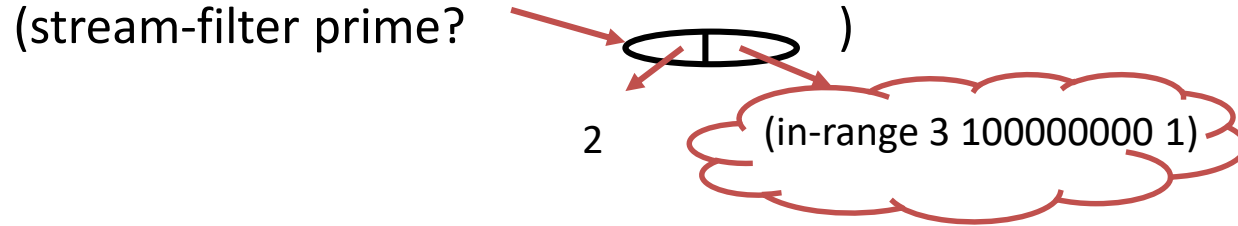
`(stream-ref (stream-filter prime? (in-range 2 1000000000 1)) 1)`

`(stream-filter prime?`



Вычисления в примере

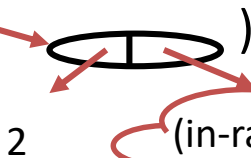
`(stream-ref (stream-filter prime? (in-range 2 1000000000 1)) 1)`



Вычисления в примере

`(stream-ref (stream-filter prime? (in-range 2 1000000000 1)) 1)`

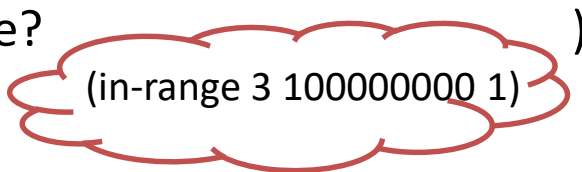
`(stream-filter prime?`



2

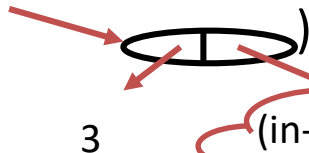
`(in-range 3 1000000000 1)`

`(stream-filter prime?`



`(in-range 3 1000000000 1)`

`(stream-filter prime?`



4

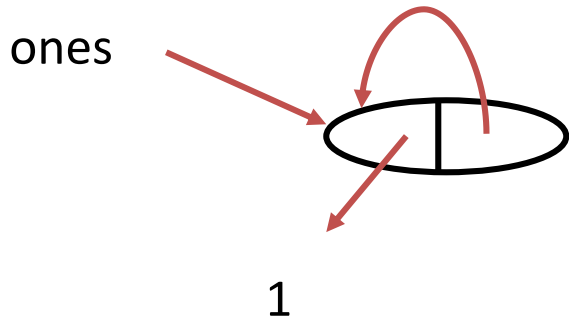
`(in-range 4 1000000000 1)`

Поток может быть бесконечным, и это нормально

рассмотрим такое определение

```
(define ones (stream-cons 1 ones))
```

```
> (stream-first (stream-rest ones)) -> 1
```



Бесконечный ряд единиц!

ones: 1 1 1 1 1 1

```
> (stream-ref ones 1) -> 1
```

```
> (stream-ref ones 1000) -> 1
```

```
> (stream-ref ones 10000000) -> 1
```

Один поток может быть описан через другой

```
(define (streams-add s1 s2)
  (cond ((stream-empty? s1) empty-stream)
        ((stream-empty? s2) empty-stream)
        (else (stream-cons
                 (+ (stream-first s1) (stream-first s2))
                 (streams-add (stream-rest s1) (stream-rest s2))))))

(define ints
  (stream-cons 1 (streams-add ones ints)))
```

неявное описание ints ones: 1 1 1 1 1 1 ...
ints: 1 2 3 ...

(+ (stream-first ones)
 (stream-first ints))

(streams-add (stream-rest ones)
 (stream-rest ints))

Явный способ задать ints

```
(define (ints-from n) ; порождающая функция  
  (stream-cons n (ints-from (+ n 1))))  
(define ints (ints-from 1))
```

Поток чисел Фибоначчи

(define fibs ; неявное описание

(stream-cons 0

(stream-cons 1

(streams-add (stream-rest fibs) fibs))))

(stream-rest fibs): 1 1 2 3 5 8 ...

fibs: 0 1 1 2 3 5 ...

fibs: 0 1 1 2 3 5 8 13 ...

или: (define fibs

(let fib-gen ((a 0) (b 1)) ; порождающая функция

(stream-cons a (fib-gen b (+ a b)))

))

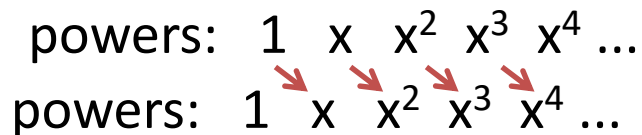
Ещё одна функция для неявного порождения

```
(define (stream-scale s f)
  (stream-map (lambda (x) (* x f)) s))
```

неявное описание потока степеней ($1 \ x \ x^2 \ x^3 \dots$)

```
(define (powers x)
  (stream-cons 1 (stream-scale (powers x) x)))
```

powers: $1 \ x \ x^2 \ x^3 \ x^4 \dots$
powers: $1 \ x \ x^2 \ x^3 \ x^4 \dots$



`(* (stream-first (powers x)) x)`

```
(define (powers x)
  (let pow-gen ((a 1) (b x)) ; порождающая функция
    (stream-cons a (pow-gen (* a b) b)))
  ))
```

Поток факториалов

```
(define (streams-mul s1 s2)
  (cond ((stream-empty? s1) empty-stream)
        ((stream-empty? s2) empty-stream)
        (else (stream-cons
                 (* (stream-first s1) (stream-first s2))
                 (streams-mul (stream-rest s1) (stream-rest s2))))))

(define n!s (stream-cons 1 (streams-mul n!s
                                         (stream-rest ints))))
```

ints: 1 2 3 4 ...
n!s: 1 → 2 → 6 → 24 ...

`(* (stream-first n!s)
 (stream-ref ints 1))`

`(streams-mul (stream-rest n!s)
 (stream-rest (stream-rest ints)))`

Явный способ задать n!

```
(define n!  
  (let n!s-gen ((a 1) (b 2)) ; порождающая функция  
    (stream-cons a (n!s-gen (* a b) (add1 b))))  
  ))
```

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли первые числа от 2 до 100

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли 2 в ответ и вычеркнули всё, что делится на 2

Поиск всех простых чисел

	2	3	5	7	9
11		13	15	17	19
21		23	25	27	29
31		33	35	37	39
41		43	45	47	49
51		53	55	57	59
61		63	65	67	69
71		73	75	77	79
81		83	85	87	89
91		93	95	97	99

взяли 3 в ответ и вычеркнули всё, что делится на 3

Поиск всех простых чисел

	2	3	5	7	
11		13		17	19
		23	25		29
31			35	37	
41		43		47	49
		53	55		59
61			65	67	
71		73		77	79
		83	85		89
91			95	97	

взяли 5 в ответ и вычеркнули всё, что делится на 5

Поиск всех простых чисел

	2	3	5	7	
11		13		17	19
		23			29
31				37	
41		43		47	49
		53			59
61				67	
71		73		77	79
		83			89
91				97	

взяли 7 в ответ и вычеркнули всё, что делится на 7

Поиск всех простых чисел

	2	3	5	7	
11		13		17	19
		23			29
31				37	
41		43		47	
		53			59
61				67	
71		73			79
		83			89
			97		

получили список всех простых чисел, меньших 100

Метод просеивания

```
(define (sieve str)
  (stream-cons
    (stream-first str)
    (sieve (stream-filter
      (lambda (x)
        (not (= 0 (remainder x (stream-first str))))))
      (stream-rest str)))))
```

```
(define primes
  (sieve (stream-rest ints)))
```

```
( 2 (sieve (stream-filter (lambda ... 2...) (stream-rest ints))) )
```

```
( 2 3 (sieve (stream-filter (lambda ... 3 ...)
```

```
(sieve (stream-filter (lambda ... 2 ...) (stream-rest ints))) ) )
```

Слияние бесконечных потоков

```
(define (interleave str1 str2)
  (stream-cons (stream-first str1)
    (interleave str2 (stream-rest str1))))

(interleave ones ints)
```

ones:	1	1	1	1	1	1	...					
ints:	1	2	3	4	5	...						
	<hr/>											
	1	1	1	2	1	3	1	4	1	5	1	...

Поток рациональных чисел

```
(define (div-by-stream n s)
  (stream-cons (/ n (stream-first s))
    (div-by-stream n (stream-rest s))))

(define (make-rats n)
  (stream-cons n
    (interleave (div-by-stream n (stream-rest ints))
      (make-rats (+ n 1)))))

(define rats (make-rats 1))
```

1/1	1/2	1/3	1/4	1/5	...
2/1	2/2	2/3	2/4	2/5	...
3/1	3/2	3/3	3/4	3/5	...
4/1	4/2	4/3	4/4	4/5	...

...

В итоге:

1/1 1/2 2/1 1/3 2/2 1/4 3/1 1/5 2/3 ... 30

Задержанные вычисления (delay и force)

Спец. форма (delay <выражение>) возвращает задержанный объект (promise), обещающий вычислить <выражение>. Функция (force <promise>) вынуждает вычисление и возвращает его результат. Чтобы ими пользоваться нужно подключить модуль (require racket/promise)

```
> (define th1 (delay (/ 1 0)))
```

```
> th1 -> #<promise:th1>
```

```
> (force th1) -> /: division by zero
```

```
> (define th2 (delay (notice (expt 2 20))))
```

```
> th2 -> #<promise:th2>
```

```
> (force th2) -> 1048576
```

1048576

```
> th2 -> #<promise:1048576>
```

```
> (force th2) -> 1048576
```

чудо мемоизации!!!

Задержанные объекты и санки

- Задержанный объект (promise) и санк (thunk) – не одно и то же.
- Явный вызов `force` – единственный способ вынудить вычисление задержанного объекта.
- Вычисление аргумента-санка в вызове нестрогой функции вынуждается само собой согласно нормальному порядку вычислений. Получается, что интерпретатор, реализующий нормальный порядок, сам вынуждает вычисления таких санков безо всяких `force`.
- Санки в составе потоков также вынуждаются при необходимости.
- Задержанные объекты мы рассматриваем как элемент ленивых вычислений, который легко реализовать своими силами.
- Строго говоря, *санк – это функция без аргументов, телом которой является отложенное вычисление*. Задержанные объекты мы реализуем на базе таких функций.

delay и force «на коленке»

вместо (delay <выражение>) будем писать код (lambda () <выражение>)

```
> (define th3 (lambda () (notice (expt 3 20))))
```

```
> th3 -> #<procedure:th3>    в роли задержанного объекта функция!
```

force -- просто вызов этой функции

```
> (define (my-force thunk) (thunk))
```

```
> (my-force th3) -> 3486784401
```

3486784401

```
> (my-force th3) -> 3486784401
```

3486784401

notice сработал дважды! наш задержанный объект не мемоизированный!

delay – спецформа, поэтому как функцию его не реализовать, но можно написать макрос.

Не мемоизированный delay «на коленке»

итак вместо (delay <выражение>) должно быть (lambda () <выражение>)

напишем макрос:

```
(define-syntax my-delay
```

```
  (syntax-rules ()
```

```
    ((_ expression)
```

```
      (lambda () expression)
```

```
  )))
```

```
> (define th3 (my-delay (notice (expt 3 20))))
```

```
> th3 -> #<procedure:th3>
```

```
> (my-force th3) -> 3486784401
```

```
3486784401
```

Мемоизированный delay «на коленке»

```
(define (memo-proc thunk)
  (let ((already-run? #f) (result #f))
    (lambda () (if (not already-run?)
                    (begin (set! result (thunk)) (set! already-run? #t) result)
                    result))))
```

теперь перепишем макрос

```
(define-syntax my-memo-delay
  (syntax-rules ()
    ((_ expression)
      (memo-proc (lambda () expression)))
  )))
```

Мемоизированный delay «на коленке»

проверяем

```
> (define th3m (my-memo-delay (notice (expt 3 20))))
```

```
> th3m -> #<procedure>
```

```
> (my-force th3m) -> 3486784401
```

3486784401

```
> (my-force th3m) -> 3486784401
```

теперь наш задержанный объект мемоизированный, и наши delay с force не отличаются от обычных!

Поисковое поведение на основе `amb`

Джон Маккарти придумал `amb` (ambiguous operator – «неоднозначный оператор»). У него произвольное количество аргументов. При вычислении выдаётся значение одного из аргументов. Особенность: можно вычислить повторно и результат будет другим. Для возврата, приводящего к повтору используют вызов `amb` без аргументов: `(amb)`

```
>(let* ((x (amb 1 2 3)) (y (amb 'a 'b))) (writeln (list x y)) (amb)) ->
```

(1 a)

(1 b)

(2 a)

(2 b)

(3 a)

(3 b)

no-more-choices

Поиск пифагоровых троек

Пифагорова тройка – $i\ j\ k$ (положительные целые $i \leq j \leq k$) $i^2 + j^2 = k^2$

```
(define (claim p) (when (not p) (amb)))  
((define (num-between low high)  
  (claim (<= low high))  
  (amb low (num-between (add1 low) high))))  
(define (pyth-3-between low high)  
  (let* ((i (num-between low high))  
        (j (num-between i high))  
        (k (num-between j high)))  
    (claim (= (+ (* i i) (* j j)) (* k k)))  
    (writeln i j k)  
    (amb))))
```

Поиск пифагоровых троек

запускаем

> (pyth-3-between 1 20) ->

(3 4 5)

(5 12 13)

(6 8 10)

(8 15 17)

(9 12 15)

(12 16 20)

no-more-choices

Реализация `amb` на коленке

(require racket/control)

(define failures null) ; список возвратов

(define (fail) ; обработка возврата

 (if (null? failures) (abort 'no-more-choices)

 (let ((failure (car failures))) (set! failures (cdr failures)) (failure))))

(define (amb/thunks choices) ; аргумент – список санков

 (call/cc (lambda (k) ; запоминаем точку

 (map (lambda (choice)

 (set! failures (cons (lambda () (call-in-continuation k choice)) failures))))

 (reverse choices)) ; записываем все альтернативы в возвраты

 (fail)))) ; вынуждаем возврат

; (call-in-continuation k choice) \approx (k (choice))

> (+ 1 (call/cc (lambda (k) (call-in-continuation k (lambda () 4))))) -> 5

Реализация amb

```
(define-syntax amb
  (syntax-rules ()
    ((amb) (fail))
    ((amb exp ...) (amb/thunks (list (lambda () exp) ...))))))
```

```
> (pyth-3-between 1 20) ->
```

```
(3 4 5)
```

```
(5 12 13)
```

```
(6 8 10)
```

```
(8 15 17)
```

```
(9 12 15)
```

```
(12 16 20)
```

```
no-more-choices
```

; ответы упорядочены по i

Другая реализация amb

; сделаем список возвратов очередью

(require data/queue)

(define failures (make-queue)) ; очередь возвратов

(define (fail2) ; обработка возврата из очереди

(if (queue-empty? failures)

(abort 'no-more-choices)

((dequeue! failures))))

(define (amb2/thunks choices) ; аргумент – список санков

(call/cc (lambda (k)

(map (lambda (choice)

(enqueue! failures2 (lambda () (call-in-continuation k choice))))

choices)

(fail2))))

Другая реализация amb

; заменим fail на fail2 и amb/thunks на amb2/thunks в макросе

```
(define-syntax amb
```

```
  (syntax-rules ()
```

```
    ((amb) (fail2))
```

```
    ((amb exp ...) (amb2/thunks (list (lambda () exp) ...))))))
```

> (pyth-3-between 1 20) ->

(3 4 5)

(6 8 10)

(5 12 13)

(9 12 15)

(8 15 17)

(12 16 20)

no-more-choices

; теперь ответы упорядочены по k

В чём разница между двумя `amb`

Можно встроить в `pyth-3-between` подсчёт проверок и искать 1ую тройку

```
(define tests 0)
```

```
(define (pyth-3-between low high)
```

```
  (let* ((i (num-between low high))
```

```
         (j (num-between i high))
```

```
         (k (num-between j high))))
```

```
(begin
```

```
  (set! tests (add1 tests))
```

```
  (claim (= (+ (* i i) (* j j)) (* k k))))
```

```
  (writeln (list i j k))
```

```
  ; (amb) по окончании могут остаться возвраты/альтернативы  
  )))
```

В чём разница между двумя `amb`

При поиске в ширину:

```
> (pyth-3-between 10 20) ->
```

```
(12 16 20)
```

```
> tests ->
```

```
246
```

При поиске в глубину:

```
> (pyth-3-between 10 20) ->
```

```
(12 16 20)
```

```
> tests ->
```

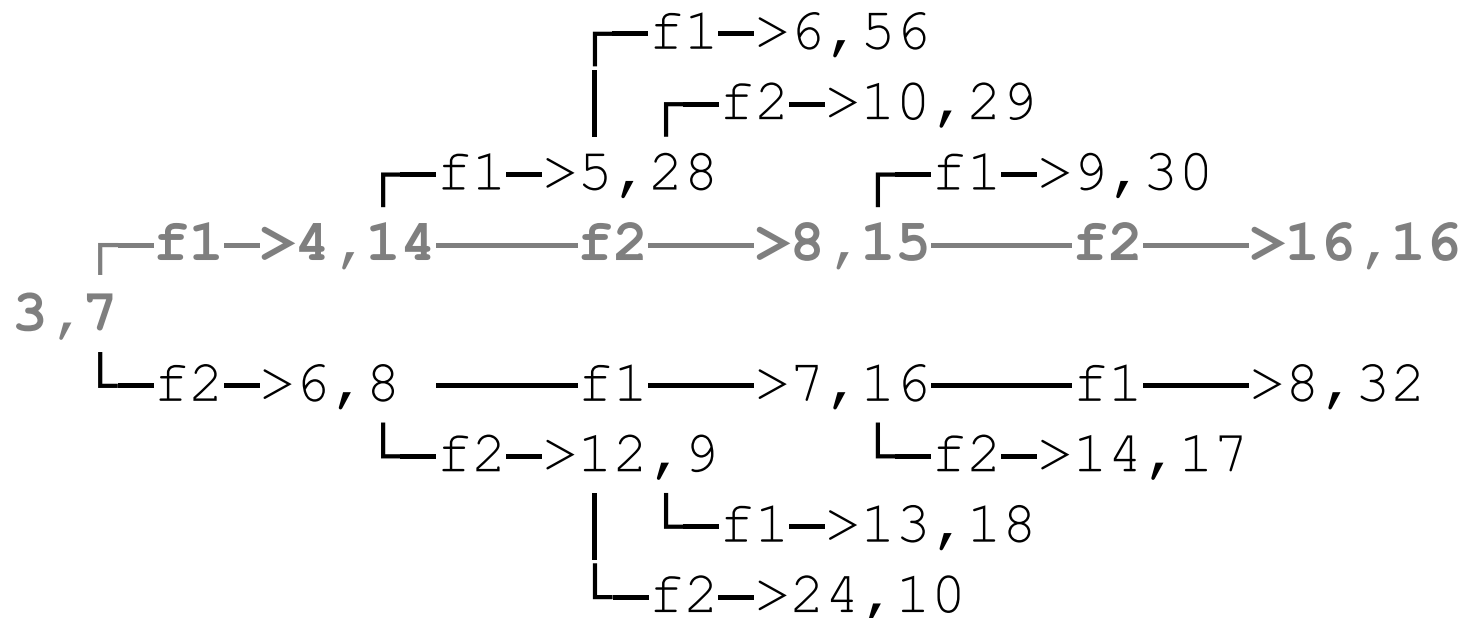
```
156
```

Можно анализировать задачу и выбирать подходящую для неё стратегию. Для каких-то задач поиск в глубину (без ограничения глубины) не годится. Недостаток поиска в ширину в том, что очередь обычно длиннее. Порядок выбора альтернатив также может иметь значение. Мы брали их по порядку. Можно выбирать их случайно. Можно пытаться делать так, чтобы более похожие на искомое решение альтернативы рассматривались в первую очередь. Можно размещать альтернативы вместе с априорными оценками и сортировать.

Ещё пример поисковой задачи. «Уравняй»

Есть пара чисел $A \leftrightarrow B$ и две операции $A, B \xrightarrow{f1} A+1, 2B$ и $A, B \xrightarrow{f2} 2A, B+1$

Найти последовательность применения операций к заданным A и B , уравнивающую числа.



«Уравняй». Решение с пределом глубины

```
(define (f1 lst) (list* (add1 (car lst)) (* 2 (cadr lst)) 1 (cddr lst)))  
(define (f2 lst) (list* (* 2 (car lst)) (add1 (cadr lst)) 2 (cddr lst)))  
(define (next-state s h)  
  (let ((s1 (f1 s))(s2 (f2 s)))  
    (claim (> h 0))  
    (amb s1 s2 (next-state s1 (sub1 h)) (next-state s2 (sub1 h)))))  
(define (search h start)  
  (let ((n (next-state start h)))  
    (claim (= (car n) (cadr n)))  
    (begin (write (list (car n) (cadr n))) (writeln (reverse (cddr n))))  
    (amb)  
    )))
```

«Уравняй». Запускаем

> (search 3 '(3 7)) ->

(16 16)(1 2 2)

no-more-choices

> (search 3 '(1 7)) ->

no-more-choices

; глубины не хватило

> (search 12 '(1 7)) ->

(20 20)(2 2 1 2 2)

(84 84)(2 2 2 1 1 2 1 2 2)

(140 140)(2 1 1 2 2 1 2 1 2 2)

(292 292)(1 2 2 1 2 2 2 2 1 2 1 1)

no-more-choices

12 – избыточная глубина. В дереве $2^{13}-1=8191$ вершин.

При поиске в глубину с $h=12$ сначала будет найдено длинное решение. 48

«Уравняй». Решение с итеративным углублением

```
(define (fail2) ; другая обработка возврата
  (if (queue-empty? failures) #f ((dequeue! failures))))
(define (next-state s h) (if (<= h 0) #f ; при исчерпании глубины #f
  (let ((s1 (f1 s))(s2 (f2 s)))
    (amb s1 s2 (next-state s1 (sub1 h)) (next-state s2 (sub1 h))))))
(define (search h start) (let ((n (next-state start h)))
  (cond ((not (pair? n)) #f) ; при исчерпании глубины #f
    ((= (car n) (cadr n))
     (begin (write (list (car n) (cadr n))) (writeln (reverse (cddr n))) (abort 'ok)
      ;(amb)
      ))
    ((queue-empty? failures2) #f) ; при исчерпании возвратов #f
    (else (amb)))))
```

«Уравняй». Решение с итеративным углублением

```
(define (iter-search start) (let loop ((h 0))
```

```
    (search h start)
```

```
    (loop (add1 h)))) ; бесконечный цикл с углублением
```

```
> (iter-search '(1 7)) ->
```

```
(20 20)(2 2 1 2 2)
```

```
ok
```

Итоги лекции

- Ленивые вычисления могут быть полезны. Ничто не даётся бесплатно, задержанные объекты и санки занимают память.
- Мемоизация может быть полезна при ленивых вычислениях.
- Программирование с потоками – оригинальный метод вычислений.
- На итоговой к/р будут задача на потоки. Решайте её с *порождающей функцией*. Если не выйдет, то решайте с *неявным описанием*.
- Бывает `stream-andmap`, `stream-ormap`, `stream-fold` – аналог `foldl`. `stream-fold` циклится на бесконечном потоке. `stream-andmap`, `stream-ormap` могут зациклиться на бесконечном потоке.
- Оператор Маккарти `amb` позволяет реализовать поиск.
- Поиски: в глубину / в ширину;
с ограниченной глубиной / с итеративным углублением