

Потоци

Трифон Трифонов

Функционално програмиране, 2023/24 г.

29 ноември – 6 декември 2023 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен 

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Дефиниция (Обещание)

Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата.

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Дефиниция (Обещание)

Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още *promise* и *отложена операция*.

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Дефиниция (Обещание)

Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още *promise* и *отложена операция*.

Изчислението на дадено обещание може да стане

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Дефиниция (Обещание)

Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още *promise* и *отложена операция*.

Изчислението на дадено обещание може да стане

- паралелно с изпълнението на основната програма (**асинхронно**)

Отложени операции

- Има случаи на тежки операции, които могат да отнемат много време за изпълнение
- Удобно е да имаме механизъм да **подготвяме** операциите и да ги **изпълняваме** само при нужда

Дефиниция (Обещание)

Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още *promise* и *отложена операция*.

Изчислението на дадено обещание може да стане

- паралелно с изпълнението на основната програма (**асинхронно**)
- при поискване от основната програма (**синхронно**)

Примитивни операции force и delay

- (delay <израз>)

Примитивни операции force и delay

- (delay <израз>)
- връща **обещание** за оценяването на <израз>

Примитивни операции force и delay

- (delay <израз>)
- връща **обещание** за оценяването на <израз>
- (force <обещание>)

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>`
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>`

Примитивни операции force и delay

- (delay <израз>)
- връща **обещание** за оценяването на <израз> (**специална форма**)
- (force <обещание>)
- форсира изчислението на <обещание> и връща оценката на <израз> (**примитивна функция**)

Примитивни операции force и delay

- (delay <израз>)
- връща **обещание** за оценяването на <израз> (**специална форма**)
- (force <обещание>)
- форсира изчислението на <обещание> и връща оценката на <израз> (**примитивна функция**)
- Примери:

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`
 - `(force error)` \longrightarrow **Грешка!**

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`
 - `(force error)` \longrightarrow **Грешка!**
 - `(define undefined (delay (+ a 3)))`

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`
 - `(force error)` \longrightarrow **Грешка!**
 - `(define undefined (delay (+ a 3)))`
 - `(define a 5)`

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`
 - `(force error)` \longrightarrow **Грешка!**
 - `(define undefined (delay (+ a 3)))`
 - `(define a 5)`
 - `(force undefined)` \longrightarrow ?

Примитивни операции `force` и `delay`

- `(delay <израз>)`
- връща **обещание** за оценяването на `<израз>` (**специална форма**)
- `(force <обещание>)`
- форсира изчислението на `<обещание>` и връща оценката на `<израз>` (**примитивна функция**)
- **Примери:**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(define error (delay (car '())))`
 - `(force error)` \longrightarrow **Грешка!**
 - `(define undefined (delay (+ a 3)))`
 - `(define a 5)`
 - `(force undefined)` \longrightarrow 8

All you need is λ — force и delay

- $(\text{delay } \langle \text{израз} \rangle) \iff (\text{lambda } () \langle \text{израз} \rangle)$
- $(\text{force } \langle \text{обещание} \rangle) \iff (\langle \text{обещание} \rangle)$

All you need is λ — force и delay

- $(\text{delay } \langle \text{израз} \rangle) \iff (\text{lambda } () \langle \text{израз} \rangle)$
- $(\text{force } \langle \text{обещание} \rangle) \iff (\langle \text{обещание} \rangle)$
- Не съвсем!

All you need is λ — force и delay

- `(delay <израз>)` \iff `(lambda () <израз>)`
- `(force <обещание>)` \iff `(<обещание>)`
- **Не съвсем!**
 - `(define bigpromise (delay (fact 30000)))`
 - `(force bigpromise)` \longrightarrow 2759537246...
 - `(force bigpromise)` \rightarrow 2759537246...

All you need is λ — force и delay

- $(\text{delay } \langle \text{израз} \rangle) \iff (\text{lambda } () \langle \text{израз} \rangle)$
- $(\text{force } \langle \text{обещание} \rangle) \iff (\langle \text{обещание} \rangle)$
- **Не съвсем!**
 - $(\text{define bigpromise } (\text{delay } (\text{fact } 30000)))$
 - $(\text{force bigpromise}) \longrightarrow 2759537246\dots$
 - $(\text{force bigpromise}) \rightarrow 2759537246\dots$
 - Обещанията в Scheme имат страничен ефект: “memoизират” вече изчислената стойност

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък $()$ или двойка $(h . t)$, където

- h — е произволен елемент (глава на потока)
- t — е **обещание** за поток (опашка на потока)

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък $()$ или двойка $(h . t)$, където

- h — е произволен елемент (глава на потока)
- t — е **обещание** за поток (опашка на потока)

В R^5RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък `()` или двойка `(h . t)`, където

- `h` — е произволен елемент (глава на потока)
- `t` — е **обещание** за поток (опашка на потока)

В R^5RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

- `(define the-empty-stream '())`

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък `()` или двойка `(h . t)`, където

- `h` — е произволен елемент (глава на потока)
- `t` — е **обещание** за поток (опашка на потока)

В R⁵RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

- `(define the-empty-stream '())`
- `(define (cons-stream h t) (cons h (delay t)))`

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък `()` или двойка `(h . t)`, където

- `h` — е произволен елемент (глава на потока)
- `t` — е **обещание** за поток (опашка на потока)

В R⁵RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

- `(define the-empty-stream '())`
- `(define (cons-stream h t) (cons h (delay t)))`
- `(define head car)`

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък `()` или двойка `(h . t)`, където

- `h` — е произволен елемент (глава на потока)
- `t` — е **обещание** за поток (опашка на потока)

В R⁵RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

- `(define the-empty-stream '())`
- `(define (cons-stream h t) (cons h (delay t)))`
- `(define head car)`
- `(define (tail s) (force (cdr s)))`

Потоци в Scheme

Дефиниция (Поток)

Списък, чиито елементи се изчисляват отложено.

По-точно: Поток е празен списък `()` или двойка `(h . t)`, където

- `h` — е произволен елемент (глава на потока)
- `t` — е **обещание** за поток (опашка на потока)

В R⁵RS няма вградени примитиви за работа с поток, но можем да си ги дефинираме.

- `(define the-empty-stream '())`
- `(define (cons-stream h t) (cons h (delay t)))`
- `(define head car)`
- `(define (tail s) (force (cdr s)))`
- `(define empty-stream? null?)`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) \longrightarrow 1`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`
- `(head (tail (tail s))) → 3`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`
- `(head (tail (tail s))) → 3`
- `(define s2 (cons-stream 3 (cons-stream b the-empty-stream))) → Грешка!`

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`
- `(head (tail (tail s))) → 3`
- `(define s2 (cons-stream 3 (cons-stream b the-empty-stream))) → Грешка!`
- **Защо?**

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`
- `(head (tail (tail s))) → 3`
- `(define s2 (cons-stream 3 (cons-stream b the-empty-stream))) → Грешка!`
- Защо?
- `cons-stream` трябва да оценява **само първия си аргумент!**

Примери

- `(define s (cons-stream 1 (cons-stream 2 (cons-stream 3 the-empty-stream))))`
- `(head s) → 1`
- `(tail s) → (2 . #<promise>)`
- `(head (tail s)) → 2`
- `(head (tail (tail s))) → 3`
- `(define s2 (cons-stream 3 (cons-stream b the-empty-stream))) → Грешка!`
- Защо?
- `cons-stream` трябва да оценява само първия си аргумент!
- `cons-stream` трябва да е **специална форма**

Дефиниране на специални форми

- `(define-syntax <СИМВОЛ>`
 `(syntax-rules () {(<шаблон> <тяло>))}))`

Дефиниране на специални форми

- `(define-syntax <символ>`
 `(syntax-rules () {(<шаблон> <тяло>)}))`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>

Дефиниране на специални форми

- `(define-syntax <символ>`
 `(syntax-rules () {(<шаблон> <тяло>)}))`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
- `define-syntax` има и други, по-сложни форми

Дефиниране на специални форми

- `(define-syntax <символ>`
 `(syntax-rules () {(<шаблон> <тяло>)}))`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
- `define-syntax` има и други, по-сложни форми
- Повечето специални форми на Scheme могат да се дефинират с `define-syntax` (за справка: R⁵RS)

Дефиниране на специални форми

- `(define-syntax <символ>`
 `(syntax-rules () {(<шаблон> <тяло>)}))`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
- `define-syntax` има и други, по-сложни форми
- Повечето специални форми на Scheme могат да се дефинират с `define-syntax` (за справка: R⁵RS)

Примери:

Дефиниране на специални форми

- `(define-syntax <символ>`
 `(syntax-rules () {(<шаблон> <тяло>)}))`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
- `define-syntax` има и други, по-сложни форми
- Повечето специални форми на Scheme могат да се дефинират с `define-syntax` (за справка: R⁵RS)

Примери:

```
(define-syntax delay  
  (syntax-rules () ((delay x) (lambda () x))))
```

Дефиниране на специални форми

- `(define-syntax <символ>`
`(syntax-rules () {(<шаблон> <тяло>))})`
- дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
- `define-syntax` има и други, по-сложни форми
- Повечето специални форми на Scheme могат да се дефинират с `define-syntax` (за справка: R⁵RS)

Примери:

```
(define-syntax delay
  (syntax-rules () ((delay x) (lambda () x))))
```

```
(define-syntax cons-stream
  (syntax-rules () ((cons-stream h t) (cons h (delay t)))))
```

Конструиране и деконструиране на потоци

Задача. Да се построи поток от целите числа в интервала $[a; b]$.

Конструирание и деконструирание на потоци

Задача. Да се построи поток от целите числа в интервала $[a; b]$.

Решение:

```
(define (enum a b)
  (if (> a b) the-empty-stream
      (cons-stream a (enum (+ a 1) b))))
```

Конструирание и деконструирание на потоци

Задача. Да се построи поток от целите числа в интервала $[a; b]$.

Решение:

```
(define (enum a b)
  (if (> a b) the-empty-stream
      (cons-stream a (enum (+ a 1) b))))
```

Задача. Да се намерят първите n елемента на даден поток.

Конструирание и деконструирание на потоци

Задача. Да се построи поток от целите числа в интервала $[a; b]$.

Решение:

```
(define (enum a b)
  (if (> a b) the-empty-stream
      (cons-stream a (enum (+ a 1) b))))
```

Задача. Да се намерят първите n елемента на даден поток.

Решение:

```
(define (take n s)
  (if (or (empty-stream? s) (= n 0)) '()
      (cons (head s) (take (- n 1) (tail s)))))
```

Приложение на потоци

Задача. Да се намери първата позиция в поток, на която има елемент с дадено свойство.

Приложение на потоци

Задача. Да се намери първата позиция в поток, на която има елемент с дадено свойство.

Решение.

```
(define (find-stream p? s)
  (cond ((empty-stream? s) #f)
        ((p? (head s)) s)
        (else (find-stream p? (tail s)))))
```


Приложение на потоци

Задача. Да се намери първата позиция в поток, на която има елемент с дадено свойство.

Решение.

```
(define (find-stream p? s)
  (cond ((empty-stream? s) #f)
        ((p? (head s)) s)
        (else (find-stream p? (tail s)))))
```

Задача. Да се намери второто по големина просто число след 10000 със сума на цифрите кратна на 5.

Приложение на потоци

Задача. Да се намери първата позиция в поток, на която има елемент с дадено свойство.

Решение.

```
(define (find-stream p? s)
  (cond ((empty-stream? s) #f)
        ((p? (head s)) s)
        (else (find-stream p? (tail s)))))
```

Задача. Да се намери второто по големина просто число след 10000 със сума на цифрите кратна на 5.

Решение.

```
(define (prime-5? n) (and (prime? n)
                          (= (remainder (sum-digits n) 5) 0)))

(define second-prime-5
  (head (find-stream prime-5?
                    (tail (find-stream prime-5?
                                      (enum 10000 100000))))))
```

Безкрайни потоци

Отлагането на операции позволява създаването на **безкрайни потоци**!

Безкрайни потоци

Отлагането на операции позволява създаването на **безкрайни потоци**!

Примери:

```
(define (from n) (cons-stream n (from (+ n 1))))  
(define nats (from 0))
```

Безкрайни потоци

Отлагането на операции позволява създаването на **безкрайни потоци**!

Примери:

```
(define (from n) (cons-stream n (from (+ n 1))))  
(define nats (from 0))
```

Задача. Да се генерира потокът от числата на Фибоначи.

Безкрайни потоци

Отлагането на операции позволява създаването на **безкрайни потоци**!

Примери:

```
(define (from n) (cons-stream n (from (+ n 1))))  
(define nats (from 0))
```

Задача. Да се генерира потокът от числата на Фибоначи.

Решение:

```
(define (generate-fibs a b)  
  (cons-stream a (generate-fibs b (+ a b))))  
(define fibs (generate-fibs 0 1))
```

Безкрайни потоци

Отлагането на операции позволява създаването на **безкрайни потоци**!

Примери:

```
(define (from n) (cons-stream n (from (+ n 1))))  
(define nats (from 0))
```

Задача. Да се генерира потокът от числата на Фибоначи.

Решение:

```
(define (generate-fibs a b)  
  (cons-stream a (generate-fibs b (+ a b))))  
(define fibs (generate-fibs 0 1))
```

- Функциите `from` и `generate-fibs` наричаме **генератори**
- Казваме, че потоците `nats` и `fibs` са **индиректно дефинирани**

Функции от по-висок ред за потоци

Трансформиране (map)

```
(define (map-stream f s) (cons-stream (f (head s))  
                                       (map-stream f (tail s))))
```


Функции от по-висок ред за потоци

Трансформиране (map)

```
(define (map-stream f s) (cons-stream (f (head s))  
                                       (map-stream f (tail s))))
```

Филтриране (filter)

```
(define (filter-stream p? s)  
  (if (p? (head s))  
      (cons-stream (head s) (filter-stream p? (tail s)))  
      (filter-stream p? (tail s))))
```

Функции от по-висок ред за потоци

Трансформиране (map)

```
(define (map-stream f s) (cons-stream (f (head s))  
                                       (map-stream f (tail s))))
```

Филтриране (filter)

```
(define (filter-stream p? s)  
  (if (p? (head s))  
      (cons-stream (head s) (filter-stream p? (tail s)))  
      (filter-stream p? (tail s))))
```

Комбиниране (zip)

```
(define (zip-streams op s1 s2)  
  (cons-stream (op (head s1) (head s2))  
               (zip-streams op (tail s1) (tail s2))))
```

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

Можем да реализираме `map-stream` аналогично да комбинира произволен брой потоци:

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

Можем да реализираме `map-stream` аналогично да комбинира произволен брой потоци:

```
(define (map-stream f . streams)
  (cons-stream (apply f (map head streams))
    (apply map-stream f (map tail streams)))))
```

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

Можем да реализираме `map-stream` аналогично да комбинира произволен брой потоци:

```
(define (map-stream f . streams)
  (cons-stream (apply f (map head streams))
               (apply map-stream f (map tail streams)))))
```

Примери:

- `(map-stream + nats ones) → 1 2 3 4 ...`

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

Можем да реализираме `map-stream` аналогично да комбинира произволен брой потоци:

```
(define (map-stream f . streams)
  (cons-stream (apply f (map head streams))
    (apply map-stream f (map tail streams))))
```

Примери:

- `(map-stream + nats ones) → 1 2 3 4 ...`
 - еквивалентно на `zip-streams`

map-stream с произволен брой параметри

Стандартната функция `map` за списъци позволява комбиниране на произволен брой n списъци с n -аргументна функция.

Можем да реализираме `map-stream` аналогично да комбинира произволен брой потоци:

```
(define (map-stream f . streams)
  (cons-stream (apply f (map head streams))
    (apply map-stream f (map tail streams))))
```

Примери:

- `(map-stream + nats ones)` \longrightarrow 1 2 3 4 ...
 - еквивалентно на `zip-streams`
- `(map-stream list nats (map-stream + nats nats fibs))` \longrightarrow
(0 0) (1 3) (2 5) (3 8) ...

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия**!

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия**!

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

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия!**

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

Построяване на nats:

$$\begin{array}{rcccccccc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
 + & & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\
 \hline
 & & 1 & 2 & 3 & 4 & 5 & 6 & \dots
 \end{array}$$

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия!**

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

Построяване на nats:

$$\begin{array}{rcccccccc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
 + & & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\
 \hline
 & & 1 & 2 & 3 & 4 & 5 & 6 & \dots
 \end{array}$$

```
(define nats (cons-stream 0 (map-stream 1+ nats)))
```

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия!**

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

Построяване на nats:

$$\begin{array}{rcccccccc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
 + & & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\
 \hline
 & & 1 & 2 & 3 & 4 & 5 & 6 & \dots
 \end{array}$$

```
(define nats (cons-stream 0 (map-stream + ones nats)))
```

Директна дефиниция на потоци

Можем да дефиниране на потоци с **директна рекурсия!**

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

Построяване на nats:

$$\begin{array}{rcccccccc}
 & + & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
 & & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\
 \hline
 & & 1 & 2 & 3 & 4 & 5 & 6 & \dots
 \end{array}$$

```
(define nats (cons-stream 0 (map-stream + ones nats)))
```

Построяване на fibs:

$$\begin{array}{rcccccccc}
 & + & 0 & 1 & 1 & 2 & 3 & 5 & \dots \\
 & & 1 & 1 & 2 & 3 & 5 & 8 & \dots \\
 \hline
 & & 1 & 2 & 3 & 5 & 8 & 13 & \dots
 \end{array}$$

Директна дефиниция на потоци

Можем да дефинираме на потоци с **директна рекурсия**!

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

Построяване на nats:

$$\begin{array}{rcccccccc}
 & + & 0 & 1 & 2 & 3 & 4 & 5 & \dots \\
 & & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\
 \hline
 & & 1 & 2 & 3 & 4 & 5 & 6 & \dots
 \end{array}$$

```
(define nats (cons-stream 0 (map-stream + ones nats)))
```

Построяване на fibs:

$$\begin{array}{rcccccccc}
 & + & 0 & 1 & 1 & 2 & 3 & 5 & \dots \\
 & & 1 & 1 & 2 & 3 & 5 & 8 & \dots \\
 \hline
 & & 1 & 2 & 3 & 5 & 8 & 13 & \dots
 \end{array}$$

```
(define fibs (cons-stream 0
  (cons-stream 1
    (map-stream + fibs (tail fibs)))))
```

Решето на Ератостен

Решето на Ератостен

Алгоритъм за намиране на прости числа

Решето на Ератостен

Алгоритъм за намиране на прости числа

- Започваме със списък от последователни цели числа
- Докато не стигнем до края на списъка, повтаряме:
 - Намираме следващото незадраскано число p , то е просто
 - Задраскваме всички следващи числа, които се делят на p

Решето на Ератостен

Алгоритъм за намиране на прости числа

- Започваме със списък от последователни цели числа
- Докато не стигнем до края на списъка, повтаряме:
 - Намираме следващото незадраскано число p , то е просто
 - Задраскваме всички следващи числа, които се делят на p

Ще реализираме решето над потенциално безкраен поток от числа:

Решето на Ератостен

Алгоритъм за намиране на прости числа

- Започваме със списък от последователни цели числа
- Докато не стигнем до края на списъка, повтаряме:
 - Намираме следващото незадраскано число p , то е просто
 - Задраскваме всички следващи числа, които се делят на p

Ще реализираме решето над потенциално безкраен поток от числа:

```
(define (nondivisor d) (lambda (n) (> (remainder n d) 0)))
```

Решето на Ератостен

Алгоритъм за намиране на прости числа

- Започваме със списък от последователни цели числа
- Докато не стигнем до края на списъка, повтаряме:
 - Намираме следващото незадраскано число p , то е просто
 - Задраскваме всички следващи числа, които се делят на p

Ще реализираме решето над потенциално безкраен поток от числа:

```
(define (nondivisor d) (lambda (n) (> (remainder n d) 0)))
```

```
(define (sieve stream)
  (cons-stream (head stream)
    (sieve (filter-stream (nondivisor (head stream)) (tail stream)))))
```

Решето на Ератостен

Алгоритъм за намиране на прости числа

- Започваме със списък от последователни цели числа
- Докато не стигнем до края на списъка, повтаряме:
 - Намираме следващото незадраскано число p , то е просто
 - Задраскваме всички следващи числа, които се делят на p

Ще реализираме решето над потенциално безкраен поток от числа:

```
(define (nondivisor d) (lambda (n) (> (remainder n d) 0)))
```

```
(define (sieve stream)
  (cons-stream (head stream)
    (sieve (filter-stream (nondivisor (head stream)) (tail stream)))))
```

```
(define primes (sieve (from 2)))
```