Лекция 7

Ленивые вычисления

Функциональное и логическое программирование

КамчатГТУ, 2013 г.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:
  - **по значению** — вызывающая функция копирует в память, непосредственное значение фактического аргумента.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:
  - **по значению** — вызывающая функция копирует в память, непосредственное значение фактического аргумента.
  - **по ссылке** — вызывающая функция получает адрес фактического аргумента.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:
  - **по значению** — вызывающая функция копирует в память, непосредственное значение фактического аргумента.
  - **по ссылке** — вызывающая функция получает адрес фактического аргумента.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:
  - **по значению** — вызывающая функция копирует в память, непосредственное значение фактического аргумента.
  - **по ссылке** — вызывающая функция получает адрес фактического аргумента.

В чистых функциональных языках вызовы по ссылке и по значеннию функционально эквивалентны.

Строгая стратегия вычислений

Все аргументы функции всегда вычисляются до вызова функции.

- **Аппликативный порядок вычислений** — аргументы функции вычисляются рекурсивно. При этом используется стратегия **обратного обхода** дерева вычислений в глубину (изнутри наружу).
- Передача аргументов:
  - **по значению** — вызывающая функция копирует в память, непосредственное значение фактического аргумента.
  - **по ссылке** — вызывающая функция получает адрес фактического аргумента.

В чистых функциональных языках вызовы по ссылке и по значеннию функционально эквивалентны.

- C
- Pascal
- Lisp, Scheme
- ML, OCaml
- Closure
- F#

Строгая стратегия вычислений

- Достоинства:

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,

if test? then $proc_1$ else $proc_2$

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,

if test? then $proc_1$ else $proc_2$

```
if 1 > 0 then 1 else 1/0
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,

`if test? then` $proc_1$ `else` $proc_2$

```
if 1 > 0 then 1 else 1/0
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0

first (filter test? (make-big-list))
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,
  - необходимость явной конечности определяемых данных,

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0
```

```
first (filter test? (make-big-list))
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,
  - необходимость явной конечности определяемых данных,
  - невозможность реализации семантики минимальных вычислений без использования примитивов языка.

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0

first (filter test? (make-big-list))
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,
  - необходимость явной конечности определяемых данных,
  - невозможность реализации семантики минимальных вычислений без использования примитивов языка.

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0

first (filter test? (make-big-list))

or (long-procedure1) (long-procedure2)
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,
  - необходимость явной конечности определяемых данных,
  - невозможность реализации семантики минимальных вычислений без использования примитивов языка.

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0

first (filter test? (make-big-list))

or (long-procedure1) (long-procedure2)

0 * (1 * (2 * (3 * (4 * (5 * 6)))))
```

Строгая стратегия вычислений

- Достоинства:
  - простота реализации,
  - экономия памяти,
  - простота оптимизации хвостовой рекурсии.
- Недостатки:
  - необходимость вводить управляющие конструкциии на уровне примитивов языка,
  - интенсивность вычислений,
  - необходимость явной конечности определяемых данных,
  - невозможность реализации семантики минимальных вычислений без использования примитивов языка.

```
if test? then proc1 else proc2
if 1 > 0 then 1 else 1/0

first (filter test? (make-big-list))

or (long-procedure1) (long-procedure2)

0 * (1 * (2 * (3 * (4 * (5 * 6)))))
0 * (1 / 0)
```

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:
  - по имени — аргументы функции подставляются в тело функции и вычисляются вместе с телом функции.

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:
  - по имени — аргументы функции подставляются в тело функции и вычисляются вместе с телом функции.
  - по необходимости (ленивая) — передача по имени с мемоизацией.

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:
  - по имени — аргументы функции подставляются в тело функции и вычисляются вместе с телом функции.
  - по необходимости (ленивая) — передача по имени с мемоизацией.

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:
  - по имени — аргументы функции подставляются в тело функции и вычисляются вместе с телом функции.
  - по необходимости (ленивая) — передача по имени с мемоизацией.

В чистых функциональных языках вызовы по имени и по необходимости функционально эквивалентны.

Нестрогая стратегия вычислений

Аргументы функции не вычисляются до тех пор, пока это не потребуется в теле функции.

- Нормальный порядок вычислений — используется стратегия прямого обхода дерева вычислений в глубину (снаружи внутрь).
- Передача аргументов:
  - по имени — аргументы функции подставляются в тело функции и вычисляются вместе с телом функции.
  - по необходимости (ленивая) — передача по имени с мемоизацией.

В чистых функциональных языках вызовы по имени и по необходимости функционально эквивалентны.

- Hope
- Miranda
- Haskell

Нестрогая стратегия вычислений

- Достоинства:

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,
  - возможность строить управляющие конструкции с помощью абстракции.

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,
  - возможность строить управляющие конструкции с помощью абстракции.
- Недостатки:

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,
  - возможность строить управляющие конструкции с помощью абстракции.
- Недостатки:
  - сложность реализации,

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,
  - возможность строить управляющие конструкции с помощью абстракции.
- Недостатки:
  - сложность реализации,
  - сложность использования исключений, операции ввода-вывода, последовательных действий.

Нестрогая стратегия вычислений

- Достоинства:
  - возможность оптимизации засчёт исключения ненужных вычислений,
  - повышение устойчивости программ засчет избегания исключительных ситуаций,
  - возможность оперирования семантически бесконечными данными,
  - возможность строить управляющие конструкции с помощью абстракции.
- Недостатки:
  - сложность реализации,
  - сложность использования исключений, операции ввода-вывода, последовательных действий.
  - сложность автоматической оптимизации хвостовой рекурсии.

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

```
length [1/0 log 0 tan π/2 World.Destroy]
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge$ `(sin x / x)` $\vee 1$

  `sinc` $0 = 0 \neq 0 \wedge$ `(sin x / x)` $\vee 1$

  `sinc` $0 =$ `false` $\wedge$ `_` $\vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

```
length [1/0 log 0 tan π/2 World.Destroy]
1 + length [log 0 tan π/2 World.Destroy]
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

`length` $[1/0\ \log 0\ \tan \pi/2$ `World.Destroy]`

`1 + length` $[\log 0\ \tan \pi/2$ `World.Destroy]`

`1 + 1 + length` $[\tan \pi/2$ `World.Destroy]`

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge$ `(sin` x `/` x`)` $\vee 1$

  `sinc` $0 = 0 \neq 0 \wedge$ `(sin` x `/` x`)` $\vee 1$

  `sinc` $0 =$ `false` $\wedge$ `_` $\vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

```
length [1/0 log 0 tan π/2 World.Destroy]
1 + length [log 0 tan π/2 World.Destroy]
1 + 1 + length [tan π/2 World.Destroy]
1 + 1 + 1 + length [World.Destroy]
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge$ `(sin` x `/` x`)` $\vee 1$

  `sinc` $0 = 0 \neq 0 \wedge$ `(sin` x `/` x`)` $\vee 1$

  `sinc` $0 =$ `false` $\wedge$ `_` $\vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

```
length [1/0 log 0 tan π/2 World.Destroy]
1 + length [log 0 tan π/2 World.Destroy]
1 + 1 + length [tan π/2 World.Destroy]
1 + 1 + 1 + length [World.Destroy]
1 + 1 + 1 + 1 + length []
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

```
length [1/0 log 0 tan π/2 World.Destroy]
1 + length [log 0 tan π/2 World.Destroy]
1 + 1 + length [tan π/2 World.Destroy]
1 + 1 + 1 + length [World.Destroy]
1 + 1 + 1 + 1 + length []
1 + 1 + 1 + 1 + 0
```

Минимальные вычисления

Нормальный порядок вычислений — стратегия прямого обхода дерева вычислений в глубину.

- `0 * (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) (1 * (2 * (3 * (4 * (5 * 6)))))`

  `(0 *) _ = 0`

  `0`

- `sinc` $x = x \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 = 0 \neq 0 \wedge (\sin\ x\ /\ x) \vee 1$

  `sinc` $0 =$ `false` $\wedge\ \_ \vee 1$

  `sinc` $0 =$ `false` $\vee\ 1$

  `sinc` $0 = 1$

  `1`

```
length [] = 0
length h:t = 1 + length t
```

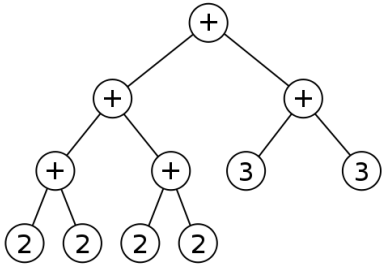
```
length [1/0 log 0 tan π/2 World.Destroy]
1 + length [log 0 tan π/2 World.Destroy]
1 + 1 + length [tan π/2 World.Destroy]
1 + 1 + 1 + length [World.Destroy]
1 + 1 + 1 + 1 + length []
1 + 1 + 1 + 1 + 0
4
```

Редукция дерева вычислений

$$((2 + 2) + (2 + 2)) + (3 + 3)$$

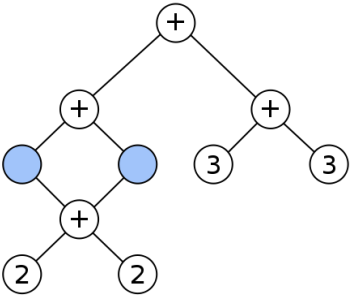
Редукция дерева вычислений

$$((2+2)+(2+2))+(3+3)$$



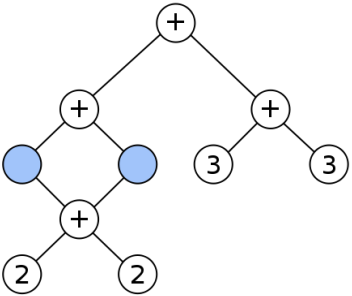
Редукция дерева вычислений

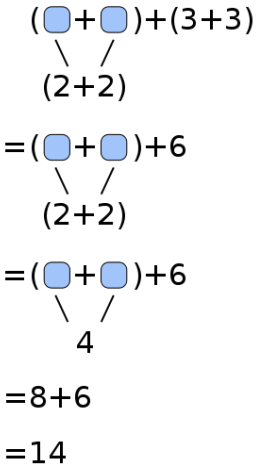
$$((2+2)+(2+2))+(3+3)$$



Редукция дерева вычислений

$$((2+2)+(2+2))+(3+3)$$





Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                   then h : filter p t
                   else filter p t
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                   then h : filter p t
                   else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                    then h : filter p t
                    else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
#f ∨ #f ∨ (foldr ∨ ∘ (> 2) [3 3 2 4 1 6 4])
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
#f ∨ #f ∨ (foldr ∨ ∘ (> 2) [3 3 2 4 1 6 4])
#f ∨ #f ∨ #t ∨
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
#f ∨ #f ∨ (foldr ∨ ∘ (> 2) [3 3 2 4 1 6 4])
#f ∨ #f ∨ #t ∨
#f ∨ #f ∨ #t
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                 then h : filter p t
                 else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
#f ∨ #f ∨ (foldr ∨ ∘ (> 2) [3 3 2 4 1 6 4])
#f ∨ #f ∨ #t ∨
#f ∨ #f ∨ #t
#f ∨ #t
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Обработка ленивых списков

```
first (filter even? [1 2 3 3 2 4 1 6 4])
first (filter even? [2 3 3 2 4 1 6 4])
first 2 : (filter even? [3 3 2 4 1 6 4])
2
```

```
first [] = #f
first h : t = h

filter p [] = []
filter p h : t = if (p h)
                    then h : filter p t
                    else filter p t
```

```
any? (> 2) [1 2 3 3 2 4 1 6 4]
#f ∨ (foldr ∨ ∘ (> 2) [2 3 3 2 4 1 6 4])
#f ∨ #f ∨ (foldr ∨ ∘ (> 2) [3 3 2 4 1 6 4])
#f ∨ #f ∨ #t ∨
#f ∨ #f ∨ #t
#f ∨ #t
#t
```

```
any? p = foldr ∨ ∘ p false

foldr f x0 [] = x0
foldr f x0 h : t = f h (foldr f x0 t)
```

Потоки, как ленивые списки

```
ones = 1 : ones
```

Потоки, как ленивые списки

```
ones = 1 : ones
head ones = 1
```

Потоки, как ленивые списки

```
ones = 1 : ones
```

```
head ones = 1
tail ones = ones = 1 : ones
```

Потоки, как ленивые списки

```
ones = 1 : ones
```

```
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Потоки, как ленивые списки

```
ones = 1 : ones
```

```
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Потоки, как ленивые списки

```
ones = 1 : ones
```

```
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Определение

Поток – последовательнось элементов, вызываемых (создаваемых) по требованию.

Потоки, как ленивые списки

```
ones = 1 : ones
```

```
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Определение

Поток – последовательнось элементов, вызываемых (создаваемых) по требованию.

Последовательность целых чисел больших заданного числа:

```
enum n = n : (enum n + 1)
enum n = {n  n + 1  n + 2  n + 3  ...}
```

Потоки, как ленивые списки

```
ones = 1 : ones
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Определение

Поток – последовательнось элементов, вызываемых (создаваемых) по требованию.

Последовательность целых чисел больших заданного числа:

```
enum n = n : (enum n + 1)
enum n = {n  n + 1  n + 2  n + 3  ...}
```

Последовательность натуральных чисел:

```
nats = enum 1 = {1 2 3 4 ...}
```

Потоки, как ленивые списки

```
ones = 1 : ones
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Определение

Поток – последовательнось элементов, вызываемых (создаваемых) по требованию.

Последовательность целых чисел больших заданного числа:

```
enum n = n : (enum n + 1)
enum n = {n  n + 1  n + 2  n + 3  ...}
```

Последовательность натуральных чисел:

```
nats = enum 1 = {1 2 3 4 ...}
```

Циклическая последовательность:

```
cycle = 1 : 2 : 3 : cycle
cycle = {1 2 3 1 2 3 ...}
```

Потоки, как ленивые списки

```
ones = 1 : ones
head ones = 1
tail ones = ones = 1 : ones
head (tail ones) = 1
```

Список `ones` функционально эквивалентен бесконечному списку (потоку) единиц:

```
ones = {1 1 1 1 1 1 ...}
```

Определение

Поток – последовательнось элементов, вызываемых (создаваемых) по требованию.

Последовательность целых чисел больших заданного числа:

```
enum n = n : (enum n + 1)
enum n = {n  n + 1  n + 2  n + 3  ...}
```

Последовательность натуральных чисел:

```
nats = enum 1 = {1 2 3 4 ...}
```

Циклическая последовательность:

```
cycle = 1 : 2 : 3 : cycle
cycle = {1 2 3 1 2 3 ...}
```

Последовательность символов, считываемых из порта:

```
chars p = (read-char p) : (chars p)
```

Оперирование потоками

Поэлементные арифметические операции:

$\langle + \rangle$ = map $+$
$\langle \times \rangle$ = map $\times$

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×
```

```
ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×
```

```
ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)

a b c d e ... +
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)

a b c d e ... +
  a b c d ... +
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×
```

```
ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×
```

```
ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)

a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
    ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)

a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)

sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)

a b c d e ... +
  a b c d ... +
    a b c ... +
    ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
  sieve h:t =
    h : sieve (filter (not-div? h) t)
  primes = sieve (enum 2)

sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
    ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

```
sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 5 7 9 11 13 15 17 19 21 23 25 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

```
sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 9 11 13 15 17 19 21 23 25 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h : t = h : ((sums h : t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h : t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

```
sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 11 13 17 19 23 25 27 29 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h : t = h : ((sums h : t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h : t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

```
sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 11 13 17 19 23 25 27 29 ...
2 : 3 : 5 : sieve 7 11 13 17 19 23 25 27 29 ...
```

Оперирование потоками

Поэлементные арифметические операции:

```
⟨+⟩ = map +
⟨×⟩ = map ×

ones ⟨+⟩ ones = {2 2 2 2 ...}
nats ⟨×⟩ nats = {1 4 9 16 ...}
```

Поток факториалов натуральных чисел

```
facts = 1 : nats ⟨×⟩ facts
facts = {1 1 2 6 24 120 720 ...}
```

Поток сумм элементов потока:

```
sums h:t = h : ((sums h:t) ⟨+⟩ t)
```

```
a b c d e ... +
  a b c d ... +
    a b c ... +
      ⋮
a a+b a+b+c ...
```

Последовательность простых чисел:

```
sieve h:t =
  h : sieve (filter (not-div? h) t)
primes = sieve (enum 2)
```

```
sieve 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
2 : sieve 3 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 9 11 13 15 17 19 21 23 25 ...
2 : 3 : sieve 5 7 11 13 17 19 23 25 27 29 ...
2 : 3 : 5 : sieve 7 11 13 17 19 23 25 27 29 ...
2 : 3 : 5 : sieve 7 11 13 17 19 23 27 29 ...
```

Недетерминированная реализация метода бисекции

Рассмотрим «экзотическую» реализацию метода бисекции.

Постановка задачи:

найти с заданной точностью решение уравнения $f(x) = 0$, лежащее в заданном отрезке.

Недетерминированная реализация метода бисекции

Рассмотрим «экзотическую» реализацию метода бисекции.

Постановка задачи:

найти с заданной точностью решение уравнения $f(x) = 0$, лежащее в заданном отрезке.

Решение находится, либо в правой, либо в левой половине отрезка, содержащего решение.

Недетерминированная реализация метода бисекции

Рассмотрим «экзотическую» реализацию метода бисекции.

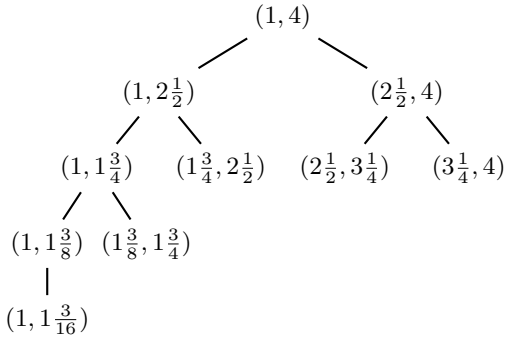
Постановка задачи:

найти с заданной точностью решение уравнения $f(x) = 0$, лежащее в заданном отрезке.

Решение находится, либо в правой, либо в левой половине отрезка, содержащего решение.

1. Для исходного отрезка строим дерево всевозможных вариантов поиска, ограниченное заданной точностью.

Дерево вариантов, ограниченное точностью 10%.



Недетерминированная реализация метода бисекции

Рассмотрим «экзотическую» реализацию метода бисекции.

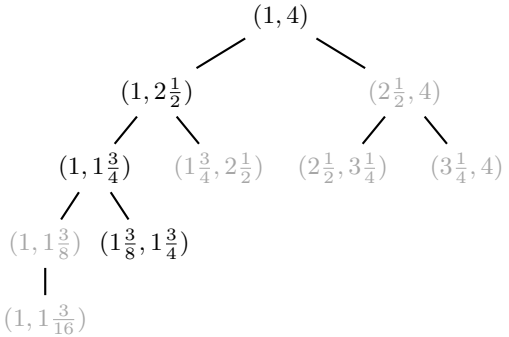
Постановка задачи:

найти с заданной точностью решение уравнения $f(x) = 0$, лежащее в заданном отрезке.

Решение находится, либо в правой, либо в левой половине отрезка, содержащего решение.

1. Для исходного отрезка строим дерево всевозможных вариантов поиска, ограниченное заданной точностью.
2. Из этих вариантов выберем отрезки, содержащие корень уравнения.

Дерево вариантов, ограниченное точностью 10%.



Недетерминированная реализация метода бисекции

Рассмотрим «экзотическую» реализацию метода бисекции.

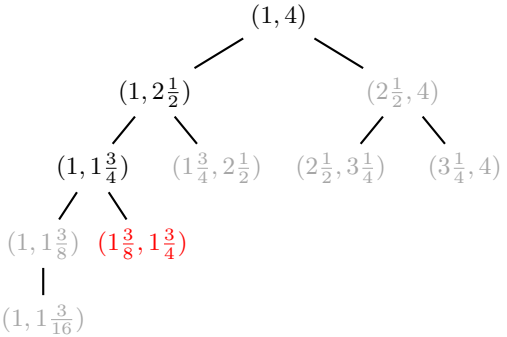
Постановка задачи:

найти с заданной точностью решение уравнения $f(x) = 0$, лежащее в заданном отрезке.

Решение находится, либо в правой, либо в левой половине отрезка, содержащего решение.

1. Для исходного отрезка строим дерево всевозможных вариантов поиска, ограниченное заданной точностью.
2. Из этих вариантов выберем отрезки, содержащие корень уравнения.
3. Результат – последний отрезок.

Дерево вариантов, ограниченное точностью 10%.



Недетерминированная реализация метода бисекции

Определим тип данных – двоичное помеченное дерево:

```
Tree :: = #f
        | tree [Num Num] Tree Tree
tree (node t) (left t) (right t) = t
```

Недетерминированная реализация метода бисекции

Определим тип данных – двоичное помеченное дерево:

```
Tree :: = #f
        | tree [Num Num] Tree Tree
tree (node t) (left t) (right t) = t
```

и свёртку для него

```
foldt f = F where
  F t = and (tree? t)
            (f (node t) (F left t)
                        (F right t))
```

Недетерминированная реализация метода бисекции

Определим тип данных – двоичное помеченное дерево:

```
Tree :: = #f
        | tree [Num Num] Tree Tree
tree (node t) (left t) (right t) = t
```

и свёртку для него

```
foldt f = F where
  F t = and (tree? t)
            (f (node t) (F left t)
                        (F right t))
```

Определим дерево отрезков, ограниченное точностью ε:

```
segments ε [a b] =
  and (b − a)/(b + a) > ε
      (tree [a b] (segments ε [a c])
                  (segments ε [c b]))
      where  c = (a + b)/2
```

Недетерминированная реализация метода бисекции

Определим тип данных – двоичное помеченное дерево:

```
Tree :: = #f
        | tree [Num Num] Tree Tree
tree (node t) (left t) (right t) = t
```

и свёртку для него

```
foldt f = F where
  F t = and (tree? t)
            (f (node t) (F left t)
                        (F right t))
```

Определим дерево отрезков, ограниченное точностью ε:

```
segments ε [a b] =
  and (b − a)/(b + a) > ε
      (tree [a b] (segments ε [a c])
                  (segments ε [c b]))
      where  c = (a + b)/2
```

и функцию, ищущую корень функции f на дереве отрезков s:

```
bisection f s =
  last-node
    (subtree [a b] ↦ (f a) · (f b) ≤ 0
             (segments s))
```

Недетерминированная реализация метода бисекции

Определим тип данных – двоичное помеченное дерево:

```
Tree :: = #f
        | tree [Num Num] Tree Tree
tree (node t) (left t) (right t) = t
```

и свёртку для него

```
foldt f = F where
  F t = and (tree? t)
            (f (node t) (F left t)
                        (F right t))
subtree test? =
  foldt (t,l,r) ↦(and (test? t) (tree t l r))

last-node = foldt (t,l,r) ↦(or l r t)
```

Определим дерево отрезков, ограниченное точностью ε:

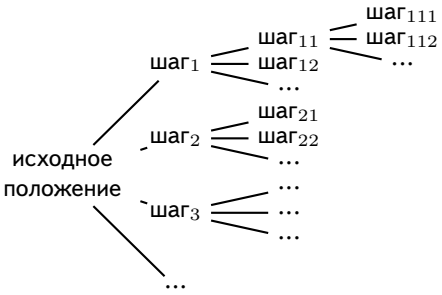
```
segments ε [a b] =
  and (b − a)/(b + a) > ε
      (tree [a b] (segments ε [a c])
                  (segments ε [c b]))
      where  c = (a + b)/2
```

и функцию, ищущую корень функции f на дереве отрезков s:

```
bisection f s =
  last-node
    (subtree [a b] ↦ (f a) · (f b) ≤ 0
             (segments s))
```

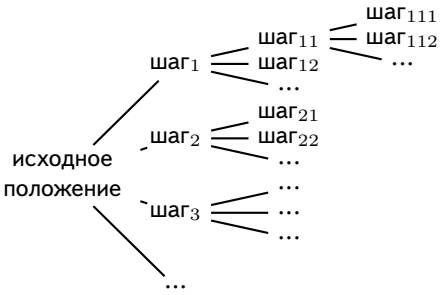
Использование недетерминированного программирования

- выбор оптимальных стратегий;



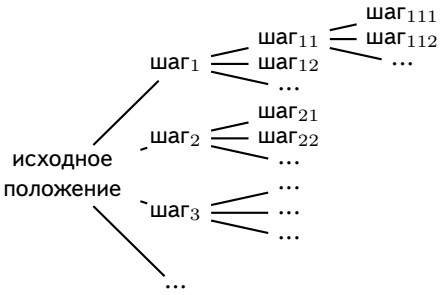
Использование недетерминированного программирования

- выбор оптимальных стратегий;
- символьные преобразования;



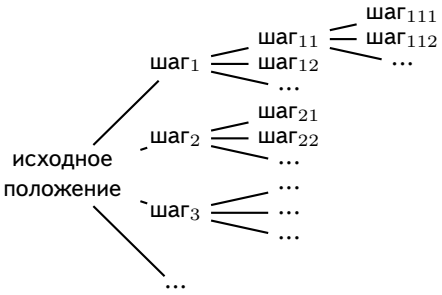
Использование недетерминированного программирования

- выбор оптимальных стратегий;
- символьные преобразования;
- решение логических задач;



Использование недетерминированного программирования

- выбор оптимальных стратегий;
- символьные преобразования;
- решение логических задач;
- самообучающиеся системы.



Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

(**define** (f x) $body$)
(**if** $test$ $expr_1$ $expr_2$)
(**λ** (x) $body$)

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

(**define** (f x) $body$)
(**if** $test$ $expr_1$ $expr_2$)
(**λ** (x) $body$)

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

(**define** $(f\ x)\ body$)
(**if** $test\ expr_1\ expr_2$)
(**λ** $(x)\ body$)

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Основа ленивых вычислений — **вызов по необходимости**. Инструментом для вызова по необходимости может служить **замыкание**:

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

(**define** (f x) $body$)
(**if** $test$ $expr_1$ $expr_2$)
(**λ** (x) $body$)

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Основа ленивых вычислений — **вызов по необходимости**. Инструментом для вызова по необходимости может служить **замыкание**:

```
(define x (λ () (+ 1 2)))
```

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

(**define** (f x) $body$)
(**if** $test$ $expr_1$ $expr_2$)
(**λ** (x) $body$)

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Основа ленивых вычислений — **вызов по необходимости**. Инструментом для вызова по необходимости может служить **замыкание**:

```
(define x (λ () (+ 1 2)))

> x
#\<procedure>
```

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

```
(define (f x) body)
(if test expr1 expr2)
(λ (x) body)
```

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Основа ленивых вычислений — **вызов по необходимости**. Инструментом для вызова по необходимости может служить **замыкание**:

```
(define x (λ () (+ 1 2)))
```

```
> x
#\<procedure>
> (x)
3
```

Ленивые вычисления в строгих языках

В строгих функциональных языках различаются **функции** и **синтаксические формы**:

- **Функции** строги и вычисляются в аппликативном порядке.
- **Синтаксичекие формы** не подчиняются аппликативному порядку и не строги. Не являются объектами первого класса языка и не могут быть выражены через функции.

```
(define (f x) body)
(if test expr1 expr2)
(λ (x) body)
```

С помощью одних только функций реализовать нестрогие конструкции в строгом языке нельзя!

Основа ленивых вычислений — **вызов по необходимости**. Инструментом для вызова по необходимости может служить **замыкание**:

```
(define x (λ () (+ 1 2)))

> x
#\<procedure>
> (x)
3
```

Определение

Функция без аргументов, задерживающая вычисление своего тела, называется **санк** (thunk).

Ленивые вычисления в строгих языках

Определим синтаксическую форму **@**, задерживающую вычисления с помощью замыканий:

```
(define-syntax-rule (@ expr)
  (λ () expr))
```

Ленивые вычисления в строгих языках

Определим синтаксическую форму **@**, задерживающую вычисления с помощью замыканий:

```
(define-syntax-rule (@ expr)
  (λ () expr))
```

И функцию `!`, форсирующую вычисление задержанного выражения:

```
(define (! expr) (expr))
```

Ленивые вычисления в строгих языках

Определим синтаксическую форму **@**, задерживающую вычисления с помощью замыканий:

```
(define-syntax-rule (@ expr)
  (λ () expr))
```

И функцию `!`, форсирующую вычисление задержанного выражения:

```
(define (! expr) (expr))
```

```
> (@ (+ 1 2))
#\<procedure>
> (! (@ (+ 1 2)))
3
```

Ленивые вычисления в строгих языках

Определим синтаксическую форму **@**, задерживающую вычисления с помощью замыканий:

```
(define-syntax-rule (@ expr)
  (λ () expr))
```

И функцию **!**, форсирующую вычисление задержанного выражения:

```
(define (! expr) (expr))
```

```
> (@ (+ 1 2))
#\<procedure>
> (! (@ (+ 1 2)))
3
```

С её помощью мы уже можем создавать потоки:

```
(define ones (cons 1 (@ ones))
```

```
> ones
(1 . #\<procedure>)
> (! (cdr ones))
(1 . #\<procedure>)
```

Ленивые вычисления в строгих языках

Определим синтаксическую форму **@**, задерживающую вычисления с помощью замыканий:

```
(define-syntax-rule (@ expr)
  (λ () expr))
```

И функцию **!**, форсирующую вычисление задержанного выражения:

```
(define (! expr) (expr))
```

```
> (@ (+ 1 2))
#\<procedure>
> (! (@ (+ 1 2)))
3
```

С её помощью мы уже можем создавать потоки:

```
(define ones (cons 1 (@ ones)))
```

```
> ones
(1 . #\<procedure>)
> (! (cdr ones))
(1 . #\<procedure>)
```

```
(define (enum n)
  (cons n (@ (enum (+ n 1)))))
```

```
> (enum 1)
(1 . #\<procedure>)
> (! (cdr (enum 1)))
(2 . #\<procedure>)
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Имеет смысл сразу написать свёртку ленивых списков:

```
(define (@foldr f x0 l)
  (let F ([x l])
    (! (if (empty? x)
           x0
           (f (!head x) (@ (F (!tail x))))))))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Имеет смысл сразу написать свёртку ленивых списков:

```
(define (@foldr f x0 l)
  (let F ([x l])
    (! (if (empty? x)
           x0
           (f (!head x) (@ (F (!tail x))))))))
```

Теперь можно определять различные полезные функции:

```
(define (@list . x)
  (foldr (λ(el res)
           (@cons el res)) '() x))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Имеет смысл сразу написать свёртку ленивых списков:

```
(define (@foldr f x0 l)
  (let F ([x l])
    (! (if (empty? x)
           x0
           (f (!head x) (@ (F (!tail x))))))))
```

Теперь можно определять различные полезные функции:

```
(define (@list . x)
  (foldr (λ(el res)
           (@cons el res)) '() x))

(define* (@map f)
  (@foldr (λ(el res)
            (@cons (f el) res)) '()))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Имеет смысл сразу написать свёртку ленивых списков:

```
(define (@foldr f x0 l)
  (let F ([x l])
    (! (if (empty? x)
           x0
           (f (!head x) (@ (F (!tail x))))))))
```

Теперь можно определять различные полезные функции:

```
(define (@list . x)
  (foldr (λ(el res)
           (@cons el res)) '() x))

(define* (@map f)
  (@foldr (λ(el res)
            (@cons (f el) res)) '()))

(define* (any? p)
  (@foldr (λ(el res)
            (or (p el) res)) #f))
```

Ленивые списки в строгих языках

Определим конструктор ленивых списков, как синтаксическую форму:

```
(define-syntax-rule (@cons h t)
  (cons (@ h) (@ t)))
(define (!head p) (! (car p)))
(define (!tail p) (! (cdr p)))
```

Имеет смысл сразу написать свёртку ленивых списков:

```
(define (@foldr f x0 l)
  (let F ([x l])
    (! (if (empty? x)
           x0
           (f (!head x) (@ (F (!tail x))))))))
```

Теперь можно определять различные полезные функции:

```
(define (@list . x)
  (foldr (λ(el res)
           (@cons el res)) '() x))

(define* (@map f)
  (@foldr (λ(el res)
            (@cons (f el) res)) '()))

(define* (any? p)
  (@foldr (λ(el res)
            (or (p el) res)) #f))

(define (length l)
  (if (empty? l)
      0
      (+ 1 (length (!tail l)))))
```

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,
4. создание первого элемента списка,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,
4. создание первого элемента списка,
5. обработка элемента функцией f,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,
4. создание первого элемента списка,
5. обработка элемента функцией f,
6. обработка элемента предикатом p и фильтрация,

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,
4. создание первого элемента списка,
5. обработка элемента функцией f,
6. обработка элемента предикатом p и фильтрация,
7. включение (или нет) элемента в результат.

Конвеерная обработка списков

```
set (filter p (map f [1 2 3 4]))
```

Обработка строгих списков:

1. `set` требует результат фильтрации,
2. `filter` требует результат отображения,
3. `map` требует список,
4. создание всего списка,
5. поэлементная обработка функцией f,
6. поэлементная обработка предикатом p и фильтрация,
7. удаление дубликатов.

Обработка ленивых списков:

1. `set` требует первый элемент из фильтрованного списка,
2. `filter` требует первый элемент у списка-отображения,
3. `map` требует первый элемент списка,
4. создание первого элемента списка,
5. обработка элемента функцией f,
6. обработка элемента предикатом p и фильтрация,
7. включение (или нет) элемента в результат.
8. При необходимости, повторение с шага 1.

Обработка файлов

Обработка списков

```
variables = pipe  file->lines
                  (map string->words)
                  (filter set-operator?)
                  (map first)
                  set

set-operator? s = (second s)=":="

file->lines = pipe  open-input-file
                    lines
                    stream->list

string->words = pipe  open-input-string
                      words
                      stream->list
```

Обработка ленивых списков

```
variables = pipe  file->lines
                  (map string->words)
                  (filter set-operator?)
                  (map first)
                  set

set-operator? s = (second s)=":="

file->lines = pipe  open-input-file
                    lines

string->words = pipe  open-input-string
                      words

lines p = (read-line p) : (lines p)
words p = (read p) : (words p)
```

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Во многих ЯП введён синтаксис для генерации перечислимых множеств.

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Во многих ЯП введён синтаксис для генерации перечислимых множеств.

- Haskell:

```
S = [ 2 * x | x <- [1..], x*x > 3 ]
```

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Во многих ЯП введён синтаксис для генерации перечислимых множеств.

- HASKELL:
  ```
  S = [ 2 * x | x <- [1..], x*x > 3 ]
  ```
- PYTHON:
  ```
  S = [2*x for x in range(101) if x**2 > 3]
  ```

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Во многих ЯП введён синтаксис для генерации перечислимых множеств.

- HASKELL:
```
S = [ 2 * x | x <- [1..], x*x > 3 ]
```
- PYTHON:
```
S = [2*x for x in range(101) if x**2 > 3]
```
- C#:
```
var s = from x in Enumerable.Range(0, 100)
        where x*x > 3
        select x*2;
```

Генерация списков

В математике принято обозначение для определения множеств:

$$S = \left\{ 2x | x \in N, x^2 > 3 \right\}$$

$$A = \left\{ x | x \in R, x = x^2 \right\}$$

$$T = \left\{ (a, b, c) | a, b, c \in N, a^2 = b^2 + c^2 \right\}$$

Для перечислимых множеств мы можем повторить эти определения на ФЯП:

```
S = map (2 *) (filter (> 3) nats)
```

Во многих ЯП введён синтаксис для генерации перечислимых множеств.

- HASKELL:
```
S = [ 2 * x | x <- [1..], x*x > 3 ]
```
- PYTHON:
```
S = [2*x for x in range(101) if x**2 > 3]
```
- C#:
```
var s = from x in Enumerable.Range(0, 100)
        where x*x > 3
        select x*2;
```
- FORMICA:
```
(define S (collect (* 2 x)
            [x <- (in-range 100)]
            (> (* x x) 3)))
```

Генерация последовательностей

Генерация списков является обобщением итеративной свёртки для произвольных индуктивных типов данных.

Примеры генераторов и итераторов SCHEME:

```
in-generator
in-list
in-range
in-naturals
in-string
in-hash
in-port
in-lines
in-cycle
...
```

Генерация последовательностей

Генерация списков является обобщением итеративной свёртки для произвольных индуктивных типов данных.

Примеры генераторов и итераторов SCHEME:

```
in-generator
in-list
in-range
in-naturals
in-string
in-hash
in-port
in-lines
in-cycle
...
```

Пример: генерация «египетских треугольников»:

```
(define in-triangles
  (collect (list a b c)
   [a <- (in-naturals)]
   [b <- (in-range a)]
   [c <- (in-range b)]
   (= (* a a) (+ (* b b) (* c c)))))
```

Генерация последовательностей

Генерация списков является обобщением итеративной свёртки для произвольных индуктивных типов данных.

Примеры генераторов и итераторов SCHEME:

```
in-generator
in-list
in-range
in-naturals
in-string
in-hash
in-port
in-lines
in-cycle
...
```

Пример: генерация «египетских треугольников»:

```
(define in-triangles
  (collect (list a b c)
   [a <- (in-naturals)]
   [b <- (in-range a)]
   [c <- (in-range b)]
   (= (* a a) (+ (* b b) (* c c)))))

> (stream-take x 4)
'((5 4 3) (10 8 6) (13 12 5) (15 12 9))

> (stream-ref x 300)
'(407 385 132)

> (stream-first (collect x
                 [x <- in-triangles]
                 (> (area x) 100)))
'(25 20 15)
```

Генерация последовательностей

Генерация списков является обобщением итеративной свёртки для произвольных индуктивных типов данных.

Примеры генераторов и итераторов SCHEME:

```
in-generator
in-list
in-range
in-naturals
in-string
in-hash
in-port
in-lines
in-cycle
...
```

Пример: генерация «египетских треугольников»:

```
(define in-triangles
  (collect (list a b c)
   [a <- (in-naturals)]
   [b <- (in-range a)]
   [c <- (in-range b)]
   (= (* a a) (+ (* b b) (* c c)))))

> (stream-take x 4)
'((5 4 3) (10 8 6) (13 12 5) (15 12 9))

> (stream-ref x 300)
'(407 385 132)

> (stream-first (collect x
                 [x <- in-triangles]
                 (> (area x) 100)))
'(25 20 15)
```

Генерация последовательностей

Генераторы и итераторы, в C#:

```
foreach (T var in IEnumerable<T> seq)
  body
```

Генерация последовательностей

Генераторы и итераторы, в C#:

```
foreach (T var in IEnumerable<T> seq)
  body

public interface IEnumerator
{
  bool MoveNext();
  object Current { get };
  void Reset();
}

public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Генерация последовательностей

Генераторы и итераторы, в C#:

```
foreach (T var in IEnumerable<T> seq)
  body
```

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get };
  void Reset();
}

public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

В ленивом языке HASKELL все типы являются генераторами.

Подведение итогов

Нестрогие вычисления выгодны

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

- когда вычисления существенно итеративны;

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

- когда вычисления существенно итеративны;
- при интенсивном использовании исключений, операций ввода-вывода, последовательных действий.

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

- когда вычисления существенно итеративны;
- при интенсивном использовании исключений, операций ввода-вывода, последовательных действий.

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

- когда вычисления существенно итеративны;
- при интенсивном использовании исключений, операций ввода-вывода, последовательных действий.

Не существует абсолютно строгих и нестрогих языков программирования.

Подведение итогов

Нестрогие вычисления выгодны

- когда часть вычислений может быть опущена;
- при работе с концептуально бесконечными типами данных;
- когда выгодна мемоизация.

Строгие вычисления выгодны

- когда вычисления существенно итеративны;
- при интенсивном использовании исключений, операций ввода-вывода, последовательных действий.

Не существует абсолютно строгих и нестрогих языков программирования.

При грамотно разработанной логике типов и вычислительных процессов строгость вычислений не должна быть принципиальной.

Закрепление материала

- Какой результат мы получим в строгом и нестрогом языках при выполнении следующей программы?

```
(define infinity (λ () infinity))
((λ (x) 3) (inifinity))
```

Закрепление материала

- Какой результат мы получим в строгом и нестрогом языках при выполнении следующей программы?

```
(define infinity (λ () infinity))
((λ (x) 3) (inifinity))
```

- Какой порядок роста у функции `append` в строгом и ленивом языках?

```
append a b = foldr (:) b a
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

```
sqr (sqr (sqr 2)
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * ((sqr 2) * (sqr 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * ((sqr 2) * (sqr 2))
((2 * 2) * (2 * 2)) * ((2 * 2) * (2 * 2))
```

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

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

Закрепление материала

- Каким образом вычисляется следующее выражение?

```
sqr (sqr (sqr 2))
sqr x = x * x
```

- В аппликативном порядке:

```
(sqr (sqr (sqr 2))
(sqr (sqr (2 * 2))
(sqr (sqr 4))
(sqr (4 * 4))
(sqr 16)
16 * 16
256
```

- В нормальном порядке:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * (sqr (sqr 2))
((2 * 2) * (sqr 2)) * (sqr (sqr 2))
(4 * (sqr 2)) * (sqr (sqr 2))
(4 * (2 * 2)) * (sqr (sqr 2))
(4 * 4) * (sqr (sqr 2))
16 * (sqr (sqr 2))
...
256
```

- В ленивых вычислениях:

```
sqr (sqr (sqr 2)
(sqr (sqr 2)) * (sqr (sqr 2))
((sqr 2) * (sqr 2)) * ((sqr 2) * (sqr 2))
((2 * 2) * (2 * 2)) * ((2 * 2) * (2 * 2))
(4 * 4) * (4 * 4)
16 * 16
256
```