Лекция 5

Абстракция процедур

Функциональное и логическое программирование

КамчатГТУ, 2013 г.

1. Абстракция данных и процедур
   - Мотивационные примеры

2. Комбинирование функций и данных
   - Аппликация
   - Каррирование
   - Сечение
   - Замыкания

3. ООП на функциях
   - Абстракция
   - Инкапсуляция
   - Полиморфизм
   - Наследование

Абстракция

Абстракция данных:

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
    - определение массивов,

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
    - определение массивов,
    - структур и объектов,

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
    - определение массивов,
    - структур и объектов,
    - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:
  - аппликация,

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:
  - аппликация,
  - композиция,

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:
  - аппликация,
  - композиция,
  - замыкание.

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:
  - аппликация,
  - композиция,
  - замыкание.
- Отделение деталей реализации процедуры от способа её использования барьером абстракции.

Абстракция

Абстракция данных:

- Построение составных типов данных путём комбинирования простых типов. Инструменты:
  - определение массивов,
  - структур и объектов,
  - функциональных типов и т.д.
- Отделение деталей реализации от способа использования барьером абстракции.
- Объединение типа данных со множеством операций над этим типом.

Абстракция процедур

- Построение вычислительных процедур путём комбинирования других процедур. Инструменты:
  - аппликация,
  - композиция,
  - замыкание.
- Отделение деталей реализации процедуры от способа её использования барьером абстракции.
- Объединение процедур со множеством операций над ними.

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

sumsqr 0 = 0
sumsqr i = i^2 + sumsqr $i - 1$

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

sumsqr 0 = 0
sumsqr i = i^2 + sumsqr $i - 1$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

sumf f 0 = 0
sumf f i = (f i) + sumf f ($i - 1$)

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

sumsqr 0 = 0
sumsqr i = i^2 + sumsqr $i - 1$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

sumf f 0 = 0
sumf f i = (f i) + sumf f ($i - 1$)

Целочисленный аккумулятор:

accum g x_0 f n = F n
  where F 0 = x_0
        F i = g (f i) (F ($i - 1$))

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

sumsqr 0 = 0
sumsqr i = i^2 + sumsqr $i - 1$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

sumf f 0 = 0
sumf f i = (f i) + sumf f ($i - 1$)

Целочисленный аккумулятор:

accum g x_0 f n = F n
  where F 0 = x_0
        F i = g (f i) (F ($i - 1$))

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

$$\texttt{sumsqr}\ 0 = 0$$
$$\texttt{sumsqr}\ i = i^2 + \texttt{sumsqr}\ i - 1$$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

$$\texttt{sumf}\ f\ 0 = 0$$
$$\texttt{sumf}\ f\ i = (f\ i) + \texttt{sumf}\ f\ (i-1)$$

Целочисленный аккумулятор:

$$\texttt{accum}\ g\ x_0\ f\ n = \texttt{F}\ n$$
$$\quad \texttt{where}\ \texttt{F}\ 0 = x_0$$
$$\qquad \texttt{F}\ i = g\ (f\ i)\ (\texttt{F}\ (i-1))$$

$$\texttt{sumsqr}\ n = \texttt{sumf}\ (x \mapsto x^2)\ n$$

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

```
sumsqr 0 = 0
sumsqr i = i^2 + sumsqr i − 1
```

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

```
sumf f 0 = 0
sumf f i = (f i) + sumf f (i − 1)
```

Целочисленный аккумулятор:

```
accum g x0 f n = F n
  where F 0 = x0
        F i = g (f i) (F (i − 1))
```

```
sumsqr n = sumf (x ↦ x^2) n
sumf f n = accum (+) 0 f n
```

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

$$\texttt{sumsqr}\ 0\ =\ 0$$
$$\texttt{sumsqr}\ i\ =\ i^2\ +\ \texttt{sumsqr}\ i-1$$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

$$\texttt{sumf}\ f\ 0\ =\ 0$$
$$\texttt{sumf}\ f\ i\ =\ (f\ i)\ +\ \texttt{sumf}\ f\ (i-1)$$

Целочисленный аккумулятор:

$$\texttt{accum}\ g\ x_0\ f\ n\ =\ \texttt{F}\ n$$
$$\quad \texttt{where}\ \texttt{F}\ 0\ =\ x_0$$
$$\qquad \texttt{F}\ i\ =\ g\ (f\ i)\ (\texttt{F}\ (i-1))$$

$$\texttt{sumsqr}\ n\ =\ \texttt{sumf}\ (x \mapsto x^2)\ n$$
$$\texttt{sumf}\ f\ n\ =\ \texttt{accum}\ (+)\ 0\ f\ n$$
$$\texttt{prodf}\ f\ n\ =\ \texttt{accum}\ (*)\ 1\ f\ n$$

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

```
sumsqr 0 = 0
sumsqr i = i^2 + sumsqr i-1
```

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

```
sumf f 0 = 0
sumf f i = (f i) + sumf f (i-1)
```

Целочисленный аккумулятор:

```
accum g x0 f n = F n
  where F 0 = x0
        F i = g (f i) (F (i-1))
```

```
sumsqr n = sumf (x ↦ x^2) n
sumf f n = accum (+) 0 f n
prodf f n = accum (*) 1 f n
factorial n = prodf id n
```

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

```
sumsqr 0 = 0
sumsqr i = i^2 + sumsqr i − 1
```

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

```
sumf f 0 = 0
sumf f i = (f i) + sumf f (i − 1)
```

Целочисленный аккумулятор:

```
accum g x0 f n = F n
  where F 0 = x0
        F i = g (f i) (F (i − 1))
```

```
sumsqr n = sumf (x ↦ x^2) n
sumf f n = accum (+) 0 f n
prodf f n = accum (*) 1 f n
factorial n = prodf id n
power x n = prodf (i ↦ x) n
```

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

```
sumsqr 0 = 0
sumsqr i = i² + sumsqr i−1
```

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

```
sumf f 0 = 0
sumf f i = (f i) + sumf f (i−1)
```

Целочисленный аккумулятор:

```
accum g x₀ f n = F n
  where F 0 = x₀
        F i = g (f i) (F (i−1))
```

```
sumsqr n = sumf (x ↦ x²) n
sumf f n = accum (+) 0 f n
prodf f n = accum (*) 1 f n
factorial n = prodf id n
power x n = prodf (i ↦ x) n
table f n = accum cons [] f n
```

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

sumsqr 0 = 0
sumsqr i = i^2 + sumsqr $i - 1$

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

sumf f 0 = 0
sumf f i = (f i) + sumf f ($i - 1$)

Целочисленный аккумулятор:

accum g x_0 f n = F n
  where F 0 = x_0
        F i = g (f i) (F ($i - 1$))

sumsqr n = sumf ($x \mapsto x^2$) n

sumf f n = accum (+) 0 f n

prodf f n = accum (*) 1 f n

factorial n = prodf id n

power x n = prodf ($i \mapsto x$) n

table f n = accum cons [] f n

make-list x n = table ($i \mapsto x$) n

Пример абстракции процедур

Вычисление суммы: $\sum_{i=0}^{n} i^2$

```
sumsqr 0 = 0
sumsqr i = i^2 + sumsqr i − 1
```

Вычисление суммы: $\sum_{i=0}^{n} f(i)$

```
sumf f 0 = 0
sumf f i = (f i) + sumf f (i − 1)
```

Целочисленный аккумулятор:

```
accum g x0 f n = F n
  where F 0 = x0
        F i = g (f i) (F (i − 1))
```

```
sumsqr n = sumf (x ↦ x^2) n
sumf f n = accum (+) 0 f n
prodf f n = accum (*) 1 f n
factorial n = prodf id n
power x n = prodf (i ↦ x) n
table f n = accum cons [] f n
make-list x n = table (i ↦ x) n
```

Итеративная реализация:

```
accum g x0 f n = F x0 0
 where F r n = r
       F r i = F (g (f i) r) (i + 1)
```

1. Абстракция данных и процедур
   - Мотивационные примеры

2. Комбинирование функций и данных
   - Аппликация
   - Каррирование
   - Сечение
   - Замыкания

3. ООП на функциях
   - Абстракция
   - Инкапсуляция
   - Полиморфизм
   - Наследование

Комбинирование функций и данных

Инструменты для комбинирования функций и данных:

Комбинирование функций и данных

Инструменты для комбинирования функций и данных:

1. **Абстракция** — создания функций с заданными свойствами

Комбинирование функций и данных

Инструменты для комбинирования функций и данных:

1. **Абстракция** — создания функций с заданными свойствами
2. **Аппликация** — механизм применения функции к аргументам

Комбинирование функций и данных

Инструменты для комбинирования функций и данных:

1. **Абстракция** — создания функций с заданными свойствами
2. **Аппликация** — механизм применения функции к аргументам
3. **Композиция** — механизм комбинирования функций

Комбинирование функций и данных

Инструменты для комбинирования функций и данных:

1. **Абстракция** — создания функций с заданными свойствами
2. **Аппликация** — механизм применения функции к аргументам
3. **Композиция** — механизм комбинирования функций
4. **Замыкание** — механизм комбинирования функций и данных

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

Тип (сигнатура) функций

- **унарная функция:**

$$f :: A \to B$$

- **бинарная функция:**

$$f :: A \times B \to C$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

Тип (сигнатура) функций

- **унарная функция:**

$$f :: A \to B$$

- **бинарная функция:**

$$f :: A \times B \to C$$

- **мультиарная функция:**

$$f :: A \times B \times \ldots \to C$$

- **функции высшего порядка (операторы):**

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin`

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin` $:: Num \to Num$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin` $:: Num \to Num$
- `expt`

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times ... \to C$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times ... \to C$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+`

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr`

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$
- `sumf`

Тип (сигнатура) функций

- **унарная функция:**

$$f :: A \to B$$

- **бинарная функция:**

$$f :: A \times B \to C$$

- **мультиарная функция:**

$$f :: A \times B \times \ldots \to C$$

- **функции высшего порядка (операторы):**

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$
- `sumf` $:: (Num \to Num) \times Num \to Num$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$

- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$
- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ :: (A \to B) \times (B \to C) \to (A \to C)$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times ... \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times ... \to Num$
- `sumsqr` $:: Num \to Num$

- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ :: (A \to B) \times (B \to C) \to (A \to C)$
- `dup`

Тип (сигнатура) функций

- **унарная функция:**

$$f :: A \to B$$

- **бинарная функция:**

$$f :: A \times B \to C$$

- **мультиарная функция:**

$$f :: A \times B \times \ldots \to C$$

- **функции высшего порядка (операторы):**

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$

- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ :: (A \to B) \times (B \to C) \to (A \to C)$
- `dup` $:: (A \to A) \to (A \to A)$

Тип (сигнатура) функций

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \times B \to C$$

- мультиарная функция:

$$f :: A \times B \times \ldots \to C$$

- функции высшего порядка (операторы):

$$f :: (A \to B) \to C$$

$$f :: (A \to B) \times C \to D$$

$$f :: (A \to B) \times C \to (C \to B)$$

$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$

- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ :: (A \to B) \times (B \to C) \to (A \to C)$
- `dup` $:: (A \to A) \to (A \to A)$
- `accum`

Тип (сигнатура) функций

- **унарная функция:**

$$f :: A \to B$$

- **бинарная функция:**

$$f :: A \times B \to C$$

- **мультиарная функция:**

$$f :: A \times B \times \ldots \to C$$

- **функции высшего порядка (операторы):**

$$f :: (A \to B) \to C$$
$$f :: (A \to B) \times C \to D$$
$$f :: (A \to B) \times C \to (C \to B)$$
$$f :: (A \to C) \times (B \to C) \to ((A \times B) \to C)$$

- `sin` $:: Num \to Num$
- `expt` $:: Num \times Num \to Num$
- `+` $:: Num \times \ldots \to Num$
- `sumsqr` $:: Num \to Num$

- `sumf` $:: (Num \to Num) \times Num \to Num$
- $\circ :: (A \to B) \times (B \to C) \to (A \to C)$
- `dup` $:: (A \to A) \to (A \to A)$
- `accum` $:: (A \times B \to B) \times B \times (Num \to A) \times Num \to B$

Оператор аппликации

Оператор аппликации унарной функции имеет следующий тип:

$$apply :: (A \to B) \times A \to B$$

Оператор аппликации

Оператор аппликации унарной функции имеет следующий тип:

$$apply :: (A \to B) \times A \to B$$

Для функции валентности n имеем:

$$apply :: (A_1 \times \ldots \times A_n \to B) \times A_1 \times \ldots \times A_n \to B$$

Оператор аппликации

Оператор аппликации унарной функции имеет следующий тип:

$$apply :: (A \to B) \times A \to B$$

Для функции валентности n имеем:

$$apply :: (A_1 \times ... \times A_n \to B) \times A_1 \times ... \times A_n \to B$$

Реализация зависит от принятой модели вычислений.

Оператор аппликации

Оператор аппликации унарной функции имеет следующий тип:

$$apply :: (A \to B) \times A \to B$$

Для функции валентности n имеем:

$$apply :: (A_1 \times \ldots \times A_n \to B) \times A_1 \times \ldots \times A_n \to B$$

Реализация зависит от принятой модели вычислений.

- Аппликация в и Lisp-е:

$$apply :: (A \times \ldots \to B) \times [A] \to B,$$

то есть, это применение функции к **списку** своих аргументов.

Оператор аппликации

Оператор аппликации унарной функции имеет следующий тип:

$$apply :: (A \to B) \times A \to B$$

Для функции валентности n имеем:

$$apply :: (A_1 \times ... \times A_n \to B) \times A_1 \times ... \times A_n \to B$$

Реализация зависит от принятой модели вычислений.

- Аппликация в и LISP-е:

  $$apply :: (A \times ... \to B) \times [A] \to B,$$

  то есть, это применение функции к **списку** своих аргументов.
- В языках HASKELL, ML и т.п. определена аппликация только для унарных функций. На функции произвольной валентности она расширяется с помощью **каррирования**.

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Что произойдёт при аппликации $g(x)$?

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Что произойдёт при аппликации $g(x)$?

$$g(x) = (y \mapsto x + y)$$

это унарная функция, увеличивающая аргумент на величину x. Например,

$$g(5) = (y \mapsto 5 + y)$$

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Что произойдёт при аппликации $g(x)$?

$$g(x) = (y \mapsto x + y)$$

это унарная функция, увеличивающая аргумент на величину x. Например,

$$g(5) = (y \mapsto 5 + y)$$

Запишем то же самое в бесскобочной нотации:

$$(g\ x)\ y = x + y$$

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Что произойдёт при аппликации $g(x)$?

$$g(x) = (y \mapsto x + y)$$

это унарная функция, увеличивающая аргумент на величину x. Например,

$$g(5) = (y \mapsto 5 + y)$$

Запишем то же самое в бесскобочной нотации:

$$(g\ x)\ y = x + y$$

Если считать аппликацию левоассоциативной, то можно переписать это выражение, как

$$g\ x\ y = x + y,$$

где (g) и $(g\ x)$ – унарные функции.

Каррирование функций

Рассмотрим бинарную функцию

$$f(x, y) = x + y$$

Её можно представить в виде унарной функции g, следующим образом:

$$g = x \mapsto (y \mapsto x + y)$$

Что произойдёт при аппликации $g(x)$?

$$g(x) = (y \mapsto x + y)$$

это унарная функция, увеличивающая аргумент на величину x. Например,

$$g(5) = (y \mapsto 5 + y)$$

Запишем то же самое в бесскобочной нотации:

$$(g\ x)\ y = x + y$$

Если считать аппликацию левоассоциативной, то можно переписать это выражение, как

$$g\ x\ y = x + y,$$

где (g) и $(g\ x)$ – унарные функции.

Подобное преобразование можно сделать и с функцией произвольной валентности:

$$f(x, y, z) \quad \longrightarrow \quad x \mapsto (y \mapsto (z \mapsto f(x, y, z)))$$

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.

Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.
Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Если рассматривать все функции, как каррированные, то типы функций различной валентности будут записываться так:

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.
Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Если рассматривать все функции, как каррированные, то типы функций различной валентности будут записываться так:

- унарная функция:

$$f :: A \to B$$

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.
Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Если рассматривать все функции, как каррированные, то типы функций различной валентности будут записываться так:

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \to B \to C$$

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.
Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Если рассматривать все функции, как каррированные, то типы функций различной валентности будут записываться так:

- унарная функция:
$$f :: A \to B$$
- бинарная функция:
$$f :: A \to B \to C$$
- функция произвольной валентности:
$$f :: A \to B \to \ldots \to C$$

Каррирование функций

Определение

Преобразование аппликации бинарной функции $f : A \times B \to C$ в последовательность аппликаций унарных функций: $f' : A \to (B \to C)$ называется **каррированием**.
Оператор

$$curry : (A \times B \to C) \to (A \to (B \to C))$$

называется **оператором каррирования**.

Если рассматривать все функции, как каррированные, то типы функций различной валентности будут записываться так:

- унарная функция:

$$f :: A \to B$$

- бинарная функция:

$$f :: A \to B \to C$$

- функция произвольной валентности:

$$f :: A \to B \to ... \to C$$

при этом операцию отображения $\to$ считаем левоассоциативной.

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$

$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$
\begin{aligned}
(f)x\ y\ z &= x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))\\
(f\ x)y\ z &= y \mapsto (z \mapsto proc(x, y, z))\\
(f\ x\ y)z &= z \mapsto proc(x, y, z)\\
f\ x\ y\ z &= proc(x, y, z)
\end{aligned}
$$

Различают **левое** и **правое** сечения.

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$
\begin{aligned}
(f)x\ y\ z &= x \mapsto (y \mapsto (z \mapsto proc(x, y, z))) \\
(f\ x)y\ z &= y \mapsto (z \mapsto proc(x, y, z)) \\
(f\ x\ y)z &= z \mapsto proc(x, y, z) \\
f\ x\ y\ z &= proc(x, y, z)
\end{aligned}
$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$
\begin{aligned}
(f)x\ y\ z &= x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))\\
(f\ x)y\ z &= y \mapsto (z \mapsto proc(x, y, z))\\
(f\ x\ y)z &= z \mapsto proc(x, y, z)\\
f\ x\ y\ z &= proc(x, y, z)
\end{aligned}
$$

Различают **левое** и **правое** сечения.

- Левое:

$$(: x) = y \mapsto (y : x)$$

- Правое:

$$(x :) = y \mapsto (x : y)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:x) = y \mapsto (y : x)$$
- Правое:
$$(x:) = y \mapsto (x : y)$$

Примеры:

$$(+\ 4)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:\ x) = y \mapsto (y : x)$$
- Правое:
$$(x\ :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$$(+\ 4)$$
$$(+)$$
$$(-\ 1)$$
$$(1\ -)$$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ \ :)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ :)$

$(<\ 0)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:

$$(: x) = y \mapsto (y : x)$$

- Правое:

$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ \ :)$

$(<\ \ 0)$ = `negative?`

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ :)$

$(<\ 0)$ = `negative?`

$(0\ <)$

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ \ :)$

$(<\ \ 0)$ = `negative?`

$(0\ \ <)$ = `positive?`

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(:\ x) = y \mapsto (y : x)$$
- Правое:
$$(x\ :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ \ :)$

$(<\ \ 0)$ = `negative?`

$(0\ \ <)$ = `positive?`

`(sumf` $(x \mapsto x^2)$`)`

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f) x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x) y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y) z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ :)$

$(<\ 0)$ = `negative?`

$(0\ <)$ = `positive?`

`(sumf` $(x \mapsto x^2)$`)` = `sumsqr`

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ :)$

$(<\ 0)$ = `negative?`

$(0\ <)$ = `positive?`

`(sumf` $(x \mapsto x^2)$`)` = `sumsqr`

`(accum` $+$ `0)`

Сечение функций

Каррирование даёт возможность **сечения (частичного применения, замыкания)** функции путём передачи ей части аргументов.

$$(f)x\ y\ z = x \mapsto (y \mapsto (z \mapsto proc(x, y, z)))$$
$$(f\ x)y\ z = y \mapsto (z \mapsto proc(x, y, z))$$
$$(f\ x\ y)z = z \mapsto proc(x, y, z)$$
$$f\ x\ y\ z = proc(x, y, z)$$

Различают **левое** и **правое** сечения.

- Левое:
$$(: x) = y \mapsto (y : x)$$
- Правое:
$$(x :) = y \mapsto (x : y)$$

Примеры:

$(+\ 4)$

$(+)$

$(-\ 1)$ = `dec`

$(1\ -)$

$(a\ \ :)$

$(<\ \ 0)$ = `negative?`

$(0\ \ <)$ = `positive?`

`(sumf` $(x \mapsto x^2)$`)` = `sumsqr`

`(accum` $+$ `0)` = `sumf`

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.

Аргументы можно отбрасывать только при хвостовом вызове.

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$f\ x\ y = g\ x\ y$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$f\ x\ y = g\ x\ y \qquad \Leftrightarrow \quad f = g$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$f\ x\ y = g\ x\ y \qquad \Leftrightarrow \qquad f = g$$
$$f\ x\ y = g\ z\ x\ y$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$f\ x\ y = g\ x\ y \qquad \Leftrightarrow \qquad f = g$$
$$f\ x\ y = g\ z\ x\ y \qquad \Leftrightarrow \qquad f = (g\ z)$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.

Аргументы можно отбрасывать только при хвостовом вызове.

$$
\begin{array}{lll}
f\ x\ y = g\ x\ y & \Leftrightarrow & f = g \\
f\ x\ y = g\ z\ x\ y & \Leftrightarrow & f = (g\ z) \\
f\ z\ x\ y = g\ w\ x\ y & &
\end{array}
$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$
\begin{array}{lcl}
f\ x\ y = g\ x\ y & \Leftrightarrow & f = g \\
f\ x\ y = g\ z\ x\ y & \Leftrightarrow & f = (g\ z) \\
f\ z\ x\ y = g\ w\ x\ y & \Leftrightarrow & (f\ z) = (g\ w)
\end{array}
$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$
\begin{array}{llll}
f\ x\ y = g\ x\ y & \Leftrightarrow & f = g & \\
f\ x\ y = g\ z\ x\ y & \Leftrightarrow & f = (g\ z) & \\
f\ z\ x\ y = g\ w\ x\ y & \Leftrightarrow & (f\ z) = (g\ w) & \Leftrightarrow \quad f\ z = g\ w.
\end{array}
$$

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$
\begin{array}{lclcl}
f\ x\ y = g\ x\ y & \Leftrightarrow & f = g & & \\
f\ x\ y = g\ z\ x\ y & \Leftrightarrow & f = (g\ z) & & \\
f\ z\ x\ y = g\ w\ x\ y & \Leftrightarrow & (f\ z) = (g\ w) & \Leftrightarrow & f\ z = g\ w.
\end{array}
$$

```
sumsqr n = sumf (x ↦ x^2) n
sumf f n = accum + 0 f n
prodf f n = accum * 1 f n
factorial n = prodf id n
table f n = accum cons [] f n
make-list x n = table (i ↦ x) n
```

Бесточечная нотация

Определение

Бесточечная запись определений функций заключается в отбрасывании одинаковых свободных аргументов, стоящих справа в обеих частях определения.
Аргументы можно отбрасывать только при хвостовом вызове.

$$f\ x\ y = g\ x\ y \quad\Leftrightarrow\quad f = g$$
$$f\ x\ y = g\ z\ x\ y \quad\Leftrightarrow\quad f = (g\ z)$$
$$f\ z\ x\ y = g\ w\ x\ y \quad\Leftrightarrow\quad (f\ z) = (g\ w) \quad\Leftrightarrow\quad f\ z = g\ w.$$

```
sumsqr n = sumf (x ↦ x²) n
sumf f n = accum + 0 f n
prodf f n = accum * 1 f n
factorial n = prodf id n
table f n = accum cons [] f n
make-list x n = table (i ↦ x) n
```

```
sumsqr = sumf (x ↦ x²)
sumf = accum + 0
prodf = accum * 1
factorial = prodf id
table = accum : []
make-list x = table (i ↦ x)
```

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

Замыкание

Создание каррированных функций
и частичного применения основано
на **замыкании** — способности функции
сохранять окружение в котором она была
создана.

$$f = x \mapsto (y \mapsto x + y)$$

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

формальный аргумент

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

фактический аргумент

формальный аргумент

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

фактический аргумент

формальный аргумент

Определение

Замыкание — процедура, которая ссылается на свободные переменные в своём лексическом контексте.

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

фактический аргумент

формальный аргумент

Определение

Замыкание — процедура, которая ссылается на свободные переменные в своём лексическом контексте.

Фактически, замыкание — это функция, создаваемая в теле другой функции и возвращаемая при её выполнении.

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

Определение

Замыкание — процедура, которая ссылается на свободные переменные в своём лексическом контексте.

формальные аргументы

$$f = {\color{blue}x} \mapsto (y \mapsto {\color{blue}x} + y)$$

$$f\ {\color{red}x} = y \mapsto {\color{red}x} + y$$

фактический аргумент

формальный аргумент

Фактически, замыкание — это функция, создаваемая в теле другой функции и возвращаемая при её выполнении.

При этом в функции-замыкании инкапсулируется её лексический контекст.

Замыкание

Создание каррированных функций и частичного применения основано на **замыкании** — способности функции сохранять окружение в котором она была создана.

формальные аргументы

$$f = x \mapsto (y \mapsto x + y)$$

$$f\ x = y \mapsto x + y$$

фактический аргумент

формальный аргумент

Определение

Замыкание — процедура, которая ссылается на свободные переменные в своём лексическом контексте.

Фактически, замыкание — это функция, создаваемая в теле другой функции и возвращаемая при её выполнении.

При этом в функции-замыкании инкапсулируется её лексический контекст.

Замыкание предоставляет способ комбинирования функциональности и данных, связанных и упакованных вместе.

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11

> (define inc (add 1))
> (inc 3)
4
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11

> (define inc (add 1))
> (inc 3)
4

> (define dec (add -1))
> (dec 3)
2
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11

> (define inc (add 1))
> (inc 3)
4

> (define dec (add -1))
> (dec 3)
2
```

```
(define (add n)
  (λ (x) (+ n x)))
```

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11

> (define inc (add 1))
> (inc 3)
4

> (define dec (add -1))
> (dec 3)
2
```

```
(define (add n)
  (λ (x) (+ n x)))
```

Это определение в точности соответствует определению каррированной функции сложения:

add n = $x \mapsto x + n$ = (n +)

add = $n \mapsto (x \mapsto x + n)$ = (+)

Замыкание в Scheme

Пример:

```
(define (add n)
  (define (f x) (+ n x))
  f)

> (add 5)
#<procedure>

> ((add 5) 6)
11

> (define inc (add 1))
> (inc 3)
4

> (define dec (add -1))
> (dec 3)
2
```

```
(define (add n)
  (λ (x) (+ n x)))
```

Это определение в точности соответствует определению каррированной функции сложения:

$$\text{add}\ n\ =\ x \mapsto x + n\ =\ (n\ +)$$
$$\text{add}\ =\ n \mapsto (x \mapsto x + n)\ =\ (+)$$

В Formica существует синтаксис для определения замыканий и каррированных функций:

```
(define (add n) (+ n))
(define add (+))
```

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)
```

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11
```

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11

> (define inc (add 1))
I am add
```

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11

> (define inc (add 1))
I am add

> (inc 5)
I am closure
6
```

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11

> (define inc (add 1))
I am add

> (inc 5)
I am closure
6
```

- Замыкания как объекты первого класса появились в Scheme.

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11

> (define inc (add 1))
I am add

> (inc 5)
I am closure
6
```

- Замыкания как объекты первого класса появились в Scheme.
- Сейчас их можно создавать в языках C#, Delphi с версии 2009, Python, Java Script и во всех функциональных языках.

Замыкание в Scheme

Удостоверимся, что замыкание – это самостоятельный объект.

```
(define (add n)
  (define (f x)
    (displayln "I am closure")
    (+ n x))
  (displayln "I am add")
  f)

> ((add 5) 6)
I am add
I am closure
11

> (define inc (add 1))
I am add

> (inc 5)
I am closure
6
```

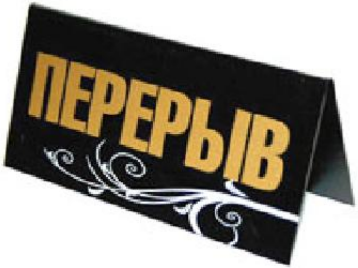
- Замыкания как объекты первого класса появились в Scheme.
- Сейчас их можно создавать в языках C#, Delphi с версии 2009, Python, Java Script и во всех функциональных языках.
- Замыкания можно рассматривать, как функциональный аналог объекта: замыкание, как и объект-экземпляр хранит данные и методы – процедуры обработки данных.

Резонный вопрос

Резонный вопрос

А не сделать ли нам перерыв?

Резонный вопрос



1. Абстракция данных и процедур
   - Мотивационные примеры
2. Комбинирование функций и данных
   - Аппликация
   - Каррирование
   - Сечение
   - Замыкания
3. ООП на функциях
   - Абстракция
   - Инкапсуляция
   - Полиморфизм
   - Наследование

Абстракция комплексных чисел

Определение комплексных чисел

Множество упорядоченных пар (a, b), $a, b \in R$, на которых определены операции сложения, умножения и сопряжения:

$$(a, b) + (c, d) = (a + b, c + d),$$
$$(a, b) \cdot (c, d) = (a^2 - c^2, ac + bd),$$
$$(a, b)^* = (a, -b).$$

Абстракция комплексных чисел

Определение комплексных чисел

Множество упорядоченных пар $(a, b),\ a, b \in R$, на которых определены операции сложения, умножения и сопряжения:

$$(a, b) + (c, d) = (a + b, c + d),$$
$$(a, b) \cdot (c, d) = (a^2 - c^2, ac + bd),$$
$$(a, b)^* = (a, -b).$$

Реализация с помощью пар:

```
(define-type (Comp Real Real))
(:: plus (Comp Comp -> Comp)
(define plus
  (/. (Comp r₁ i₁)
      (Comp r₂ i₂) --> (Comp (+ r₁ r₂)
                             (+ i₁ i₂)))))
(:: mult (Comp Comp -> Comp)
(define mult
  (/. (Comp r₁ i₁)
      (Comp r₂ i₂) --> (Comp (- (* r₁ r₂)
                                (* i₁ i₂))
                             (+ (* r₁ i₂)
                                (* r₂ i₁))))))
(:: conj (Comp -> Comp)
(define conj
  (/. (Comp r i) --> (Comp r (- i)))))
```

Абстракция комплексных чисел

Определение комплексных чисел

Множество упорядоченных пар $(a, b),\ a, b \in R$, на которых определены операции сложения, умножения и сопряжения:

$$(a, b) + (c, d) = (a + b, c + d),$$
$$(a, b) \cdot (c, d) = (a^2 - c^2, ac + bd),$$
$$(a, b)^* = (a, -b).$$

```
> (define a (Comp 1 2))
> (define b (Comp 3 4))
> (plus a b)
(Comp 4 6)
> (mult b (conj b))
(Comp 25 0)
```

Реализация с помощью пар:

```
(define-type (Comp Real Real))

(:: plus (Comp Comp -> Comp)
(define plus
  (/. (Comp r1 i1)
      (Comp r2 i2) --> (Comp (+ r1 r2)
                             (+ i1 i2)))))

(:: mult (Comp Comp -> Comp)
(define mult
  (/. (Comp r1 i1)
      (Comp r2 i2) --> (Comp (- (* r1 r2)
                                (* i1 i2))
                             (+ (* r1 i2)
                                (* r2 i1))))))

(:: conj (Comp -> Comp)
(define conj
  (/. (Comp r i) --> (Comp r (- i)))))
```

Абстракция рациональных чисел

Определение рациональных чисел

Множество упорядоченных пар $(a, b),\ a, b \in Q$, на которых определены операции сложения, умножения и деления:

$$(a, b) + (c, d) = (ad + cb, bd),$$
$$(a, b) \cdot (c, d) = (ac, bd),$$
$$(a, b)/(c, d) = (ad, bc).$$

Абстракция рациональных чисел

Определение рациональных чисел

Множество упорядоченных пар (a, b), $a, b \in Q$, на которых определены операции сложения, умножения и деления:

$$(a, b) + (c, d) = (ad + cb, bd),$$
$$(a, b) \cdot (c, d) = (ac, bd),$$
$$(a, b)/(c, d) = (ad, bc).$$

Реализация с помощью пар:

```
(define-type (Rat Int Nat))

(:: plus (Rat Rat -> Rat)
(define plus
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (Rat (+ (* n1 d2)
                              (* d2 n1))
                           (* d1 d2)))))

(:: mult (Rat Rat -> Rat)
(define mult
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (Rat (* n1 n2)
                           (* d1 d2)))))
```

Абстракция рациональных чисел

Определение рациональных чисел

Множество упорядоченных пар $(a, b),\ \ a, b \in Q$, на которых определены операции сложения, умножения и деления:

$$(a, b) + (c, d) = (ad + cb, bd),$$
$$(a, b) \cdot (c, d) = (ac, bd),$$
$$(a, b)/(c, d) = (ad, bc).$$

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (plus a b)
(Rat 7 6)
> (mult a b)
(Rat 2 6)
> (plus a a)
(Rat 2 2)
```

Реализация с помощью пар:

```
(define-type (Rat Int Nat))

(:: plus (Rat Rat -> Rat))
(define plus
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (Rat (+ (* n1 d2)
                               (* d2 n1))
                            (* d1 d2)))))

(:: mult (Rat Rat -> Rat))
(define mult
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (Rat (* n1 n2)
                            (* d1 d2)))))
```

Абстракция рациональных чисел

Определение рациональных чисел

Множество упорядоченных пар $(a, b),\ a, b \in Q$, на которых определены операции сложения, умножения и деления:

$$(a, b) + (c, d) = (ad + cb, bd),$$
$$(a, b) \cdot (c, d) = (ac, bd),$$
$$(a, b)/(c, d) = (ad, bc).$$

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (plus a b)
(Rat 7 6)
> (mult a b)
(Rat 1 3)
> (plus a a)
(Rat 1 1)
```

Реализация с помощью пар:

```
(define-type (Rat Int Nat))
(:: make-rat (Int Nat -> Rat))
(define (make-rat n d)
  (let ([x (gcd n d)])
   (Rat (/ n x) (/ d x)))))
(:: plus (Rat Rat -> Rat))
(define plus
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (make-rat (+ (* n1 d2)
                                    (* d2 n1))
                                 (* d1 d2)))))

(:: mult (Rat Rat -> Rat))
(define mult
  (/. (Rat d1 n1)
      (Rat d2 n2) --> (make-rat (* n1 n2)
                                 (* d1 d2)))))
```

Совместное использование абстракций

- Как использовать функции `plus` и `mul` для обоих типов?

Совместное использование абстракций

- Как использовать функции `plus` и `mul` для обоих типов?

  Можно создать обобщённые операции `plus` и `mult`, выбирающие процедуры `plus-comp`/`plus-rat` или `mult-comp`/`mult-rat` в зависимости от типа аргументов.

Совместное использование абстракций

- Как использовать функции `plus` и `mul` для обоих типов?

  Можно создать обобщённые операции `plus` и `mult`, выбирающие процедуры `plus-comp`/`plus-rat` или `mult-comp`/`mult-rat` в зависимости от типа аргументов.
- Как переводить из одного типа в другой?

Совместное использование абстракций

- Как использовать функции `plus` и `mul` для обоих типов?

  Можно создать обобщённые операции `plus` и `mult`, выбирающие процедуры `plus-comp`/`plus-rat` или `mult-comp`/`mult-rat` в зависимости от типа аргументов.

- Как переводить из одного типа в другой?

  Можно создать функцию-трансформер, знающую все определённые типы и переводящую из одного типа в другой.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать разные типы, знают операторы сложения и умножения.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать разные типы, знают операторы сложения и умножения.
- О том, какие бывают типы знают функции `type` и функция трансформер.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать разные типы, знают операторы сложения и умножения.
- О том, какие бывают типы знают функции `type` и функция трансформер.

Комбинирование абстракции данных и процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать разные типы, знают операторы сложения и умножения.
- О том, какие бывают типы знают функции `type` и функция трансформер.

Комбинирование абстракции данных и процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать, прописано в типах и объекты сами умеют производить эти операции.

Совместное использование абстракций

Упор на абстракцию процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать разные типы, знают операторы сложения и умножения.
- О том, какие бывают типы знают функции `type` и функция трансформер.

Комбинирование абстракции данных и процедур

- Существует несколько определённых типов и возможность создавать объекты этих типов.
- О том, как складывать и умножать, прописано в типах и объекты сами умеют производить эти операции.
- О том, какие бывают типы, говорят сами объекты и сами же объекты умеют приводить типы друг к другу.

Объектно-ориентированное программирование

Объект

Наследование

Инкапсуляция

Полиморфизм

Объектно-ориентированное программирование

Объект

Сущность, которой можно посылать сообщения, и которая может на них реагировать, используя свои данные.

Инкапсуляция

Наследование

Полиморфизм

Объектно-ориентированное программирование

Объект

Сущность, которой можно посылать сообщения, и которая может на них реагировать, используя свои данные.

Инкапсуляция

Свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя и других объектов.

Наследование

Полиморфизм

Объектно-ориентированное программирование

Объект

Сущность, которой можно посылать сообщения, и которая может на них реагировать, используя свои данные.

Инкапсуляция

Свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя и других объектов.

Наследование

Свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Полиморфизм

Объектно-ориентированное программирование

Объект

Сущность, которой можно посылать сообщения, и которая может на них реагировать, используя свои данные.

Наследование

Свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Инкапсуляция

Свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя и других объектов.

Полиморфизм

Свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Замыкание в качестве объекта

Функциональный объект:

```
(define (Obj x y)
  (/. 'x --> x
      'y --> y
      'sum --> (+ x y)
      'show --> (printf "x: a y: a" x y)
      m --> (error "Unknown message!" m)))
```

Замыкание в качестве объекта

Функциональный объект:

```
(define (Obj x y)
  (/. 'x --> x
      'y --> y
      'sum --> (+ x y)
      'show --> (printf "x: a y: a" x y)
      m --> (error "Unknown message!" m)))
```

Замыкание в качестве объекта

Функциональный объект:

```
(define (Obj x y)
  (/. 'x --> x
      'y --> y
      'sum --> (+ x y)
      'show --> (printf "x: a y: a" x y)
      m --> (error "Unknown message!" m)))
```

```
> (define a (Obj 5 6))
> (define b (Obj 7 8))
```

Замыкание в качестве объекта

Функциональный объект:

```
(define (Obj x y)
  (/. 'x --> x
      'y --> y
      'sum --> (+ x y)
      'show --> (printf "x: a y: a" x y)
      m --> (error "Unknown message!" m)))
```

```
> (define a (Obj 5 6))
> (define b (Obj 7 8))
> (a 'x)
5
> (b 'y)
8
> (a 'sum)
11
> (b 'show)
x:7 y:8
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

```
> (define a (complex 1 2))
> (define b (complex 3 4))
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

```
> (define a (complex 1 2))
> (define b (complex 3 4))
> (a 'show)
"1+2i"
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

```
> (define a (complex 1 2))
> (define b (complex 3 4))
> (a 'show)
"1+2i"
> ((a '+ b) 'show)
"4+6i"
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

```
> (define a (complex 1 2))
> (define b (complex 3 4))
> (a 'show)
"1+2i"
> ((a '+ b) 'show)
"4+6i"
> ((b '* (b 'conj)) 'show)
"25+0i"
```

Замыкание, как объект

Реализация комплексных чисел:

```
(define (Comp r i)
  (/. 'Re --> r
      'Im --> i

      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))

      '* x --> (Comp (- (* (x 'Re) r) (* (x 'Im) i))
                     (+ (* (x 'Re) i) (* (x 'Im) r)))

      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i)))
```

```
> (define a (complex 1 2))
> (define b (complex 3 4))
> (a 'show)
"1+2i"
> ((a '+ b) 'show)
"4+6i"
> ((b '* (b 'conj)) 'show)
"25+0i"
```

Определение операций

```
(define (<+> a b) (a '+ b))
(define (<*> a b) (a '* b))
(define (conj a) (a 'conj))
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x)
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x))
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x))
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (a 'show)
"1/2"
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x))
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (a 'show)
"1/2"
> ((a '+ b) 'show)
"7/6"
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x))
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (a 'show)
"1/2"
> ((a '+ b) 'show)
"7/6"
> ((a '* b) 'show)
"1/3"
```

Замыкание, как объект

Реализация рациональных чисел:

```
(define (Rat n d)
  (define x (gcd n d))
  (define num (/ n x))
  (define den (/ d x))
  (/. 'num --> num
      'den --> den

      '+ x --> (Rat (+ (* (x 'num) den)
                       (* (x 'den) num))
                    (* (x 'den) den))

      '* x --> (Rat (* (x 'num) num) (* (x 'den) den))

      'show --> (format "~a/~a"  num den)))
```

```
> (define a (Rat 1 2))
> (define b (Rat 2 3))
> (a 'show)
"1/2"
> ((a '+ b) 'show)
"7/6"
> ((a '* b) 'show)
"1/3"
```

Определение операций

```
(define (<+> a b) (a '+ b))
(define (<*> a b) (a '* b))
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))

(define (complex? a)
  (and (object? a) (a 'complex?)))
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))

(define (complex? a)
  (and (object? a) (a 'complex?)))

(define (rational? a)
  (and (object? a) (a 'rational?)))
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))

(define (complex? a)
  (and (object? a) (a 'complex?)))

(define (rational? a)
  (and (object? a) (a 'rational?)))
```

```
> (object? (Comp 1 2))
#t
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))

(define (complex? a)
  (and (object? a) (a 'complex?)))

(define (rational? a)
  (and (object? a) (a 'rational?)))
```

```
> (object? (Comp 1 2))
#t
> (object? 3)
#f
```

Система типов

Добавление типа:

```
(define (Comp r i)
  (/. (or 'object? 'complex?) --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp
                 (- (* (x 'Re) r) (* (x 'Im) i))
                 (+ (* (x 'Re) i) (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      _ --> #f))
```

Квантификаторы типов

```
(define (object? a)
  (and (function? a) (a 'object?)))

(define (complex? a)
  (and (object? a) (a 'complex?)))

(define (rational? a)
  (and (object? a) (a 'rational?)))
```

```
> (object? (Comp 1 2))
#t
> (object? 3)
#f
> (complex? (Comp 1 2))
#t
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (/.
    ...
    '<- x --> (cond
                [(complex? x) x]
                [(or (rational? x) (number? x)) (Comp x 0)]
                [else
                  (error "Can't convert to complex " x)]))]
    ...))
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (/.
    ...
    '<- x --> (cond
                [(complex? x) x]
                [(or (rational? x) (number? x)) (Comp x 0)]
                [else
                  (error "Can't convert to complex " x)])))]
    ...))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (/.
    ...
    '<- x --> (cond
                [(complex? x) x]
                [(or (rational? x) (number? x)) (Comp x 0)]
                [else
                  (error "Can't convert to complex " x)]))]
    ...))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
> (show (z '<- 4))
"4+0i"
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (/.
    ...
    '<- x --> (cond
                [(complex? x) x]
                [(or (rational? x) (number? x)) (Comp x 0)]
                [else
                  (error "Can't convert to complex " x)])))]
    ...))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
> (show (z '<- 4))
"4+0i"
> (show (z '<- r))
"2/3+0i
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (/.
    ...
    '<- x --> (cond
                [(complex? x) x]
                [(or (rational? x) (number? x)) (Comp x 0)]
                [else
                  (error "Can't convert to complex " x)]))]
    ...))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))

> (show (z '<- 4))
"4+0i"

> (show (z '<- r))
"2/3+0i

> (show (z '<- z))
"1+2i"
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                  [(complex? x) x]
                  [(or (rational? x) (number? x)) (Comp x 0)]
                  [else
                    (error "Can't convert to complex " x)]))]

     '+ x --> (let ([x ((self) '<- x)])
                  (Comp (<+> (x 'Re) r)
                        (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                  [(complex? x) x]
                  [(or (rational? x) (number? x)) (Comp x 0)]
                  [else
                    (error "Can't convert to complex " x)])))]

     '+ x --> (let ([x ((self) '<- x)])
                 (Comp (<+> (x 'Re) r)
                       (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                 [(complex? x) x]
                 [(or (rational? x) (number? x)) (Comp x 0)]
                 [else
                   (error "Can't convert to complex " x)]))]

     '+ x --> (let ([x ((self) '<- x)])
                (Comp (<+> (x 'Re) r)
                      (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
> (show (z '+ z))
"2+4i"
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                  [(complex? x) x]
                  [(or (rational? x) (number? x)) (Comp x 0)]
                  [else
                    (error "Can't convert to complex " x)]))]

     '+ x --> (let ([x ((self) '<- x)])
                (Comp (<+> (x 'Re) r)
                      (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))

> (show (z '+ z))
"2+4i"

> (show (z '+ 4))
"5+2i"
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                 [(complex? x) x]
                 [(or (rational? x) (number? x)) (Comp x 0)]
                 [else
                   (error "Can't convert to complex " x)]))]

     '+ x --> (let ([x ((self) '<- x)])
                (Comp (<+> (x 'Re) r)
                      (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))

> (show (z '+ z))
"2+4i"

> (show (z '+ 4))
"5+2i"

> (show (z '+ r))
"5/3+2i"
```

Система типов

Приведение к типу:

```
(define (Comp r i)
  (define (self) (Comp r i))
  (/.
     ...
     '<- x --> (cond
                  [(complex? x) x]
                  [(or (rational? x) (number? x)) (Comp x 0)]
                  [else
                    (error "Can't convert to complex " x)]))]

     '+ x --> (let ([x ((self) '<- x)])
                 (Comp (<+> (x 'Re) r)
                       (<+> (x 'Im) i)))
     ...)))

(define (<+> a b)
  (if (object? a) (a '+ b) (+ a b)))
```

```
> (define z (Comp 1 2))
> (define r (Rat 2 3))
> (show (z '+ z))
"2+4i"
> (show (z '+ 4))
"5+2i"
> (show (z '+ r))
"5/3+2i"
> (show (z '+ (r '+ 1)))
"8/3+2i"
```

Наследование в функциональных объектах

Базовый объект

```
(define Obj
  (/. 'object? --> #t
      _ --> #f))
```

Наследование в функциональных объектах

Базовый объект

```
(define Obj
  (/. 'object? --> #t
      _ --> #f))
```

→

Объект-наследник

```
(define Alg
  (define parent Obj)
  (/. 'algebraic? --> #t
      m --> (parent m))))
```

Наследование в функциональных объектах

Базовый объект

```
(define Obj
  (/. 'object? --> #t
      _ --> #f))
```

Объект-наследник

```
(define Alg
  (define parent Obj)
  (/. 'algebraic? --> #t
      m --> (parent m))))
```

Наследник Alg

```
(define (Comp r i)
  (define parent Alg)
  (/. 'complex? --> #t
      ...
      m --> (parent m))))
```

Наследование в функциональных объектах

Базовый объект

```
(define Obj
  (/. 'object? --> #t
      _ --> #f))
```

Объект-наследник

```
(define Alg
  (define parent Obj)
  (/. 'algebraic? --> #t
      m --> (parent m))))
```

Наследник Alg

```
(define (Comp r i)
  (define parent Alg)
  (/. 'complex? --> #t
      ...
      m --> (parent m))))
```

Наследник Alg

```
(define (Rat n d)
  (define parent Alg)
  (/. 'rational? --> #t
      ...
      m --> (parent m))))
```

Наследование в функциональных объектах

Объект с виртуальными методами

```
(define (Alg v)
  (define parent Obj)
  (/. 'algebraic? --> #t
      '+ x --> (format "(~a + ~a)" v (show x))
      '* x --> (format "(~a * ~a)" v (show x))
      'show --> (format "~a" v)
      m --> (parent m)])))
```

Наследование в функциональных объектах

Объект с виртуальными методами

```
(define (Alg v)
  (define parent Obj)
  (/. 'algebraic? --> #t
      '+ x --> (format "(~a + ~a)" v (show x))
      '* x --> (format "(~a * ~a)" v (show x))
      'show --> (format "~a" v)
      m --> (parent m)]))))
```

```
> (define a (algebraic 'a))
> (define b (algebraic 'b))
> (show (a '+ b))
"(a + b)"
> (show ((a '+ b) '* 3))
"((a + b) * 3)""
```

Наследование в функциональных объектах

Объект с виртуальными методами

```
(define (Alg v)
  (define parent Obj)
  (/. 'algebraic? --> #t
      '+ x --> (format "(~a + ~a)" v (show x))
      '* x --> (format "(~a * ~a)" v (show x))
      'show --> (format "~a" v)
      m --> (parent m)])))
```

```
> (define a (algebraic 'a))
> (define b (algebraic 'b))
> (show (a '+ b))
"(a + b)"
> (show ((a '+ b) '* 3))
"((a + b) * 3)""
```

Наследник algebraic

```
(define complex
  (define parent (algebraic 'z)))
  (/. 'complex --> #t
      ...
      m --> (parent m))))
```

Подведение итогов

Что позволяют делать фундаментальные свойства функций:

- абстракция,
- аппликация,
- замыкание

  плюс

- декомпозиция,
- управляющие конструкции?

Подведение итогов

Что позволяют делать фундаментальные свойства функций:

- абстракция,
- аппликация,
- замыкание

плюс

- декомпозиция,
- управляющие конструкции?

Определять абстрактные типы данных:

```
(define-type (Comp Real Real))

(:: plus (Comp Comp -> Comp)
(define plus
  (/. (Comp r1 i1)
      (Comp r2 i2) --> (Comp (+ r1 r2)
                             (+ i1 i2)))))

(:: mult (Comp Comp -> Comp)
(define mult
  (/. (Comp r1 i1)
      (Comp r2 i2) --> (Comp (- (* r1 r2)
                                (* i1 i2))
                             (+ (* r1 i2)
                                (* r2 i1))))))
```

Подведение итогов

Что позволяют делать фундаментальные свойства функций:

- абстракция,
- аппликация,
- замыкание

плюс

- декомпозиция,
- управляющие конструкции?

Использовать обобщённые процедуры и операторы:

$$
\begin{aligned}
&\texttt{accum}\ g\ x_0\ f\ n = \texttt{F}\ n\\
&\quad \texttt{where F}\ 0 = x_0\\
&\qquad\qquad \texttt{F}\ i = g\ (f\ i)\ (\texttt{F}\ (i-1))
\end{aligned}
$$

$\texttt{sumf} = \texttt{accum}\ (+)\ 0$

$\texttt{prodf} = \texttt{accum}\ (*)\ 1$

$\texttt{sumsqr} = \texttt{sumf}\ (x \mapsto x^2)$

$\texttt{factorial} = \texttt{prodf}\ id$

$\texttt{power}\ x = \texttt{prodf}\ (i \mapsto x)$

$\texttt{table} = \texttt{accum}\ (:)\ [\,]$

$\texttt{make-list}\ x = \texttt{table}\ (i \mapsto x)$

Подведение итогов

Что позволяют делать фундаментальные свойства функций:

- абстракция,
- аппликация,
- замыкание

плюс

- декомпозиция,
- управляющие конструкции?

Определять объекты и ОО-системы:

```
(define (Comp r i)
  (define parent (algebraic))
  (/. 'complex? --> #t
      'Re --> r
      'Im --> i
      '+ x --> (Comp (+ (x 'Re) r)
                     (+ (x 'Im) i))
      '* x --> (Comp (- (* (x 'Re) r)
                        (* (x 'Im) i))
                     (+ (* (x 'Re) i)
                        (* (x 'Im) r)))
      'conj --> (Comp r (- i))
      'show --> (format "~a+~ai" r i))
      m --> (parent m))
```